Formal Verification of Stability in Provisioning Changes with BPlan

Eloi Perdereau✉[1][0009−0005−9257−5374], Daniel Sokolowski[3][0000−0003−2911−8304], Jolan Philippe[1][0000−0001−8759−4566], Hélène Coullon[2][0000−0003−2573−2147], and Guido Salvaneschi[4][0000−0002−9324−8894]

[1] Univ. Orléans, INSA CVL, LIFO, UR 4022
{eloi.perdereau,jolan.philippe1}@univ-orleans.fr
[2] IMT Atlantique, Inria, LS2N, UMR 6004 helene.coullon@imt-atlantique.fr
[3] Independent Researcher science@d.sokolowski.org
[4] University of St. Gallen guido.salvaneschi@unisg.ch

Abstract. With provisioning Infrastructure as Code (IaC) tools like Terraform, users declaratively define the target infrastructure state, and the IaC tool automatically plans and executes the required deployment actions to reconcile the target and actual state. However, ensuring the correctness and safety of such plans remains a challenge. For example, dependencies among infrastructure resources may not be explicit, leading to ambiguous execution orders, potential security vulnerabilities, or deployment instability. In this paper, we introduce a generic temporal property to verify the *stability* of a Terraform change. This property is leveraged by our solution entitled `BPlan` that complements Terraform plan by showing the precise action sequence and stability properties it implies. `BPlan` is based on a mechanized formalization of Terraform in Maude, capturing its core behavior. Our evaluation demonstrates the correctness of our semantics, the scalability and the impact and applicability of `BPlan` on real-world use cases.

Keywords: Terraform · Infrastructure as Code · Stability · Semantics · Maude · LTL

1 Introduction

Managing the infrastructure underlying software applications is a complex and, traditionally, manual task. Infrastructure as Code (IaC) [30] specifies the infrastructure configuration as a program, similar to traditional software. This way, infrastructure configuration becomes shareable, versioned, and testable, greatly simplifying the deployment and maintenance of applications.

Cloud providers offer APIs to manage infrastructure resources through create, read, update and delete (CRUD) actions [8]. Since such APIs are heterogeneous across and even within a single cloud provider, programmatically ensuring the correct execution of thousands of requests to such low-level APIs is

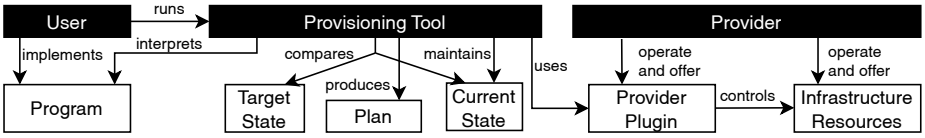


Fig. 1: Provisioning tool interaction overview.

difficult and error-prone. To address this challenge, *provisioning IaC tools* like Terraform [24], Pulumi [16], AWS CloudFormation [7], Azure Resource Manager [29] and OpenStack Heat [31], have become extremely popular. The advantage of relying on these systems is twofold. First, they provide a homogeneous, unified configuration surface across APIs, simplifying development. Second, they increase safety by offering a declarative approach which enables programmers to abstract over *change* operations. Instead of a sequence of actions, users define the *target state* of their infrastructure (dependencies and resource configurations) and the IaC infers a plan of CRUD actions to move the current infrastructure into the target state by interacting with the APIs. This plan is then executed with the actions being automatically scheduled in a valid order, e.g., constrained by dependencies between resources. For example, a storage service has to be created before the VM[5] querying the data within it. On the contrary, the storage has to be deleted before its security policies to prevent sensitive documents from being accessible by unauthorized entities.

Terraform and Pulumi, both multi-provider (or *multi-cloud*), abstract over the individual APIs of cloud providers. In this paper, we focus on Terraform (HashiCorp), Pulumi works similarly but has implementation differences not addressed in this study. Terraform is a widely used IaC provisioning tool that supports more than 3,000 integrations with more than 250 partners, including major cloud computing providers.[6] Note that all claims about Terraform should be directly applicable to its open-source fork OpenTofu [20]. Figure 1 (further detailed in Section 2) presents a schematic view of multi-provider provisioning tools and their interaction with users and providers. *Users*, e.g., a person, company, or automation, adopt a *provisioning tool* to write *programs* that provision (i.e., create, update, and delete) *infrastructure resources* like virtual networks, databases, and servers. Such resources are operated by *providers*, e.g., GCP[7] or AWS[8], which offer their resources to users through *provider plugins*, i.e., glue code between the provisioning tool and the provider APIs. As a result, users can access resources of multiple providers within the same codebase.

Terraform's `plan` command computes the differences between the current state and the specified target state. The command produces an unordered set of operations planned on each resource. Terraform's `apply` command then uses

[5] Virtual Machine
[6] `https://registry.terraform.io/browse/providers`
[7] Google Cloud Platform
[8] Amazon Web Services

the plan to schedule concrete actions to execute on resources. This two-phase approach allows users to examine and validate the generated plan. However, Terraform `plan` does not provide any information about the execution order of create, delete, and update actions, i.e., the *schedule*. Though, generally, some action orders can result in instabilities or vulnerabilities during changes, an issue recognized as intra-change safety concerns in the literature [27].

Our goal is to tackle this issue through the following contributions. First, we present a general stability temporal logic property to verify Terraform deployments (i.e., changes). This property intuitively verifies that, during a change, the state of the system is always correct (i.e., the change is stable), or if an incorrect state occurs we are sure that the fault does not persist after the end of the change (which is then eventually stable). In particular, `BPlan` implements two instances of our stability property via two state predicates: resources always have a unique instance (resource stability), and a resource never disappears when other resources depend on it (dependency stability). Second, we present `BPlan`, the first tool leveraging formal methods to complement Terraform `plan` by providing insight to the user into the execution order of actions and expected behavior of their infrastructure resources. To this purpose, `BPlan` is based on an operational semantics of Terraform's execution model (implemented in Maude) on which our stability property can be checked. Third, we implemented and evaluated `BPlan` to answer the following research questions: (**RQ1**) Can we reverse engineer Terraform to extract a semantics of its core behavior such that stability properties can be verified? (**RQ2**) To what extent can our verification methods be applied to IaC provisioning in terms of scalability? (**RQ3**) What is the potential impact of `BPlan` on the community in terms of deployment safety?

The rest of the paper is structured as follows. Section 2 details the motivation of our work through examples of Terraform programs, plans and executions. Section 3 defines our general temporal logic stability property and its instantiation to resource and dependency stability. Section 4 details the components of `BPlan` and their implementation. Section 5 evaluates the correctness of our semantics and the effectiveness of `BPlan` on synthetic case studies, as well as real-world use cases. Section 6 presents the related work and Section 7 concludes.

2 Background and Motivation

In Terraform, deployment programs are written in HCL[9] [22] whose native syntax consists of labelled blocks and attributes, i.e., key-value pairs. Various types of blocks are available but in essence, an HCL program is a set of manifests that declare a set of resource blocks encapsulating the configuration parameters and lifecycle behavior of resources defined by provider plugins. Listing 1.1 shows three resources from the GCP provider: a bucket (object storage), a VM and a security policy. Resource attributes are depicted in blue (`name`, `boot_disk`, `bucket`) and behavioral meta-arguments in dark green (`lifecycle`,

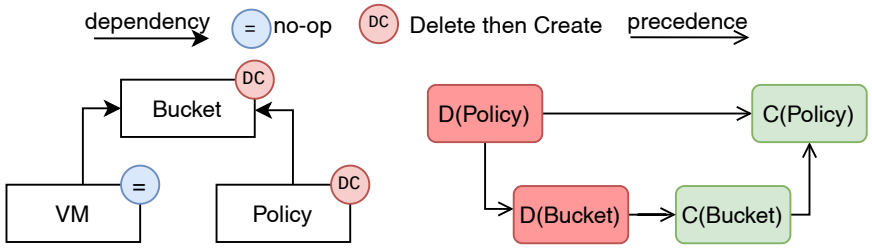
[9] HashiCorp Configuration Language

Listing 1.1: Subset of a deployment manifest in HCL.

```
resource "google_storage_bucket" "Bucket" {                          1.1
  name = "storage-bucket"                                            1.2
  lifecycle { create_before_destroy = false }                        1.3
}                                                                    1.4
resource "google_compute_instance" "VM" {                            1.5
  boot_disk { ... }                                                  1.6
  depends_on = [ google_storage_bucket.Bucket ]                      1.7
}                                                                    1.8
resource "google_storage_bucket_iam_member" "Policy" {               1.9
  bucket = google_storage_bucket.Bucket.name                         1.10
}                                                                    1.11
```

`create_before_destroy` – `cb4d` in short – and `depends_on`). Crucially, such a manifest specifies the *target state* as a dependency graph between resources. For example, Lines 1.7 and 1.10 make the VM and the policy depend on the bucket. In order to run a deployment program, the manifests are interpreted and the target state is compared with the *current state* of the infrastructure, producing a *plan*. A Terraform plan consists of the dependency graph along with the *operations* to be applied on each resource, either (`N`)othing, (`C`)reate only, (`D`)elete only or modify. The modify operation can either be a mutable (`U`)pdate or an immutable replacement, the latter involving two steps: `C-D` or `D-C`, i.e., (`C`)reate then (`D`)elete or the reverse. If the user confirms the plan, the set of operations on resources is applied. In this phase, Terraform *schedules* the concrete *actions* `C`, `U` and `D` and runs them in order, constrained by the dependencies. Only then the actions are *executed*: Terraform invokes the provider plugins that call their respective APIs to effectively apply changes to the infrastructure, and the current state is updated accordingly. Figure 2 depicts the results for Listing 1.1.

From a user perspective, Terraform surfaces the dependency graph and the operations in the plan. However, problematic action ordering may occur when the plan is confirmed, exposing the system to faults that are only discovered when changes are executed. For example, on the very simple example of Figure 2, the storage bucket is deleted while the VM is still running, and the policy is created after the bucket, thus exposing the system to downtime and attacks, respectively. A possible fix for the latter is to set `cb4d` to `true` (Line 1.3) in order to create a new bucket before deleting the old one. In this case however, two instances of the storage exist at the same time, which might induce unwanted costs. In practice, with tens or hundreds of resources in a Terraform program, the concrete ordering of actions is impossible to predict without appropriate tooling. The risks induced by a wrong action schedule such as availability issues, sensitive information leakage or high deployment costs, remain out of reach. Moreover, the practical need of checking actions schedule is illustrated by the relatively frequent deployment failures reported within the DevOps community. To build an empirical motivation, we leveraged Codex searches across platforms like GitHub, Reddit, and Terraform community forums. In 20 interactions with Codex 5.3, and during 126 minutes of search in total, 40 discussions issues have been identified and classified into groups that match `BPlan` scope: wrong order of actions



(a) Target state and planned operations, known before execution

(b) Action schedule, unknown before execution

Fig. 2: Dependency graph with planned operations and schedule for Listing 1.1

(i.e., Terraform runs two explicit actions in an unexpected order), downtime risk (i.e., the sequence is accepted, but it causes avoidable service disruption), missing detach (i.e., Terraform deletes an object before codependencies are detached or deleted), bad replacement order (i.e., the chosen replacement strategy is unsafe).

By rendering the action schedule explicit and checking for specified instabilities (in this paper: dangling dependencies and resource duplications), users are given visibility about the expected behavior of their deployments, thus allowing to balance tradeoffs, e.g., cost vs. availability, and prevent unreliable infrastructure changes. In addition, the formal semantics acts as a precise reasoning framework regarding the core behavior of Terraform, as opposed to the informal specifications. This helps to identify the main points of what constitutes an IaC tool, enables the ability to prove certain properties on the executions, and gives implementors the keys to build more robust tooling.

3 Intra-Change Stability Property

This section defines stability properties of a Terraform execution. It is, as far as we know, the first formalization of intra-change safety in provisioning tools.

3.1 Stability Definitions and Model Checking

We start by defining an *execution* as m intermediate steps of a rewriting relation.

Definition 1 (Execution). *For an initial configuration C_0, an execution is a terminating sequence of configurations $[C_0, C_1, \cdots, C_m]$ s.t., $C_{i-1} \longrightarrow C_i$ where $1 \leq i \leq m$ and $\longrightarrow$ denotes a step of the operational semantics (c.f., Section 4).*

In general terms, the stability of a system depends on its ability to maintain a legitimate state during its execution, potentially recovering from faulty states, i.e., transient failures. We define stability in a generic manner by supposing a

faulty predicate P defined for any configuration. As model-checkers proceed in an optimized but exhaustive exploration of possible futures, using faulty predicates is a common technique to quickly verify if there is a safety issue in a system.

Definition 2 (Execution Stability). *An execution* $[C_0, \ldots, C_m]$ *is* stable *iff* P *is false for all* $0 \leq i \leq m$*, and* eventually stable *iff there is a step* $0 \leq i < m$ *where* P *is true but in the final step* m*,* P *is false. Otherwise, the execution is* perpetually unstable*, i.e.,* P *is true at step* m*.*

In words, an execution is stable if it never reaches a faulty configuration, it is eventually stable if it recovers from a faulty configuration, and it is perpetually unstable if the final state is faulty. We reformulate the above properties in the LTL[10] formalism so that it is easy to express them in a model-checker. Recall the notations where $\Box F$ means F holds in the current and all successive steps and $\Diamond F$ means F will eventually hold at some point in future steps. The property on the final configuration m is materialized as occurring infinitely often.

Definition 3 (LTL Stability).

$$\begin{aligned} \textsc{stable}_P &= \Box\neg P \\ \textsc{e-stable}_P &= \Diamond P \wedge \Diamond \textsc{stable}_P \\ \textsc{p-unstable}_P &= \neg(\textsc{stable}_P \vee \textsc{e-stable}_P) = \Box\Diamond P \end{aligned}$$

Since we are interested in capturing change instabilities, it is enough to check STABLE on the initial configuration C_0: the model-checker will terminate as soon as a faulty configuration is found and will provide an unstable counterexample execution, or it will ensure that all executions are stable. It is nonetheless relevant to distinguish between the two instability properties. First, E-STABLE formally captures intra-update vulnerability issues [27], and it is the goal of `BPlan` to report such problems. Second, P-UNSTABLE may serve as a correctness criterion for Terraform itself as it captures tragic executions that leave the infrastructure in a persistent problematic state. Model-checking ¬P-UNSTABLE ensures that all executions are either STABLE or E-STABLE because the three properties are mutually exclusive. Fortunately, running this check on our systematic testbed (c.f., Section 5) revealed no such perpetual instability. We can thus assume that all counterexamples of STABLE are in fact E-STABLE.

3.2 Resource and Dependency Stability

In this paper, we exemplify our intra-change stability property by using two different faulty predicates that we specify after a few definitions. Let $\mathcal{R}$ be the set of *resources* and $\mathcal{R}_{\texttt{inst}} = \mathcal{R} \times \{\texttt{new}, \texttt{old}\}$ the set of resource *instances*, i.e., currently deployed. No more than two instances of a resource may exist for a resource because planned operations are limited to at most one create and one delete per resource (c.f., Definition 8). For a dependency arc (r, r'), r is a *codependency* of r', and r' a *dependency* of r. The context should clarify whether dependency refers to the arc itself or its destination node.

[10] Linear Temporal Logic

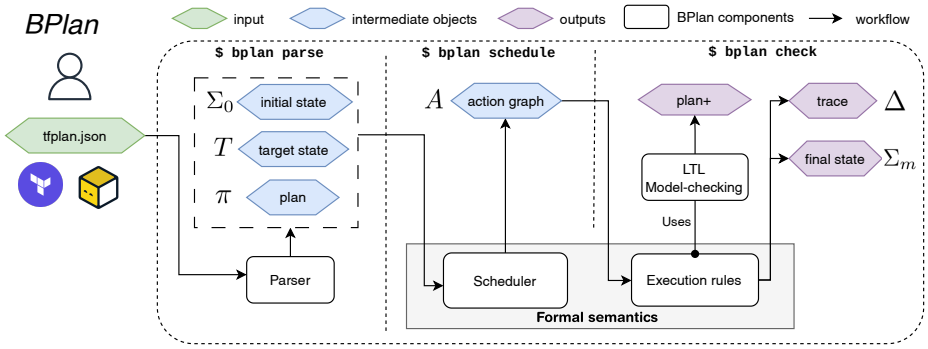


Fig. 3: Internals of `BPlan`.

Definition 4 (Current State Graph). *The current state $\Sigma = (R_\Sigma, D_\Sigma)$ holds the graph of resource instances and their dependencies. $R_\Sigma \subseteq \mathcal{R}_{inst}$ is the set of currently deployed resource instances and $D_\Sigma \subseteq \mathcal{R}^2_{inst}$ their dependencies.*

We can now state our predicates. One aims at avoiding resource duplications, the other aims at avoiding the deletion of a resource that others depend on.

Definition 5 (Duplicate Resource). *A resource is said duplicated if both its instances (`new` and `old`) coexist, i.e., are present at the same time in the same configuration. Formally, for a resource $r \in \mathcal{R}$,*

$$duplicate(r) = r_{\texttt{old}} \in R_\Sigma \wedge r_{\texttt{new}} \in R_\Sigma$$

Definition 6 (Dangling Dependency). *A dependency between two resources is said dangling if the arc linking their instances exists, but the target instance is absent from the configuration. Formally, for a dependency $d = (r, r')$,*

$$dangling(d) = \exists (r_\sigma, r'_{\sigma'}) \in D_\Sigma \wedge r'_{\sigma'} \notin R_\Sigma$$

We simply write *duplicate* and *dangling* the extensions that generalize to any resource and dependency, respectively. By replacing P with these predicates in the LTL formulas, we obtain varying stability properties characterizing the safety of infrastructure changes. In the example of Figure 2 and Listing 1.1, $dangling(VM, Bucket)$ holds, and $duplicate(Bucket)$ holds if `cb4d` is set to `true`.

In the next section, we explain how `BPlan` exploits our stability property.

4 Formal Semantics and `BPlan`

Our formal semantics and stability properties are integrated in a tool: `BPlan`. The goal of `BPlan` is to help Terraform users detect potential vulnerabilities in their changes at the planning phase, ahead of execution. Users submit a Terraform plan and `BPlan` outputs the schedule of actions and an augmented plan showing

Name	Notation
Resource	$r \in \mathcal{R}$
Instance	$\sigma \in \{\texttt{new}, \texttt{old}\}$
Resource instance	$r_\sigma \in \mathcal{R}_{\texttt{inst}}$
Plain action	$a \in \mathcal{A}$
Action on resource	$a(r) \in \mathcal{A} \times \mathcal{R}$
Trace action	$\uparrow a(r), \downarrow a(r) \in \{\uparrow, \downarrow\} \times \mathcal{A} \times \mathcal{R}$
Target state graph	$T = (R_T, D_T)$ with $R_T \subseteq \mathcal{R}$ and $D_T \subseteq \mathcal{R}^2$
Current state graph	$\Sigma = (R_\Sigma, D_\Sigma)$ with $R_\Sigma \subseteq \mathcal{R}_{\texttt{inst}}$ and $D_\Sigma \subseteq \mathcal{R}^2_{\texttt{inst}}$
Planned operations	$\pi \subseteq \mathcal{R} \times \mathcal{A}^*$
Action graph	$A = (V_A, E_A)$ with $V_A \subseteq \mathcal{A} \times \mathcal{R}$ and $E_A \subseteq {V_A}^2$
Ongoing actions	$\mathcal{Q} \subseteq \mathcal{A} \times \mathcal{R}$
Action trace	$\Delta \in (\{\uparrow, \downarrow\} \times \mathcal{A} \times \mathcal{R})^*$

Fig. 4: Notations summary

the instabilities of dependencies (i.e., *dangling*) and resources (i.e., *duplicate*). The `BPlan` tool is written in Python and the semantics is implemented in Maude [15]. Figure 3 depicts the internal architecture of `BPlan`. In this section, we gradually define the semantics of Terraform by explaining each of the components of `BPlan`. All notations are summarized in Figure 4.

4.1 Parser

As illustrated in Figure 3, the parser takes as input the JSON result of Terraform `plan` and creates three objects: Σ_0, T, and π. In the initial current state Σ_0 (c.f., Definition 4) all instances are `old`, representing the resources that preexist. Moreover, Figure 2a depicts an example of a target state T with the planned operations of π attached to resources. The formal definitions follow.

Definition 7 (Target State Graph). *The target state $T = (R_T, D_T)$ holds the state of resources and their dependencies as specified by the user within a Terraform module. $R_T \subseteq \mathcal{R}$ is the set of target resources and $D_T \subseteq \mathcal{R}^2$ the set of target dependencies.*

Let $\mathcal{A} = \{\texttt{C}, \texttt{U}, \texttt{D}\}$ be the set of plain actions, noted $a, a' \in \mathcal{A}$.

Definition 8 (Planned Operations). *The set of planned operations $\pi \subseteq \mathcal{R} \times \mathcal{A}^*$ maintains the sequence of plain actions to execute for deploying each resource. Elements of π are pairs $(r, [a_1, \ldots, a_n])$ where $[a_1, \ldots, a_n]$ is a sequence of actions (the operation) and r a resource. According to Terraform `plan`'s semantics, the operation can either be a single action (i.e., $[\texttt{C}]$, $[\texttt{U}]$, or $[\texttt{D}]$), or the two actions of a replacement, namely $[\texttt{C}, \texttt{D}]$ or $[\texttt{D}, \texttt{C}]$.*

The JSON plan given by Terraform does not expose the resources and their dependencies from submodule calls. Also, the resources' deployment does not seem to intertwine with parent modules' deployments. For this reason, we limit

`BPlan` to considering one module at a time. Moreover, Terraform allows users to specify multiple resources at once in the same `resource` block through the use of `count` and `for_each` meta-arguments. As a first approximation, `BPlan` suppose that either they are absent or they result in exactly one resource.

4.2 Scheduler

The scheduler computes the *action graph* (or schedule). An example is given in Figure 2b.

Definition 9 (Action Graph). *The action graph $A = (V_A, E_A)$ maintains the schedule of actions. Vertices $V_A \subseteq \mathcal{A} \times \mathcal{R}$ are actions on resources, noted $a(r)$. Arcs $E_A \subseteq {V_A}^2$ denote precedence, i.e., $a(r)$ before $a'(r')$ for $(a(r), a'(r'))$.*

$$
\begin{aligned}
&\mathit{deploys}(r, \pi) = \{a_i(r) \mid (r, [a_1, \ldots, a_k]) \in \pi, i \in [1, \ldots, k], a_i \in \{\texttt{C}, \texttt{U}\}\} \\
&\mathit{deletes}(r, \pi) \ = \{\texttt{D}(r) \mid (r, [a_1, \ldots, a_k]) \in \pi, i \in [1, \ldots, k], a_i = \texttt{D}\} \\
&\mathit{orderDeploys}(D_T, \pi) = \cup\{\mathit{deploys}(r', \pi) \times \mathit{deploys}(r, \pi) \mid (r, r') \in D_T\} \\
&\mathit{orderDeletes}(D_T, D_\Sigma, \pi) = \\
&\qquad \cup \{\mathit{deletes}(r, \pi) \times \mathit{deletes}(r', \pi) \mid (r, r') \in D_T \vee (r_\sigma, r'_{\sigma'}) \in D_\Sigma\} \\
&\mathit{actionGraph}(T, \Sigma, \pi) = (V_A, E_A) \text{ where} \\
&\quad V_A = \{a_i(r) \mid (r, [a_1, \ldots, a_k]) \in \pi, 1 \leq i \leq k\} \text{ and} \\
&\quad E_A = \{(a(r), a'(r)) \mid (r, [a, a']) \in \pi\} \\
&\qquad \cup \mathit{orderDeploys}(D_T, \pi) \cup \mathit{orderDeletes}(D_T, D_\Sigma, \pi)
\end{aligned}
$$

Fig. 5: Scheduling functions that produce the action graph A.

There are three general principles that govern the scheduling order. First, the sequence of actions planned for a resource must be executed in order, i.e., a_i before a_{i+1}. Second, the `C` and `U` actions of a resource must be executed before the `C` and `U` of its codependencies. For example, in Figure 2b, it is only after the Bucket recreation that the Policy can be created. Third, `D` actions are ordered in reverse compared to `C` and `U` actions: the deletion of a codependency must precede the deletion of the resources it depends on. In Figure 2b, the Bucket is deleted after the deletion of the Policy. Figure 5 formalizes the action graph computation according to these principles. Note that if a resource is planned for $[\texttt{C}, \texttt{D}]$ (i.e., `cb4d = true`) and has a dependency that is planned for $[\texttt{D}, \texttt{C}]$, the resulting action graph would be cyclic. However, Terraform prevents this scenario from happening by forcing the `cb4d` property to apply to dependencies that would otherwise be replaced with $[\texttt{D}, \texttt{C}]$ [25].

4.3 Checker

For the verification, the model-checker applies a set of rewriting rules (corresponding to $\longrightarrow$ in Definition 1) in order to execute the actions w.r.t. the partial order given by the schedule A, resulting in a sequence of events.

Definition 10 (Trace of Actions). *The action trace* $\Delta \in (\{\uparrow, \downarrow\} \times \mathcal{A} \times \mathcal{R})^*$ *maintains the sequence of starting or ending action events, respectively denoted* $\uparrow a(r)$ *and* $\downarrow a(r)$*.*

The execution thus orders the start and the end of each action, allowing actions to be executed in parallel while logging the strict order of events. In Figure 2b there is no parallelism and the trace is $[\uparrow \texttt{D}(Policy), \downarrow \texttt{D}(Policy), \uparrow \texttt{D}(Bucket), \ldots, \uparrow \texttt{C}(Policy), \downarrow \texttt{C}(Policy)]$, but in general the rules are non-deterministic. The trace Δ is notably used to validate the correctness of the semantics w.r.t. Terraform's trace in Section 5. In addition, we define *ongoing actions* that serve as a pivot between the action graph and the sequential trace.

Definition 11 (Ongoing Actions). *The set of ongoing actions* $\mathcal{Q} \subseteq \mathcal{A} \times \mathcal{R}$ *maintains the actions currently being executed, i.e., the action has started but not ended. Its elements are noted* $a(r)$*, similar to the nodes of the action graph.*

The last definition defines the main data structure that is updated according to the rules of our small-step operational semantics, shown in Figure 6.

Definition 12 (Configuration). *A configuration* C *is a tuple of four objects* $(\Sigma, A, \mathcal{Q}, \Delta)$ *following the above definitions. The initial configuration* C_0 *is* $(\Sigma_0, A_0, \emptyset, [])$ *with* Σ_0 *the initial state,* A_0 *the action graph computed by the scheduler, an empty set of ongoing actions and an empty trace sequence.*

In Figure 6 we write $D_T(r)$ and $D_\Sigma(r_\sigma)$ for the set of dependencies of r and r_σ respectively in the target or current state, i.e., $D_T(r) = \{r' \mid (r, r') \in D_T\}$ and equivalently for $D_\Sigma(r_\sigma)$. Moreover, $A \setminus a(r)$ is used as a shorthand for the removal of an action node $a(r)$ and its associated arcs, i.e., $A \setminus a(r) = (V_A \setminus \{a(r)\}, E_A \setminus \{a(r) \mid (_, a(r)) \vee (a(r), _)\})$, the underscore is used as a placeholder for any object. Also, we use $\Delta \oplus [e]$ to append an event e (either $\uparrow a(r)$ or $\downarrow a(r)$) to the trace Δ. Finally, we define $newest(R, R_\Sigma)$ as the set of newest instances of R present in R_Σ, i.e., `new`, or `old` if `new` does not exist.

The rules are separated in one generic *Start* and three *End* rules, one for each plain action. The *Start* rule copies an action $a(r) \in V_A$ with no preceding actions into $\mathcal{Q}$ and appends the starting event to Δ. Each *End* rule removes a common action from both $\mathcal{Q}$ and A, logs the corresponding ending event $\downarrow a(r)$ in Δ and updates the current state Σ. The state update involves the manipulation of dependencies between resources that are currently being deployed.

EndCreate adds the resource instance $r_{\texttt{new}}$ to R_Σ and updates all dependency arcs involving r by replacing $r_{\texttt{old}}$ with $r_{\texttt{new}}$. *EndUpdate* keeps the resource $r_{\texttt{old}}$ in R_Σ but updates its dependencies to their newest instances currently found in the state Σ. Finally, *EndDelete* simply removes $r_{\texttt{old}}$ from R_Σ and all dependency arcs originating from it. Resources that depend on $r_{\texttt{old}}$ are not updated, potentially leaving dangling dependencies, i.e., arcs pointing towards non-existent resources.

5 Evaluation

In this section, we first validate the correctness of our formal semantics. Second, we study the scalability of `BPlan`. Finally, we study the applicability of `BPlan`.

$$\frac{a(r) \in V_A \qquad \nexists(_, a(r)) \in E_A \qquad a(r) \notin \mathcal{Q}}{\Sigma, A, \mathcal{Q}, \Delta \longrightarrow \Sigma, A, \mathcal{Q} \cup \{a(r)\}, \Delta \oplus [\uparrow a(r)]}[StartAction]$$

$$\frac{\begin{array}{c}\texttt{C}(r) \in \mathcal{Q} \qquad R'_\Sigma = R_\Sigma \cup \{r_{\texttt{new}}\} \qquad D'_\Sigma = (D_\Sigma \setminus D_{\texttt{old}}) \cup D_{\texttt{new}} \\ D_{\texttt{old}} = \{(r_{\texttt{old}}, _) \in D_\Sigma\} \cup \{(_, r_{\texttt{old}}) \in D_\Sigma\} \\ D_{\texttt{new}} = (\{r_{\texttt{new}}\} \times newest(D_T(r), R_\Sigma)) \cup \{(r'_\sigma, r_{\texttt{new}}) \mid (r'_\sigma, r_{\texttt{old}}) \in D_\Sigma\}\end{array}}{(R_\Sigma, D_\Sigma), A, \mathcal{Q}, \Delta \longrightarrow (R'_\Sigma, D'_\Sigma), A \setminus \texttt{C}(r), \mathcal{Q} \setminus \{\texttt{C}(r)\}, \Delta \oplus [\downarrow \texttt{C}(r)]}[EndCreate]$$

$$\frac{\begin{array}{c}\texttt{U}(r) \in \mathcal{Q} \\ D'_\Sigma = (D_\Sigma \setminus (\{r_{\texttt{old}}\} \times D_\Sigma(r_{\texttt{old}})) \cup (\{r_{\texttt{old}}\} \times newest(D_T(r), R_\Sigma))\end{array}}{(R_\Sigma, D_\Sigma), A, \mathcal{Q}, \Delta \longrightarrow (R_\Sigma, D'_\Sigma), A \setminus \texttt{U}(r), \mathcal{Q} \setminus \{\texttt{U}(r)\}, \Delta \oplus [\downarrow \texttt{U}(r)]}[EndUpdate]$$

$$\frac{\texttt{D}(r) \in \mathcal{Q} \qquad R'_\Sigma = R_\Sigma \setminus \{r_{\texttt{old}}\} \qquad D'_\Sigma = D_\Sigma \setminus (\{r_{\texttt{old}}\} \times D_\Sigma(r))}{(R_\Sigma, D_\Sigma), A, \mathcal{Q}, \Delta \longrightarrow (R'_\Sigma, D'_\Sigma), A \setminus \texttt{D}(r), \mathcal{Q} \setminus \{\texttt{D}(r)\}, \Delta \oplus [\downarrow \texttt{D}(r)]}[EndDelete]$$

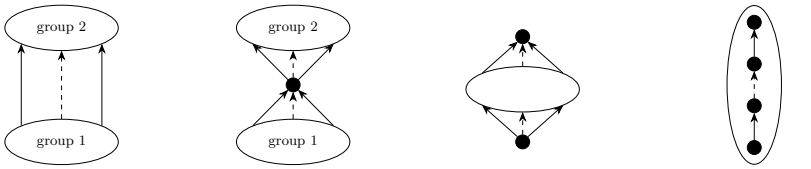
Fig. 6: Execution rules, $\longrightarrow$ rewrites a configuration $(\Sigma, A, \mathcal{Q}, \Delta)$ in one step.

Note that all experiments conducted in this section focus on the *dangling* predicate which is more challenging and difficult to check in general than *duplicate*.

5.1 Correctness

Through a systematic testbed, we give empirical evidence of the correctness of the action schedule (Figure 5) and the execution rules (Figure 6), w.r.t. Terraform's behavior, by considering the trace of all possible change scenarios for two, to four resources, resulting in more than 10k test cases. Since our property is affected by replacements and updates rather than addition or deletion operations, we only consider changes whose target and initial state graphs coincide. Moreover, in order to have full control on the planned operations and to avoid provider specific behaviors, we implement our own provider for which we control the replacement or update operations when planning a change. For each dependency graph, we generate two corresponding HCL manifests: one for the initial deployment and one for the actual change. We then execute Terraform `apply` on the former to produce an initial state, then, according to each combination of operations, we `plan` and `apply` on the latter and record the trace sequence.

First, to validate our action graph, we implement an algorithm that checks if the Terraform trace is valid according to the action graph A. Taking into account that each action $a(r)$ in A is composed of one start $\uparrow a(r)$ and one end $\downarrow a(r)$ event in the trace, we check that each action starts after all its preceding actions have ended. Second, to validate the correctness of our execution rules, we leverage Maude's `search(`C_0`, `$C^\star$`)` command that applies the rules on an initial configuration C_0, looking for a configuration C_i that matches $C^\star$. By filling $C^\star$ with Terraform's trace, the search will succeed if there is an execution that leads to the given trace. **RQ1: Except for four pathological cases in the latter**



(a) Bipartite (b) Butterfly (c) Diamond (d) Chain

Fig. 7: Topologies of scalability test cases. Dashed arrows symbolize that the number of dependencies vary according to the number of resources.

`search` validation, the `BPlan` traces all 10k test cases were checked valid compared to Terraform executions.

5.2 Scalability

To complement our previous strategy that checks all possible topologies, limited to only a small number of resources, we generate a scalability testbed following four topologies (depicted in Figure 7) that represent different dependency patterns between a varying number of resources: a *bipartite* topology with two groups connected one-to-one (7a); a *butterfly* topology following an n–1–n pattern (7b); a *diamond* topology with 1–n–1 dependencies (7c); and a *chain* topology forming a linear one-to-one sequence (7d). Each topology is instantiated with 5 to 50 resources with varying planned operations, covering well the possible cases. The Terraform traces are produced as explained in the previous section and each trace was validated correct according to the action graph. As for scalability, we run `BPlan` on each case to check the dependency stability $\text{STABLE}_{dangling}$ with the LTL model-checker of Maude, and we measure the execution time with a timeout set to 15 minutes (900s). Recall that P-UNSTABLE executions never happen, so a counterexample of STABLE corresponds to an E-STABLE execution. The results for each topology and for the systematic testbed are summarized in Table 1. The first three columns respectively show the topology, the number of resources and the total number of cases considered. The next three columns show the repartition for the different outcomes: a timeout, a positive result (all executions are STABLE) or an E-STABLE counterexample. The last two columns show the average execution times when the check did not time out, an empty cell indicates that all cases timed out.

As expected, globally the scalability of model-checking is limited, especially, no cases are checked STABLE when the number of resources is greater than 25 because the model-checker has to examine all possible executions when the result is positive. However, the different topologies show interesting variations, we notably see a clear distinction between {butterfly,chain} and {bipartite,diamond}. The E-STABLE cases with the latter topologies are checked rapidly and consist of a substantial percentage of all cases (e.g., 75% of bipartite cases with $[35, 40]$

Topology	Size	#Cases	#Timeouts	#STABLE	#E-STABLE	Avg STABLE	Avg E-STABLE
bipartite	$[5, 10]$	48	18.8%	43.8%	37.5%	0.21	0.02
	$[15, 20]$	48	52.1%	4.2%	43.8%	0.49	0.08
	$[25, 30]$	48	35.4%	0.0%	64.6%		0.34
	$[35, 40]$	24	25.0%	0.0%	75.0%		1.02
	$[45, 50]$	24	29.2%	0.0%	70.8%		3.21
butterfly	$[5, 10]$	192	34.4%	43.2%	22.4%	14.59	0.05
	$[15, 20]$	192	82.3%	2.6%	15.1%	20.35	2.48
	$[25, 30]$	192	91.7%	0.0%	8.3%		36.28
	$[35, 40]$	96	96.9%	0.0%	3.1%		373.09
	$[45, 50]$	96	100.0%	0.0%	0.0%		
chain	$[5, 10]$	48	12.5%	50.0%	37.5%	17.50	0.08
	$[15, 20]$	48	64.6%	4.2%	31.3%	50.75	8.14
	$[25, 30]$	48	50.0%	0.0%	50.0%		181.43
	$[35, 40]$	24	100.0%	0.0%	0.0%		
	$[45, 50]$	24	100.0%	0.0%	0.0%		
diamond	$[5, 10]$	256	42.6%	16.4%	41.0%	114.69	0.06
	$[15, 20]$	256	51.2%	2.0%	46.9%	154.96	0.64
	$[25, 30]$	256	52.0%	0.0%	48.0%		2.81
	$[35, 40]$	128	50.0%	0.0%	50.0%		10.06
	$[45, 50]$	128	50.0%	0.0%	50.0%		23.04
systematic	$n = 2$	15	0.0%	86.7%	13.3%	0.01	0.01
	$n = 3$	252	0.0%	80.2%	19.8%	0.19	0.02
	$n = 4$	10,455	0.1%	75.1%	24.8%	4.17	0.02

Table 1: Stability results per topology and systematic cases.

resources are unstable and checked in ∼1s), suggesting that almost all instabilities are successfully reported. On the contrary, {butterfly,chain} are unable to scale reasonably when larger than 15 resources, even for unstable cases. **RQ2: The execution time heavily depends on the topology at stake and the final result, but bipartite and diamond topologies scale better.** Moreover, **the percentage of instability is relatively important in all testbeds and increases with the number of resources (RQ3)**.

We also set to estimate which proportion of Terraform modules we are able to check with `BPlan`. For this purpose, we rely on the TerraDS dataset [12], consisting of 279,344 publicly available Terraform modules spread amongst 67,360 repositories. By counting the total number of modules according to their sizes (Table 2), we see that a majority of modules have less than 10 resources. Also, Terraform users are encouraged to build small reusable modules [23], suggesting that our approach is able to check a large proportion of modules in general.

Finally, note that despite the limitations on the model-checking component of `BPlan`, the computation of the action graph can be computed efficiently, thus the expected schedule of actions is available for very large cases.

$[1, 10]$	$[11, 20]$	$[21, 30]$	$[31, 40]$	≥ 41
82.8%	10.8%	3.4%	1.3%	1.7%

Table 2: Percentage of TerraDS modules different sizes.

Module		Change		
Link	#Resources	#Operations $\neq$ `N`	STABLE	E-STABLE
[2]	7	3	13.96s	0.02s
[3]	8	4	7.97s	0.01s
[1]	12	6	627.24s	0.02s
[4]	33	7	771.91s	0.13s
[5]	49	4	1111.20s	0.53s

Table 3: Modules and changes selected from TerraDS.

5.3 Impact and Applicability of `BPlan`

We investigate the volumetry of potential instabilities in Terraform modules. We explore the repositories in TerraDS and extract commits that might produce unstable dependencies. Looking for such instabilities in TerraDS is notably difficult and partial. We thus adopt a heuristic, based on the following observation: for a dependency (r, r'), `BPlan` reports an instability when the planned operation is $[\texttt{D}, \texttt{C}]$ for r' and either $[\texttt{N}]$ or $[\texttt{U}]$ for r. Indeed since r is not deleted but the operation on r' must occur before r's, (r, r') is *dangling* between the $\texttt{D}(r')$ and $\texttt{C}(r')$ actions. This scenario helps to understand where instabilities might occur for a given plan. Consequently, we scrape all commits that affect an attribute marked as requiring a replacement when changed.[11]. **RQ3: For 169,894 modules in TerraDS (filtered for common providers[12]) and 481,908 commits on associated Git repositories, 4,808 modules (2.83%) contain at least one problematic commit, representing a total of 11,287 (2.24%) problematic commits**. If these numbers are relatively low percentages, recall that: (1) they only capture a subset of possible reasons for instabilities; (2) they only represent committed issues; (3) highly critical infrastructures are potentially concerned, for example [1].

Finally, among the 4,808 modules containing problematic commits, we select five that we adapt to make them applicable (hidden resources, input variables, etc.). Table 3 summarizes a STABLE and an E-STABLE change for each of them. **RQ2 and RQ3: The execution times for the unstable changes confirm that such changes are checked rapidly. This illustrates that `BPlan`'s model-checking is a viable approach in order to provide an early-warning when eventual stabilities may happen**.

[11] Set as *RequiresReplace*/*ForceNew* in the provider's schema.
[12] GCP, AWS, Azure, Kubernetes

6 Related Work

Existing work on Infrastructure as Code formalization mainly targets *configuration management* rather than *provisioning*. Anderson et al. [9] formalize the SmartFrog configuration language to verify properties such as termination and program equivalence, while μPuppet [21] provides an operational semantics for a declarative subset of Puppet. However, these approaches focus on language constructs, compilation, and reconfiguration semantics, rather than on resource provisioning or the ordering of CRUD actions in declarative IaC.

Formal models for dynamic reconfiguration of component-based distributed systems [17] are also related to IaC. Concerto [13] and Aeolus [19] model the lifecycle of software service-oriented entities, and their associated planners rely on formal methods [32,18,6] to synthesize correct-by-construction reconfiguration plans. These approaches address complex coordination problems over programmable lifecycles, whereas `BPlan` aims to reverse-engineer and check Terraform's existing CRUD-based planning and scheduling logic. Similarly, the Cloud modeling specification language TOSCA has been formalized to reason about cloud-application orchestration [11,35]. For instance, Chareonsuk et al. [14] use the SPIN model-checker to verify orchestration workflows. In contrast, `BPlan` targets Terraform plans directly and checks intra-change properties induced by Terraform's own execution semantics.

In the Kubernetes ecosystem, several works study the correctness of controllers, which reconcile the actual state of cluster resources with a desired state. Sieve [33] and Anvil [34] use automated testing and formal techniques to detect reconciliation bugs in controller implementations. This differs from `BPlan`: Terraform also performs reconciliation, but `BPlan` does not aim to verify that the final state is correct; instead, it checks whether the infrastructure remains stable during the change. The closest comparison is Kivi [28], which uses SPIN to detect unsafe future states and analyzes oscillations caused by interactions between Kubernetes controllers. Although oscillation is a liveness concern, like our stability property, it captures a different phenomenon: instability of the cluster state due to concurrent controllers, rather than instability of cloud-resource dependencies or resource multiplicity during a Terraform change.

In [27], Häyhä identifies potential intra-update vulnerabilities in CloudFormation. Our approach is based on a similar idea but goes further by introducing a model-checking-based solution that enhances intra-update safety. Indeed, if potential security vulnerabilities during changes ("intra-changes") are introduced, no concrete solutions are given to detect such vulnerabilities and no concrete properties are formulated nor checked. As far as we know, we are the first to propose a solution to formally check such intra-change properties by examining Terraform codes. Regarding IaC code quality, efforts to enhance Terraform module quality include the static-analysis study of Terraform alerts by Hu et al. [26] and the metric-based evaluation approach offered by TerraMetrics [10]. While these works focus on the practitioner's perspective and the structural interface of the code, they differ from `BPlan`, which performs a formal analysis of the underlying semantics of Terraform.

7 Conclusion

Infrastructure as Code enables to specify cloud infrastructure programmatically. Yet, while declarative provisioning tools like Terraform and OpenTofu are becoming increasingly popular, ensuring the correctness of their execution remains an open challenge. In this paper, we introduced `BPlan` that embeds a mechanized semantics of Terraform executions in Maude and computes augmented outputs compared to the Terraform `plan` command: an action graph representing the execution schedule, and a stability-augmented plan. To check the stability, we defined a generic LTL property and leveraged Maude's model-checker.

Our results show the correctness of our formalization by running Terraform on multiple testbeds and validating the resulting trace w.r.t. our semantics. We demonstrate that the model-checking approach is able to efficiently check eventual stability, thus capturing problematic executions in a preliminary phase. Tests on the TerraDS dataset [12] illustrate that most modules have a size compatible with being checked by `BPlan`. Furthermore, we have illustrated the potential impact of `BPlan` through an extraction of commits in TerraDS.

This work opens various perspectives for future research. First, the semantics can be extended to account for the multiplicity of resources and submodules. Both introduce a group semantics for resources and it is unclear how such groups are scheduled relative to other resources. Second, our experiments do not consider cases where resources are added or removed. It is a challenge to reason about such changes, especially when dependencies (i.e., arcs) are consequently added or removed during the change. Third, our semantics allows precise reasoning on Terraform's behavior, enabling the development of heuristics or approximation algorithms that would improve the scalability of verifications. Finally, generalizing our semantics to other provisioning tools will open verification to a larger provisioning ecosystem. This would allow to identify which parts of the semantics are common to all provisioning tools and which parts are specific.

Data Availability Statement The codebase of `BPlan`, scripts and data used for processing the experiments, the search for case studies on TerraDS and AI-assisted search are publicly available at `https://doi.org/10.5281/zenodo.22747162`.

Acknowledgments This work was supported by a French government grant managed by the Agence Nationale de la Recherche (ANR) under the France 2030 program, under grant "ANR-23-PECL-0008" (Taranis project), by the Swiss National Science Foundation (SNSF, Grant No. 10001777), by ArmaSuisse Science and Technology, and by the European Union's Horizon research and innovation program (CAPE Project, Grant No. 101189899).

Disclosure of Interests The authors have no competing interests to declare.

References

1. https://github.com/alphagov/govuk-infrastructure/terraform/deployments/vpc
2. https://github.com/Crown-Commercial-Service/digitalmarketplace-aws/terraform/modules/router
3. https://github.com/ploio/hyperion-k8s/terraform/openstack
4. https://github.com/smarla/core-network/infra/terraform
5. https://github.com/Laxman-SM/terraform-mesos-aws/modules/demo/network/vpc
6. Ábrahám, E., Corzilius, F., Johnsen, E.B., Kremer, G., Mauro, J.: Zephyrus2: On the fly deployment optimization using smt and cp technologies. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) Dependable Software Engineering: Theories, Tools, and Applications. pp. 229–245. Springer International Publishing, Cham (2016)
7. Amazon: Aws cloud formation. https://aws.amazon.com/cloudformation/
8. Amazon Web Services: AWS Cloud Control API Documentation (2025), https://docs.aws.amazon.com/cloudcontrolapi/, accessed: 2025-02-13
9. Anderson, P., Herry, H.: A formal semantics for the smartfrog configuration language. Journal of Network and Systems Management **24**, 309–345 (2016). https://doi.org/10.1007/s10922-015-9351-y
10. Begoug, M., Chouchen, M., Ouni, A.: TerraMetrics: An open source tool for infrastructure-as-code (iac) quality metrics in terraform. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. p. 450–454. ICPC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3643916.3644439
11. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: Service Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015, Proceedings 4. vol. 9306, pp. 19–33. Springer (2015). https://doi.org/10.1007/978-3-319-24072-5_2
12. Bühler, C., Spielmann, D., Meier, R., Salvaneschi, G.: TerraDS: A dataset for terraform hcl programs. In: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR). pp. 654–658 (2025). https://doi.org/10.1109/MSR66628.2025.00101
13. Chardet, M., Coullon, H., Robillard, S.: Toward safe and efficient reconfiguration with concerto. Science of Computer Programming **203**, 102582 (2021). https://doi.org/10.1016/j.scico.2020.102582
14. Chareonsuk, W., Vatanawood, W.: Formal verification of cloud orchestration design with TOSCA and BPEL. In: 2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON). pp. 1–5. IEEE (2016). https://doi.org/10.1109/ECTICon.2016.7561358
15. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude manual (version 3.4) (2024), https://maude.lcc.uma.es/maude-manual/
16. Corporation, P.: Pulumi. https://www.pulumi.com/, 2017
17. Coullon, H., Henrio, L., Loulergue, F., Robillard, S.: Component-based distributed software reconfiguration:a verification-oriented survey. ACM Comput. Surv. **56**(1) (Aug 2023). https://doi.org/10.1145/3595376

18. Di Cosmo, R., Lienhardt, M., Treinen, R., Zacchiroli, S., Zwolakowski, J., Eiche, A., Agahi, A.: Automated synthesis and deployment of cloud applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 211–222. ASE '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642980
19. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Information and Computation **239**, 100–121 (1 2014). https://doi.org/10.1016/j.ic.2014.11.002
20. foundation, L.: Opentofu. https://opentofu.org/, 2023
21. Fu, W., Perera, R., Anderson, P., Cheney, J.: mupuppet: A declarative subset of the puppet configuration language **74**, 1–27 (2017). https://doi.org/10.4230/LIPICS.ECOOP.2017.12
22. HashiCorp: HCL. https://github.com/hashicorp/hcl
23. HashiCorp: Standard module structure. https://developer.hashicorp.com/terraform/language/modules/develop/structure
24. HashiCorp: Terraform. https://developer.hashicorp.com/terraform, july 2014
25. HashiCorp: Terraform core resource destruction notes. https://github.com/hashicorp/terraform/blob/main/docs/destroying.md#forced-create-before-destroy
26. Hu, H., Bu, Y., Wong, K., Sood, G., Smiley, K., Rahman, A.: Characterizing static analysis alerts for terraform manifests: An experience report. In: 2023 IEEE Secure Development Conference (SecDev). pp. 7–13 (2023). https://doi.org/10.1109/SecDev56634.2023.00014
27. Lepiller, J., Piskac, R., Schäf, M., Santolucito, M.: Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 105–123. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_6
28. Liu, B., Lim, G., Beckett, R., Godfrey, P.B.: Kivi: verification for cluster management. In: Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference. USENIX ATC'24, USENIX Association, USA (2024). https://doi.org/10.48550/arXiv.2311.02800
29. Microsoft: Azure resource manager (arm). https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/overview
30. Morris, K.: Infrastructure as Code: Dynamic Systems for the Cloud Age. O'Reilly Media, Inc., Sebastopol, CA, USA, second edn. (2021)
31. OpenStack: Heat. https://wiki.openstack.org/wiki/Heat
32. Robillard, S., Coullon, H.: SMT-based planning synthesis for distributed system reconfigurations. In: Johnsen, E.B., Wimmer, M. (eds.) Fundamental Approaches to Software Engineering. pp. 268–287. Springer International Publishing, Cham (2022)
33. Sun, X., Luo, W., Gu, J.T., Ganesan, A., Alagappan, R., Gasch, M., Suresh, L., Xu, T.: Automatic reliability testing for cluster management controllers. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). pp. 143–159 (2022)
34. Sun, X., Ma, W., Gu, J.T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., Xu, T.: Anvil: verifying liveness of cluster management

controllers. In: Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation. OSDI'24, USENIX Association, USA (2024)
35. Yoshida, H., Ogata, K., Futatsugi, K.: Formalization and verification of declarative cloud orchestration. In: Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings 17. pp. 33–49. Springer (2015). https://doi.org/10.1007/978-3-319-25423-4_3