Testing Idempotence for Infrastructure as Code

Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, Tamar Eilam

Testing Idempotence for Infrastructure as Code

Waldemar Hummer[1], Florian Rosenberg[2], Fábio Oliveira[2], and Tamar Eilam[2]

[1] Distributed Systems Group, Vienna University of Technology, Austria
Email: hummer@dsg.tuwien.ac.at
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
Email: {rosenberg,fabolive,eilamt}@us.ibm.com

Abstract. Due to the competitiveness of the computing industry, software developers are pressured to quickly deliver new code releases. At the same time, operators are expected to update and keep production systems stable at all times. To overcome the development–operations barrier, organizations have started to adopt *Infrastructure as Code* (IaC) tools to efficiently deploy middleware and applications using automation scripts. These automations comprise a series of steps that should be *idempotent* to guarantee repeatability and convergence. Rigorous testing is required to ensure that the system idempotently converges to a desired state, starting from arbitrary states. We propose and evaluate a model-based testing framework for IaC. An abstracted system model is utilized to derive state transition graphs, based on which we systematically generate test cases for the automation. The test cases are executed in light-weight virtual machine environments. Our prototype targets one popular IaC tool (Chef), but the approach is general. We apply our framework to a large base of public IaC scripts written by operators, showing that it correctly detects non-idempotent automations.

Keywords: Middleware Deployment, Software Automation, Idempotence, Convergence, Infrastructure as Code, Software Testing

1 Introduction

The ever-increasing need for rapidly delivering code changes to satisfy new requirements has led many organizations to rethink their development practices. A common impediment to this demand for quick code delivery cycles is the well-known tension between software developers and operators: the former are constantly pressured to deliver new releases, whereas the latter must keep production systems stable at all times. Not surprisingly, operators are reluctant to accept changes and tend to consume new code slower than developers would like.

In order to repeatedly deploy middleware and applications to the production environment, operations teams typically rely on automation logic (e.g., scripts). As new application releases become available, this logic may need to be revisited to accommodate new requirements imposed on the production infrastructure. As automation logic is traditionally not developed following the same rigor of software engineering used by application developers (e.g., modularity, re-usability), automations tend to never achieve the same level of maturity and quality, incurring an increased risk of compromising the stability of the deployments.

This state-of-affairs has been fueling the adoption of *DevOps* [1–3] practices to bridge the gap between developers and operators. One of the pillars of DevOps is the notion of *Infrastructure as Code (IaC)* [1, 4], which facilitates the development of automation logic for deploying, configuring, and upgrading inter-related middleware components following key principles in software engineering. IaC automations are expected to be repeatable by design, so they can bring the system to a *desired state* starting from any arbitrary state. To realize this model, state-of-the-art IaC tools, such as Chef [5] and Puppet [6], provide developers with several abstractions to express automation steps as *idempotent* units of work.

The notion of *idempotence* has been identified as the foundation for repeatable, robust automations [7, 8]. Idempotent tasks can be executed multiple times always yielding the same result. Importantly, idempotence is a requirement for *convergence* [7], the ability to reach a certain desired state under different circumstances in potentially multiple iterations. The algebraic foundations of these concepts are well-studied; however, despite (1) their importance as key elements of DevOps automations and (2) the critical role of automations to enable frequent deployment of complex infrastructures, testing of idempotence in real systems has received little attention. To the best of our knowledge, no work to date has studied the practical implications of idempotence or sought to support developers ascertain that their automations idempotently make the system converge.

We tackle this problem and propose a framework for systematic testing of IaC automation scripts. Given a formal model of the problem domain and input coverage goals based on well-defined criteria, a State Transition Graph (STG) of the automation under test is constructed. The resulting STG is used to derive test cases. Although our prototype implementation is based on Chef, the approach is designed for general applicability. We rely on Aspect-Oriented Programming (AOP) to seamlessly hook the test execution harness into Chef, with practically no configuration effort required. Since efficient execution of test cases is a key issue, our prototype utilizes Linux containers (LXC) as light-weight virtual machine (VM) environments that can be instantiated within seconds. Our extensive evaluation covers testing of roughly 300 publicly available, real-life Chef scripts [9]. After executing 3671 test cases, our framework correctly identified 92 of those scripts as non-idempotent in our test environment.

Next, we provide some background on Chef and highlight typical threats to idempotence in automations (§ 2), present an overview of our approach (§ 3), detail the underlying formal model (§ 4), delve into STG-based test case generation and execution (§ 5), unveil our prototype implementation (§ 6), discuss evaluation results (§ 7), summarize related work (§ 8), and wrap up the paper (§ 9).

2 Background and Motivation

In this section we explain the principles behind modern IaC tools and the importance of testing IaC automations for idempotence. Although we couch our discussion in the context of Chef [5], the same principles apply to all such tools.

Chef background. In Chef terminology, automation logic is written as *recipes*, and a *cookbook* packages related recipes. Following a declarative paradigm, recipes

describe a series of *resources* that should be in a particular state. Listing 1.1 shows a sample recipe for the following desired state: directory "/tmp/my_dir" must exist with the specified permissions; package "tomcat6" must be installed; OS service "tomcat6" must run and be configured to start at boot time.

Each resource type (e.g., package) is implemented by platform-dependent providers that properly configure the associated resource instances. Chef ensures the implementation of resource providers is idempotent. Thus, even if our sample recipe is executed multiple times, it will not fail trying to create a directory that already exists. These declarative, idempotent abstractions provide a uniform mechanism for repeatable execution. This model of repeatability is important because recipes are meant to be run periodically to override out-of-band changes, i.e., prevent drifts from the desired state. In other words, a recipe is expected to continuously make the system converge to the desired state.

```
1  directory "tmp/my_dir" do
2    owner "root"
3    group "root"
4    mode 0755
5    action :create
6  end
7  package "tomcat6" do
8    action :install
9  end
10 service "tomcat6" do
11   action [:start, :enable]
12 end
```

Listing 1.1. Declarative Chef Recipe

```
1  bash "build php" do
2    cwd Config[:file_cache_path]
3    code <<-EOF
4
5  tar -zxvf php-#{version}.tar.gz
6  cd php-#{version}
7  ./configure #{options}
8  make && make install
9
10 EOF
11   not_if "which php"
12 end
```

Listing 1.2. Imperative Chef Recipe

Supporting the most common configuration tasks, Chef currently provides more than 20 declarative resource types whose underlying implementation guarantees idempotent and repeatable execution. However, given the complexity of certain tasks that operators need to automate, the available declarative resource types may not provide enough expressiveness. Hence, Chef also supports imperative scripting resources such as *bash* (shell scripts) or *ruby_block* (Ruby code).

Listing 1.2 illustrates an excerpt from a recipe that installs and configures PHP (taken from [9]). This excerpt shows the common scenario of installing software from source code—unpack, compile, install. The imperative shell statements are in the *code* block (lines 5–8). To encourage idempotence even for arbitrary scripts, Chef gives users statements such as `not_if` (line 11) and `only_if` to indicate conditional execution. In our sample, PHP is not compiled and installed if it is already present in the system. Blindly re-executing those steps could cause the script to fail; thus, checking if the steps are needed (line 11) is paramount to avoid errors upon multiple recipe runs triggered by Chef.

Threats to overall idempotence. Idempotence is critical to the correctness of recipes in light of Chef's model of continuous execution and desired-state convergence. Nonetheless, we identify several challenges when it comes to ensuring that a recipe as a whole is idempotent and can make the system converge to a desired state irrespective of the system's state at the start of execution. Because of these challenges, IaC automation developers need thorough testing support.

First, for imperative script resources, the user has the burden of implementing the script in an idempotent way. The user has to decide the appropriate granularity at which idempotence must be enforced so that desired-state convergence can always be achieved with no failures or undesirable side effects. This may not be trivial for recipes with long code blocks or multiple script resources.

Second, the need to use script resources, not surprisingly, occurs often. E.g., out of all 665 publicly available cookbooks in the Opscode community [9] (as of February 2013, only counting cookbooks with at least one resource), we found that 364 (more than 50%) use at least one script resource. What is more, out of 7077 resources from all cookbooks, almost 15% were script resources.

Third, although Chef guarantees that the declarative resource types (e.g., `directory`) are idempotent, there is no guarantee that a sequence of multiple instances as a whole is idempotent, as outlined in [7], specially in the face of script resources. Recall that a recipe typically contains a series of several resource instances of different types, and the entire recipe is re-executed periodically.

Finally, if recipes depend on an external component (e.g., a download server), writing the recipe to achieve overall idempotence may become harder due to unforeseen interactions with the external component (e.g., server may be down).

3 Approach Synopsis

Our work proposes an approach and framework for testing IaC automations for idempotence. We follow a model-based testing approach [10], according to the process outlined in Figure 1. The process contains five main steps with different input and output artifacts. Our test model consists of two main parts: 1) a system model of the automation under test and its environment, including the involved tasks, parameters, system states, and state changes; 2) a state transition graph (STG) model that can be directly derived from the system model.

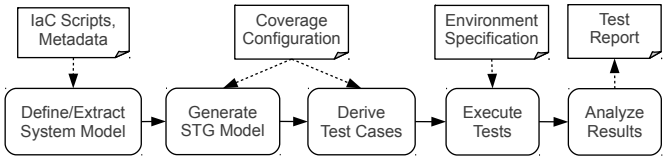

Fig. 1. Model-based testing process.

The input to the first step in Figure 1 consists of the IaC scripts, and additional metadata. The scripts are parsed to obtain the basic system model. IaC frameworks like Chef allow for automatic extraction of most required data, and additional metadata can be provided to complete the model (e.g., value domains for automation parameters). Given the sequence of tasks and their expected state transitions, an STG is constructed which models the possible state transitions that result from executing the automation in different configurations and starting from arbitrary states. Step three in the process derives test case specifications, taking into account user-defined coverage criteria. The test cases are materialized and executed in the real system in step four. During execution, the

system is monitored for state changes by intercepting the automation tasks. Test analysis is applied to the collected data in step five, which identifies idempotence issues based on well-defined criteria, and generates a detailed test report.

4 System Model

This section introduces a model for the IaC domain and a formal definition of idempotence, as considered in this paper. The model and definitions provide the foundation for test generation and the semantics of our test execution engine.

Symbol	Description
K, V	Set of possible **state property keys** (K) and **values** (V).
$d : K \to \mathcal{P}(V)$	**Domain** of possible values for a given state property key.
$P := K \times V$	Possible **property assignments**. $\forall\ (k, v) \in P$: $v \in d(k)$
$S \subseteq [K \to V]$	Set of possible **system states**. The state is defined by (a subset of) the state properties and their values.
$A = \{a_1, a_2, .., a_n\}$	Set of **tasks** (or activities) an automation consists of.
$p : A \to I$	Set of **input parameters** (denoted by set I) for a task.
$D \subseteq \mathcal{P}(A \times A)$	**Task dependency** relationship: task a_1 must be executed before task a_2 iff $(a_1, a_2) \in D$.
$R = \{r_1, r_2, .., r_m\}$	Set of all historical **automation runs**.
$E = \{e_1, e_2, .., e_l\}$	Set of all historical **task executions**.
$r : E \to R$	Maps task executions to automation runs.
$e : (A \cup R) \to E^{\mathbb{N}}$	**List of task executions** for a task or automation run.
$o : E \to \{success, error\}$	Whether a task execution yielded a **success output**.
$succ, pred : A \to A \cup \varnothing$	Task's **successor** or **predecessor** within an automation.
$st, ft : (E \cup R) \to \mathbb{N}$	Timestamp of the **start time** (st) and **finish time** (ft).
$t : (S \times A) \to S$	Expected **state transition** of each task. **Pre-state** maps to **post-state**.
$c : E^{\mathbb{N}} \to [S \to S]$	Actual **state changes** effected by a list of task executions. (state difference between first pre-state and last post-state)
$pre, post : A \to \mathcal{P}(S)$ $pre, post : E \to S$	Return all potential (for a task) or concrete (for a task execution) pre-states (pre) and post-states ($post$).

Table 1. System Model

Table 1 describes each element of our model and the used symbols. Note that $\mathcal{P}$ denotes the *powerset* of a given set. We use the notation $x[i]$ to refer to the ith item of a tuple x, whereas $idx(j, x)$ gives the (one-based) index of the first occurrence of item j in tuple x or $\varnothing$ if j does not exist in x. Moreover, $X^{\mathbb{N}} := \bigcup_{n \in \mathbb{N}} X^n$ denotes the set of all tuples (with any length) over the set X.

4.1 Automation and Automation Tasks

An automation (A) consists of multiple tasks with dependencies (D) between them. We assume a total ordering of tasks, i.e., $\forall a_1, a_2 \in A : (a_1 \neq a_2) \iff ((a_1, a_2) \in D) \oplus ((a_2, a_1) \in D)$. An automation is executed in one or multiple automation runs (R), which in turn consist of a multitude of task executions (E).

#	Task	Parameters
a_1	Install MySQL	-
a_2	Set MySQL password	$p2$ = root password
a_3	Install Apache & PHP	$p3$ = operating system distribution (e.g., 'debian')
a_4	Deploy Application	$p4$ = application context path (e.g., '/myapp')

Table 2. Key Automation Tasks of the Sample Scenario

For clarity, we relate the above concepts to a concrete Chef scenario. Consider a Chef recipe that installs and configures a *LAMP* stack (Linux-Apache-MySQL-PHP) to run a Web application. For simplicity, let us assume our recipe defines four resource instances corresponding to the tasks described in Table 2.

A Chef recipe corresponds to an *automation*, and each resource in the recipe is a *task*. Given our model and the recipe summarized in Table 2, we have $A = \{a_1, a_2, a_3, a_4\}$. Note that a_1 could be a package resource to install MySQL, similar to the package resource shown in the recipe of Listing 1.1, whereas a_3 could be implemented by a script resource similar to the one shown in Listing 1.2 (see Section 2). Table 2 also shows the input parameters consumed by each task.

As discussed in Section 2, an automation (Chef recipe) is supposed to make the system converge to a *desired state*. Each task leads to a certain state transition, converting the system from a pre-state to a post-state. A system state $s \in S$ consists of a number of system properties, defined as (key,value) pairs. For our scenario, let us assume we track the state of open ports and OS services installed, such that $K = \{$'*open_ports*', '*services*'$\}$. Also, suppose that, prior to the automation run, the initial system state is given by $s_0 = \{$('*open_ports*', $\{22\}$), ('*services*', $\{$'*ssh*', '*acpid*'$\})\}$, i.e., port 22 is open and two OS services (`ssh` and `acpid`) are running. After task a_1's execution, the system will transition to a new state $s_1 = \{$('*open_ports*', $\{22, 3306\}$), ('*services*', $\{$'*ssh*', '*acpid*', '*mysql*'$\})\}$, i.e., task a_1 installs the `mysql` service which will be started and open port 3306. Our prototype testing framework tracks the following pieces of state: network routes, OS services, open ports, mounted file systems, file contents and permissions, OS users and groups, cron jobs, installed packages, and consumed resources.

We distinguish the *expected state transition* (expressed via function t) and the *actual state change* (function c) that took place after executing a task. The expected state transitions are used to build a state transition graph (Section 4.2), whereas the actual state changes are monitored and used for test result analysis.

4.2 State Transition Graph

The system model established so far in this section can be directly translated into a *state transition graph* (STG) which we then use for test generation. The $STG = (V_G, T_G)$ is a directed graph, where V_G represents the possible system states, and T_G is the set of edges representing the expected state transitions.

Figure 2 depicts an STG which contains the pre-states and post-states of the four tasks used in our scenario. For illustration, a tuple of four properties is encoded in each state: *my* (MySQL installed?), *pw* (password configured?), *php* (Apache and PHP installed?), and *app* (set of applications deployed in the

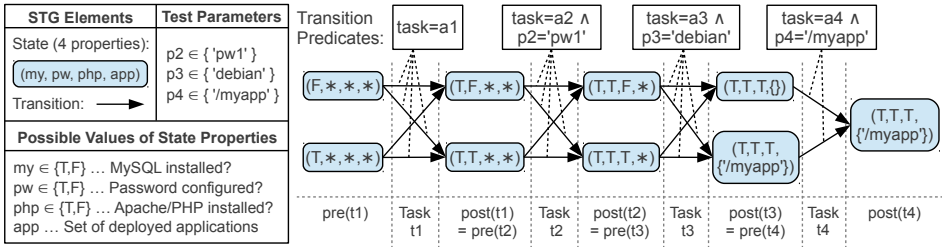

Fig. 2. Simple State Transition Graph Corresponding to Table 2

Apache Web server). For space limitations, branches (e.g., based on which operating system is used) are not included in the graph, and the wildcard symbol ($*$) is used as a placeholder for arbitrary values. The pre-states of each task should cover all possible values of the state properties that are (potentially) changed by this task. For instance, the automation should succeed regardless of whether MySQL is already installed or not. Hence, the pre-states of task $t1$ contain both values $my = F$ and $my = T$. Note that instead of the wildcard symbol we could also expand the graph and add one state for each possible value, which is not possible here for space reasons.

4.3 Idempotence of Automation Tasks

Following [7], a task $a \in A$ is *idempotent* with respect to an equivalence relation $\approx$ and a sequence operator $\circ$ if repeating a has the same effect as executing it once, $a \circ a \approx a$. Applied to our model, we define the conditions under which a task is considered idempotent based on the evidence provided by historical task executions (see Definition 3). As the basis for our definition, we introduce the notion of *non-conflicting system states* in Definition 1.

Definition 1 *A state property assignment* $(k, v_2) \in P$ *is* non-conflicting *with another assignment* $(k, v_1) \in P$*, denoted* $nonConf((k, v_1), (k, v_2))$*, if either 1)* $v_1 = v_2$ *or 2)* v_1 *indicates a state which eventually leads to state* v_2*.*

That is, non-conflicting state is used to express state properties in transition. For example, consider that k denotes the status of the MySQL server. Clearly, for two state values $v_1 = v_2 =$ '*started*', (k, v_2) is non-conflicting with (k, v_1). If v_1 indicates that the server is currently starting up ($v_1 =$ '*booting*'), then (k, v_2) is also non-conflicting with (k, v_1). The notion of non-conflicting state properties accounts for long-running automations which are repeatedly executed until the target state is eventually reached. In general, domain-specific knowledge is required to define concrete non-conflicting properties. By default, we consider state properties as non-conflicting if they are equal. Moreover, if we use a wildcard symbol ($*$) to denote that the value of k is unknown, then (k, v_x) is considered non-conflicting with $(k, *)$ for any $v_x \in V$.

Definition 2 *A state* $s_2 \in S$ *is* non-conflicting *with some other state* $s_1 \in S$ *if* $\forall (k_1, v_1) \in s_1, (k_2, v_2) \in s_2 : (k_1 = k_2) \implies nonConf((k_1, v_1), (k_2, v_2))$.

Put simply, non-conflicting states require that all state properties in one state be non-conflicting with corresponding state properties in the other state. Based on the notion of non-conflicting states, Definition 3 introduces idempotent tasks.

Definition 3 *An automation task $a \in A$ is considered* idempotent *with respect to its historical executions $e(a) = \langle e_1, e_2, \ldots, e_n \rangle$ iff for each two executions $e_x, e_y \in e(a)$ the following holds:*
$(ft(e_x) \leq st(e_y) \wedge o(e_x) = success) \Rightarrow$
$(o(e_y) = success \wedge (c(\langle e_y \rangle) = \emptyset \vee nonConf(post(e_y), pre(e_y))))$

In verbal terms, if a task execution $e_x \in e(a)$ succeeds at some point, then all following executions (e_y) must yield a successful result, and either (1) effect no state change, or (2) effect a state change where the post-state is non-conflicting with the pre-state. Equivalently, we define idempotence for task sequences.

Definition 4 *A task sequence $a_{seq} = \langle a_1, a_2, ..., a_n \rangle \in A^n$ is considered* idempotent *iff for each two sequences of subsequent task executions $e'_{seq}, e''_{seq} \in (e(a_1) \times e(a_2) \times ... \times e(a_n))$ the following holds:*
$ft(e'_{seq}[n]) \leq st(e''_{seq}[1]) \Rightarrow$
$((\forall i \in \{1, \ldots, n\} : o(e'_{seq}[i]) = success \Rightarrow o(e''_{seq}[i]) = success) \wedge$
$(c(e''_{seq}) = \emptyset \vee nonConf(post(e''_{seq}[i]), pre(e''_{seq}[i]))))$

Note that our notion of idempotence basically corresponds to the definition in [7], with two subtle differences: first, we not only consider the tasks' post-state, but also distinguish between successful/unsuccessful task executions; second, we do not require post-states to be strictly equal, but allow for non-conflicting states.

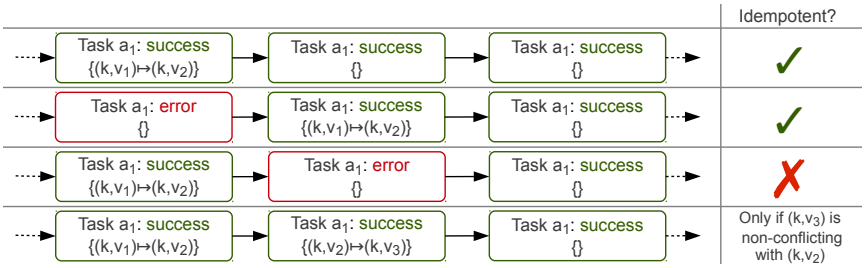

Fig. 3. Idempotence for Different Task Execution Patterns

Figure 3 illustrates idempotence of four distinct task execution sequences. Each execution is represented by a rounded rectangle which contains the result and the set of state changes. For simplicity, the figure is based on a single task a_1, but the same principle applies also to task sequences. Sequence 1 is clearly idempotent, since all executions are successful and the state change from pre-state (k, v_1) to post-state (k, v_2) only happens for the first execution. Sequence 2 is idempotent, even though it contains an unsuccessful execution in the beginning. This is an important case that accounts for repeatedly executed automations which initially fail until a certain requirement is fulfilled (e.g., Apache server waits until MySQL has been configured on another host). Sequence 3 is non-idempotent (even though no state changes take place after the first execution)

because an execution with error follows a successful one. As a typical example, consider a script resource which moves a file using command "`mv X Y`". On second execution, the task returns an error code, because file `X` does not exist anymore. In sequence 4, idempotence depends on whether (k, v_3) represents a state property value that is non-conflicting with (k, v_2). For instance, assume $k =$ '*service.mysql*' denotes whether MySQL is started. If $v_2 =$ '*booting*' and $v_3 =$ '*started*', then a_1 is considered idempotent. Otherwise, if $v_2 =$ '*booting*' and $v_3 =$ '*stopped*', then v_3 is conflicting with v_2, and hence a_1 is not idempotent.

5 Test Design

This section details the approach for testing idempotence of IaC automations. In Section 5.1, we discuss how test cases are derived from a graph representation of the possible system states and transitions, thereby considering customizable test coverage goals. Section 5.2 covers details about the test execution in isolated virtualized environments, as well as test parallelization and distribution.

5.1 STG-Based Test Generation

We observe that the illustrative STG in Figure 2 represents a baseline vanilla case. Our aim is to transform and "perturb" this baseline execution sequence in various ways, simulating different starting states and repeated executions of task sequences, which a robust and idempotent automation should be able to handle. Based on the system model (Section 4) and user-defined coverage configuration, we systematically perform graph transformations to construct an STG for test case generation. The coverage goals have an influence on the size of the graph and the set of generated test cases. Graph models for testing IaC may contain complex branches (e.g., for different test input parameters) and are in general cyclic (to account for repeated execution). However, in order to efficiently apply test generation to the STG, we prefer to work with an acyclic graph (see below).

In the following, we briefly introduce the test coverage goals applied in our approach, discuss the procedure for applying the coverage configuration to concrete graph instances, and finally define the specification of test cases.

Test Coverage Goals We define specific test coverage goals that are tailored to testing idempotence and convergence of IaC automations.
idemN: This coverage parameter specifies a set of task sequence lengths for which idempotence should be tested. The possible values range from $idemN = \{1\}$ (idempotence of single tasks) to $idemN = \{1, \ldots, |A|\}$ (maximum sequence length covering all automation tasks). Evidently, higher values produce more test cases, whereas lower values have the risk that problems related to dependencies between "distant" tasks are potentially not detected (see also Section 7.2).
repeatN: This parameter controls the number of times each task is (at most) repeated. If the automation is supposed to converge after a single run (most Chef recipes are designed that way, see our evaluation in Section 7), it is usually

sufficient to have $repeatN = 1$, because many idempotence related problems are already detected after executing a task (sequence) twice. However, certain scenarios might require higher values for $repeatN$, in particular automations that are continuously repeated in order to eventually converge. The tester then has to use domain knowledge to set a reasonable upper bound of repetitions.
restart: The boolean parameter $restart$ determines whether tasks are arbitrarily repeated in the middle of the automation ($restart = false$), or the whole automation always gets restarted from scratch ($restart = true$). Consider our scenario automation with task sequence $\langle a_1, a_2, a_3, a_4 \rangle$. If we require $idemN = 3$ with $restart = true$, then the test cases could for instance include the task sequences $\langle a_1, a_1, ... \rangle$, $\langle a_1, a_2, a_1, ... \rangle$, $\langle a_1, a_2, a_3, a_1, ... \rangle$. If $restart = false$, we have additional test cases, including $\langle a_1, a_2, a_3, a_2, a_3, ... \rangle$, $\langle a_1, a_2, a_3, a_4, a_2, a_3, ... \rangle$, etc.
forcePre: This parameter specifies whether different pre-states for each task are considered in the graph. If $forcePre = true$, then there needs to exist a graph node for each potential pre-state $s \in pre(a)$ of each task $a \in A$ (see, e.g., Figure 2). Note that the potential pre-state should also cover all post-states, because of repeated task execution. Contrary, $forcePre = false$ indicates that a wildcard can be used for each pre-state, which reduces the number of state nodes in Figure 2 from 9 to 5. The latter ($forcePre = false$) is a good baseline case if pre-states are unknown or hard to produce. In fact, enforcing a certain pre-state either involves executing the task (if the desired pre-state matches a corresponding post-state) or accessing the system state directly, which is in general not trivial.
graph: This parameter refers to the STG-based coverage goal that should be achieved. Offut et al. [11] define four testing goals (with increased level of coverage) to derive test cases from state-based specifications. *Transition coverage*, *full predicate coverage* (one test case for each clause on each transition predicate, cf. Figure 2), *transition-pair coverage* (for each state node, all combinations of incoming and outgoing transitions are tested), and *full sequence coverage* (each possible and relevant execution path is tested, usually constrained by applying domain knowledge to ensure a finite set of tests [11]). By default, we utilize transition coverage on a cycle-free graph. Details are discussed next.

Coverage-Specific STG Construction In Figure 4, graph construction is illustrated by means of an STG which is gradually enriched and modified as new coverage parameters are defined. The STG is again based on our scenario (labels of state properties and transition predicates are left out). First, $forcePre = false$ reduces the number of states as compared to Figure 2. Then, we require that task sequences of any length should be tested for idempotence ($idemN = \{1, 2, 3, 4\}$), which introduces new transitions and cycles into the graph. The configuration $restart = true$ removes part of the transitions, cycles still remain. After the fourth configuration step, $repeatN = 1$, we have determined the maximum number of iterations and construct an acyclic graph.

To satisfy the $graph = transition$ criterion in the last step, we perform a deep graph search to find any paths from the start node to the terminal node. The procedure is trivial, since the graph is already acyclic at this point. Each generated execution path corresponds to one test case, and the transition

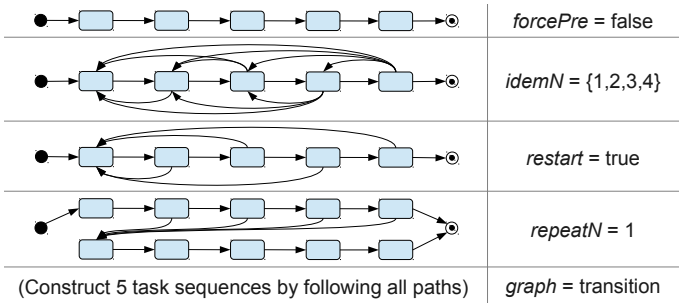

Fig. 4. Coverage-Specific STG Construction

predicates along the path correspond to the inputs for each task (e.g., MySQL password parameter $p2$, cf. Figure 2). For brevity, our scenario does not illustrate the use of alternative task parameter inputs, but it is easy to see how input parameters can be mapped to transition predicates. As part of our future work, we consider combining our approach with combinatorial testing techniques [12] to cover different input parameters. It should be noted, though, that (user-defined) input parameters in the context of testing IaC are way less important than in traditional software testing, since the core "input" to automation scripts is typically defined by the characteristics of the environment they operate in.

Test Case Specification The coverage-specific graph-based test model is used to generate executable tests. Table 3 summarizes the key information of a test case: 1) the input parameters consumed by the tasks (in), 2) the end-to-end sequence of tasks to be executed (seq), and 3) the automation run that resulted from executing the test case (res), which is used for result analysis. For 1), default parameters can be provided along with the system model (cf. Figure 1). Moreover, automation scripts in IaC frameworks like Chef often define reasonable default values suitable for most purposes. For 2), we traverse the cycle-free STG constructed earlier, and each path (task sequence) represents a separate test.

Symbol	**Description**
C; $T \subseteq C$	Set of all possible **test cases** (C) for the automation under test; **test suite** (T) with the set of actual test cases to be executed.
$in : C \to [I \to V]$	**Parameter assignment** with concrete input values for a test case.
$seq : C \to A^{\mathbb{N}}$	Entire **task sequence** to be executed by a test case.
$res : C \to R$	**Automation run** that results from executing a test case.

Table 3. Simplified Model for Test Case Specification

5.2 Test Execution

Since our tests rely on extraction of state information, it is vital that each test be executed in a clean and isolated environment. At the same time, tests should be parallelized for efficient usage of computing resources. Virtual machine (VM) containers provide the right level of abstraction for this purpose. A VM operates

within a host operating system (OS) and encapsulates the filesystem, networking stack, process space, and other relevant system state. Details about VM containers in our implementation are given in Section 6.

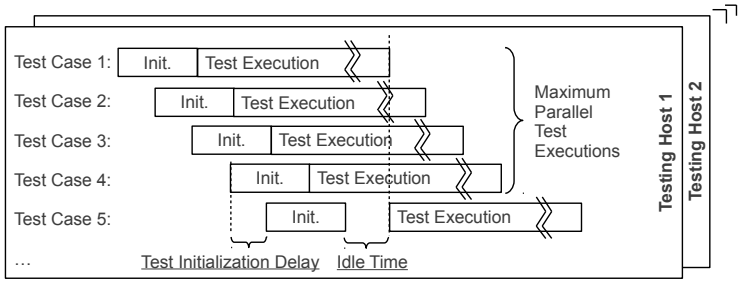

Fig. 5. Test Execution Pipeline

The execution is managed in a testing pipeline, as illustrated in Figure 5. Prior to the actual execution, each container is provided with a short initialization time with exclusive resource access for booting the OS, initializing the automation environment and configuring all parameters. Test execution is then parallelized in two dimensions: the tests are distributed to multiple testing hosts, and a (limited) number of test containers can run in parallel on a single host.

6 Implementation

This section discusses the prototypical implementation of our distributed testing framework. Figure 6 illustrates the architecture from the perspective of a single testing host. A Web user interface guides the test execution. Each host runs a test manager which materializes tests and creates new containers for each test case.

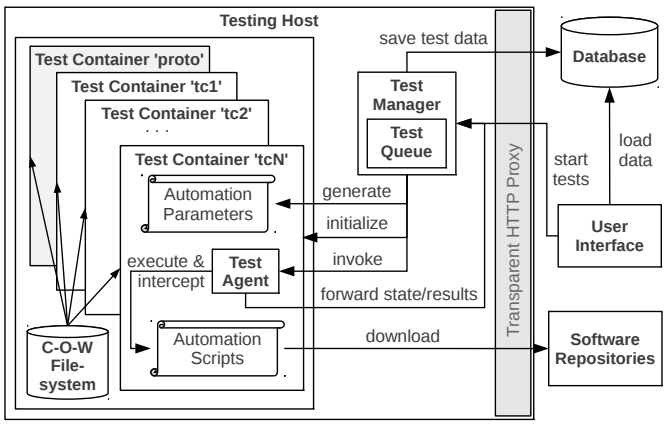

Fig. 6. Test Framework Architecture

Our framework parallelizes the execution in two dimensions: first, multiple testing hosts are started from a pre-configured VM image; second, each testing

host contains several containers executing test cases in parallel. We utilize the highly efficient Linux containers[3] (LXC). Each container has a dedicated root directory within the host's file system. We use the notion of *prototype* container templates (denoted 'proto' in Figure 6) to provide a clean environment for each test. Each prototype contains a base operating system (Ubuntu 12.04 and Fedora 16 in our case) and basic services such as a secure shell (SSH) daemon. Instead of duplicating the entire filesystem for each container, we use a *btrfs*[4] copy-on-write (C-O-W) filesystem, which allows to spawn new instances within a few seconds. To avoid unnecessary re-downloads of external resources (e.g., software packages), each host is equipped with a *Squid*[5] proxy server.

The test agent within each container is responsible for launching the automation scripts and reporting the results back to the test manager which stores them in a MongoDB database. Our framework uses *aquarium*[6], an AOP library for Ruby, to intercept the execution of Chef scripts and extract the relevant system state. Chef's execution model makes that task fairly easy: an aspect that we defined uses a method join point `run_action` in the class `Chef::Runner`. The aspect then records the state snapshots before and after each task. We created an extensible mechanism to define which Chef resources can lead to which state changes. For example, the `user` Chef resource may add a user. Whenever this resource is executed we record whether a user was actually added in the OS. As part of the interception, we leverage this mapping to determine the corresponding system state in the container via Chef's discovery tool *Ohai*. We extended *Ohai* with our own plugins to capture the level of detail required. In future work, we plan to additionally monitor the execution on system call level using *strace*, which will allow to capture additional state changes that we currently miss.

If an exception is raised during the test execution, the details are stored in the testing DB. Finally, after each task execution we check whether any task needs to be repeated at this time (based on the test case specification).

7 Evaluation

To assess the effectiveness of our approach and prototype implementation, we have performed a comprehensive evaluation, based on publicly available Chef cookbooks maintained by the Opscode community. Out of the 665 executable Opscode cookbooks (as of February 2013), we selected a representative sample of 161 cookbooks, some tested in different versions (see Section 7.4), resulting in a total of 298 tested cookbooks. Our selection criteria were based on 1) popularity in terms of number of downloads, 2) achieving a mix of recipes using imperative scripting (e.g., `bash`, `execute`) and declarative resources (e.g., `service`, `file`).

In Section 7.1 we present aggregated test results over the set of automation scripts used for evaluation, Section 7.2 discusses some interesting cases in more

[3] http://lxc.sourceforge.net/
[4] https://btrfs.wiki.kernel.org/
[5] http://www.squid-cache.org/
[6] http://aquarium.rubyforge.org/

detail, in Section 7.3 we contrast the idempotence results for different task types, and Section 7.4 analyzes the evolution of different versions of popular cookbooks.

7.1 Aggregated Test Results

In this section we summarize the test results achieved from applying our testing approach to the selected Opscode Chef cookbooks. For space limitations, we can only highlight the core findings, but we provide a Web page[7] with accompanying material and detailed test results. Table 4 gives an overview of the overall evaluation results. The "min/max/total" values indicate the minimum/maximum value over all individual cookbooks, and the total number for all cookbooks.

Tested Cookbooks	298
Number of Test Cases	3671
Number of Tasks (min/max/total)	1 / 103 / 4112
Total Task Executions	187986
Captured State Changes	164117
Total Non-Idempotent Tasks	263
Cookbooks With Non-Idempotent Tasks	92
Overall Net Execution Time	25.7 CPU-days
Overall Gross Execution Time	44.07 CPU-days

Table 4. Aggregated Evaluation Test Results

We have tested a total of 298 cookbooks, selected by high popularity (download count) and number of imperative tasks (script resources). Cookbooks were tested in their most recent version, and for the 20 most popular cookbooks we tested (up to) 10 versions into the past, in order to assess their evolution with respect to idempotence (see Section 7.4). As part of the selection process, we manually filtered cookbooks that are not of interest or not suitable for testing: for instance, cookbook `application` defines only attributes and no tasks, or cookbook `pxe_install_server` downloads an entire 700MB Ubuntu image file.

The 298 tested cookbooks contain 4112 tasks in total. In our experiments, task sequences of arbitrary length are tested ($\{1, .., |A|\}$), tasks are repeated at most once ($repeatN = 1$), and the automation is always restarted from the first task ($restart = true$). Based on this coverage, a total of 3671 test cases (i.e., individual instantiations with different configurations) were executed. 187986 task executions were registered in the database, and 164117 state changes were captured as a direct result. The test execution occupied our hardware for an overall gross time of 44.07 CPU-days. Extracting the overhead of our tool, which includes mostly capturing of system state and computation of state changes, the net time is 25.7 CPU-days. Due to parallelization (4 testing hosts, max. 5 containers each) the tests actually finished in much shorter time (roughly 5 days).

The tests have led to the identification of 263 non-idempotent tasks. Recall from Section 4 that a task is non-idempotent if any repeated executions lead to state changes or yield a different success status than the previous executions.

[7] http://dsg.tuwien.ac.at/testIaC/

7.2 Selected Result Details

To provide a more detailed picture, we discuss interesting cases of non-idempotent recipes. We explain for each case how our approach detected the idempotence issue. We also discuss how we tracked down the actual problem, to verify the results and understand the underlying implementation bug. It should be noted, however, that our focus is on problem detection, not debugging or root cause analysis. However, using the comprehensive data gathered during testing, our framework has also significantly helped us find the root of these problems.

Chef Cookbook `timezone`: A short illustrative cookbook is `timezone` v0.0.1 which configures the time zone in `/etc/timezone`. Table 5 lists the three tasks: a_1 installs package `tzdata` and initializes the file with "Etc/UTC", a_2 writes "UTC" to the file, and a_3 reconfigures the package `tzdata`, resetting the file content. For our tests, "UTC" and "Etc/UTC" are treated as conflicting property values. Hence, tasks a_2 and a_3 are clearly non-idempotent, e.g., considering the execution sequence $\langle a_1, a_2, a_3, a_1, a_2, a_3 \rangle$: on second execution, a_1 has no effect (package is already installed), but a_2, a_3 are re-executed, effectively overwriting each other's state changes. Note that $\langle a_1, a_2 \rangle$ and $\langle a_1, a_2, a_3 \rangle$ are idempotent as a sequence; however, a perfectly idempotent automation would ensure that tasks do not alternatingly overwrite changes. Moreover, the overhead of re-executing tasks a_2, a_3 could be avoided, which is crucial for frequently repeated automations.

Task	Resource Type	Description
a_1	`package`	Installs package `tzdata`, writes "Etc/UTC" to `/etc/timezone`
a_2	`template`	Writes timezone value "UTC" to `/etc/timezone`
a_3	`bash`	Runs `dpkg-reconfigure tzdata`, again writes "Etc/UTC" to `/etc/timezone`

Table 5. Tasks in Chef Cookbook `timezone`

Chef Cookbook `tomcat6`: In the popular cookbook `tomcat6` v0.5.4 (> 2000 downloads), we identified a non-trivial idempotence bug related to incorrect file permissions. The version number indicates that the cookbook has undergone a number of revisions and fixes, but this issue was apparently not detected.

The crucial tasks are outlined in Table 6 (the entire automation consists of 25 tasks). Applying the test coverage settings from Section 7.1, the test suite for this cookbook consists of 23 test cases, out of which two test cases (denoted $t1, t2$) failed. Test t_1 is configured to run task sequence $\langle a_1, ..., a_{21}, a_1, ..., a_{25} \rangle$ (simulating that the automation is terminated and repeated after task a_{21}), and test t_2 is configured with task sequence $\langle a_1, ..., a_{22}, a_1, ..., a_{25} \rangle$ (restarting after task a_{22}). Both test cases failed at the second execution of task a_{16}, denoted $e(a_{16})[2]$ in our model, which copies configuration files to a directory previously created by task a_9. In the following we clarify why and how this fault happens.

The reason why t_1 and t_2 failed when executing $e(a_{16})[2]$ is that at the time of execution the file `/etc/tomcat6/logging.properties` is owned by user

Task	Resource Type	Description
...	...	...
a_9	`directory`	Creates directory `/etc/tomcat6/`
...	...	...
a_{16}	`bash`	Copies files to `/etc/tomcat6/` as user `tomcat`; **only** executed if `/etc/tomcat6/tomcat6.conf` does not exist.
...	...	...
a_{21}	`file`	Writes to `/etc/tomcat6/logging.properties` as user `root`.
a_{22}	`service`	Enables the service `tomcat` (i.e., automatic start at boot)
a_{23}	`file`	Creates file `/etc/tomcat6/tomcat6.conf`
...	...	...

Table 6. Tasks in Chef Cookbook `tomcat6`

`root`, and a_{16} attempts to write to the same file as user `tomcat` (resulting in "permission denied" from the operating system). We observe that task a_{21} also writes to the same file, but in contrast to task a_{16} not as user `tomcat`, but as user `root`. At execution $e(a_{21})[1]$, the content of the file gets updated and the file ownership is set to `root`. Hence, the cookbook developer has introduced an implicit dependency between tasks a_{16} and a_{21}, which leads to idempotence problems. Note that the other 21 test cases did not fail. Clearly, all test cases in which the automation is restarted *before* the execution of task a_{21} are not affected by the bug, since the ownership of the file does not get overwritten. The remaining test cases in which the automation was restarted after a_{21} (i.e., after a_{23}, a_{24}, and a_{25}) did not fail due to a conditional statement `not_if` which ensures that a_{16} is only executed if `/etc/tomcat6/tomcat6.conf` does not exist.

Chef Cookbook `mongodb-10gen` The third interesting case we discuss is cookbook `mongodb-10gen` (installs *MongoDB*), for which our framework allowed us to detect an idempotence bug in the Chef implementation itself. The relevant tasks are illustrated in Table 7: a_{11} installs package `mongodb-10gen`, a_{12} creates a directory, and a_{13} creates another sub-directory and places configuration files in it. If installed properly, the package `mongodb-10gen` creates user and group `mongodb` on the system. However, since the cookbook does not configure the repository properly, this package cannot be installed, i.e., task a_{11} failed in our tests. Now, as task a_{12} is executed, it attempts to create a directory with user/group `mongodb`, which both do not exist at that time. Let us assume the test case with task sequence $\langle a_1, \ldots, a_{13}, a_1, \ldots, a_{13} \rangle$. As it turns out, the first execution of a_{13} creates `/data/mongodb` with user/group set to `root`/`mongodb` (even though group `mongodb` does not exist). On the second execution of a_{12}, however, Chef again tries to set the directory's ownership and reports an error that user `mongodb` does not exist. This behavior is clearly against Chef's notion of idempotence, because the error should have been reported on the first task execution already. In fact, if the cookbook was run only once, this configuration error would not be detected, but may lead to problems at runtime. We submitted a bug report (Opscode ticket `CHEF-4236`) which has been confirmed by Chef developers.

Task	Resource Type	Description
...	...	...
a_{11}	`package`	Installs package `mongodb-10gen`
a_{12}	`directory`	Creates directory `/data`
a_{13}	`remote_directory`	Creates directory `/data/mongodb` as user/group `mongodb`

Table 7. Tasks in Chef Cookbook `mongodb-10gen`

Lessons Learned The key take-away message of these illustrative real-world examples is that automations may contain complex implicit dependencies, which IaC developers are often not aware of, but which can be efficiently tested by our approach. For instance, the conditional `not_if` in a_{16} of recipe `tomcat6` was introduced to avoid that the config file gets overwritten, but the developer was apparently not aware that this change breaks the idempotence and convergence of the automation. This example demonstrates nicely that some idempotence and convergence problems (particularly those involving dependencies among multiple tasks) cannot be avoided solely by providing declarative and idempotent resource implementations (e.g., as provided in Chef) and hence require systematic testing.

7.3 Idempotence for Different Task Types

Table 8 shows the number of identified non-idempotent tasks (denoted #NI) for different task types. The task types correspond to the Chef resources used in the evaluated cookbooks. The set of scripting tasks (`execute`, `bash`, `script`, `ruby_block`) makes up for 90 of the total 263 non-idempotent tasks, which confirms our suspicion that these tasks are error-prone. Interestingly, the `service` task type also shows many non-idempotent occurrences. Looking further into this issue, we observed that `service` tasks often contain custom code commands to start/restart/enable services, which are prone to idempotence problems.

Task Type	#NI	Task Type	#NI	Task Type	#NI
`service`	66	`directory`	10	`link`	3
`execute`	44	`remote_file`	10	`bluepill_service`	2
`package`	30	`gem_package`	7	`cookbook_file`	2
`bash`	27	`file`	5	`git`	2
`template`	19	`python_pip`	5	`user`	2
`script`	15	`ruby_block`	4	`apt_package`	1

Table 8. Non-Idempotent Tasks By Task Type

7.4 Idempotence for Different Cookbook Versions

We analyzed the evolution of the 20 most popular Chef cookbooks. The results in Table 9 leave out cookbooks with empty default recipes (`application`, `openssl`, `users`) and cookbooks without any non-idempotent tasks: `mysql`, `java`, `postgresql`, `build-essential`, `runit`, `nodejs`, `git`, `ntp`, `python`, `revealcloud`, `graylog2`. For the cookbooks under test, new releases fixed idempotence issues, or at least did not introduce new issues. Our tool automatically determines these data, hence it can be used to test automations for regressions and new bugs.

Cookbook	i-9	i-8	i-7	i-6	i-5	i-4	i-3	i-2	i-1	i
apache2 (i=1.4.2)	1	1	1	0	0	0	0	0	0	0
nagios (i=3.1.0)	1	1	0	0	0	0	0	0	0	0
zabbix (i=0.0.40)	2	2	2	2	2	2	2	2	2	2
php (i=1.1.4)	1	1	0	0	0	0	0	0	0	0
tomcat6 (i=0.5.4)			3	3	3	3	3	3	2	1
riak (i=1.2.1)	1	1	1	1	1	1	0	0	0	0

Table 9. Evolution of Non-Idempotent Tasks By Increasing Version

8 Related Work

Existing work has identified the importance of idempotence for building reliable distributed systems [13] and database systems [14]. Over the last years, the importance of building testable system administration [8] based on convergent models [15, 7] became more prevalent. *cfengine* [16] was among the first tools in this space. More recently, other IaC frameworks such as Chef [5] or Puppet [6] heavily rely on these concepts. However, automated and systematic testing of IaC for verifying idempotence and convergence has received little attention, despite the increasing trend of automating multi-node system deployments (i.e., continuous delivery [17]) and placement of virtual infrastructures in the Cloud [18].

Existing IaC test frameworks allow developers to manually write test code using common Behavior-Driven Development (BDD) techniques. ChefSpec [19] or Cucumber-puppet [20] allow to encode the desired behavior for verifying individual automation tasks (unit testing). Test Kitchen [21] goes one step further by enabling testing of multi-node system deployments. It provisions isolated test environments using VMs which execute the automation under test and verify the results using the provided test framework primitives. This kind of testing is a manual and labor intensive process. Our framework takes a different approach by systematically generating test cases for IaC and executing them in a scalable virtualized environment (LXC) to detect errors and idempotence issues.

Extensive research is conducted on automated software debugging and testing techniques, including model-based testing [22] or symbolic execution [23], as well as their application to specialized problem areas, for instance control flow based [24] or data flow based [25] testing approaches. Most existing work and tools, however, are not directly applicable to the domain of IaC, for two main reasons: (i) IaC exposes fairly different characteristics than traditional software systems, i.e., idempotence and convergence; (ii) IaC needs to be tested in real environments to ensure that system state changes triggered by automation scripts can be asserted accordingly. Such tests are hard to simulate, hence symbolic execution would have little practical value. Even though dry-run capabilities exist (e.g, Chef's *why-run* capability), they cannot replace systematic testing. The applicability of automated testing is a key requirement identified by other approaches [26–28], whether the test target is system software or IaC.

Existing approaches for middleware testing have largely focused on performance and efficiency. Casale et al. [29] use automatic stress testing for multi-tier systems. Their work places bursty service demands on system resources, in order to identify performance bottlenecks as well as latency and throughput degrada-

tions. Other work focuses on testing middleware for elasticity [30], which is becoming a key property for Cloud applications. Bucur et al. [26] propose an automated software testing approach that parallelizes symbolic executions for efficiency. The system under test can interact with the environment via a "symbolic system call" layer that implements a set of common POSIX primitives. Their approach could potentially enhance our work and may speed up the performance, but requires a complete implementation of the system call layer.

Other approaches deal with finding and fixing configuration errors [31, 32]. Faults caused by configuration errors are often introduced during deployment and remain dormant until activated by a particular action. Detecting such errors is challenging, but tools like AutoBash [32] or Chronus [31] can effectively help. A natural extension would be to also take into account the IaC scripts to find the configuration parameter that potentially caused the problem. Burg et al. [28] propose automated system tests using declarative virtual machines. Declarative specifications describe external dependencies (e.g., access to external services) together with an imperative test script. Their tool then builds and instantiates the virtual machine necessary to run the script. Our approach leverages prebuilt containers in LXC; dynamically creating a declarative specification would be possible but building a VM is more costly than bringing up an LXC container.

9 Conclusion

We propose an approach for model-based testing of *Infrastructure as Code*, aiming to verify whether IaC automations, such as Chef recipes, can repeatedly make the target system converge to a desired state in an idempotent manner. Given the IaC model of periodic re-executions, idempotence is a critical property which ensures repeatability and allows automations to start executing from arbitrary initial or intermediate states. Our extensive evaluation with real-world IaC scripts from the OpsCode community revealed that the approach effectively detects non-idempotence. Out of roughly 300 tested Chef scripts, almost a third were identified as non-idempotent. In addition, we were able to detect and report a bug in the Chef implementation itself.

Our novel approach opens up exciting future research directions. First, we will extend our prototype to handle the execution of distributed automations with cross-node dependencies, which is often used to deploy multi-node systems. Second, we plan to apply the approach to other IaC frameworks like Puppet, whose execution model does not assume total task ordering. Third, we envision that systematic debugging/analysis can be pushed further to identify implicit dependencies introduced by IaC developers. Moreover, we are currently extending the state capturing mechanism to detect fine-grained changes on system call level. The hypothesis is that the improved mechanism can lead to detection of additional non-idempotence cases stemming from side effects we currently miss.

References

1. Hüttermann, M.: DevOps for Developers. Apress (2012)

2. Loukides, M.: What is DevOps? O'Reilly Media (2012)
3. Schaefer, A., Reichenbach, M., Fey, D.: Continuous Integration and Automation for Devops. IAENG Trans. on Engineering Technologies **170** (2013) 345–358
4. Nelson-Smith, S.: Test-Driven Infrastructure with Chef. O'Reilly (2011)
5. Opscode: http://www.opscode.com/chef/
6. Puppet Labs: http://puppetlabs.com/
7. Couch, A., Sun, Y.: On the algebraic structure of convergence. In: 14th Int. Workshop on Distr. Systems: Operations and Management (DSOM). (2003) 28–40
8. Burgess, M.: Testable system administration. Commun. ACM **54**(3) (2011) 44–49
9. Opscode Community: http://community.opscode.com/
10. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability **22**(5) (2012) 297–312
11. Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Software Testing, Verification and Reliability **13** (2003) 25–53
12. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comp. Surv. (2011)
13. Helland, P.: Idempotence is not a medical condition. ACM Queue **10**(4) (2012)
14. Helland, P., Campbell, D.: Building on quicksand. In: Conference on Innovative Data Systems Research (CIDR). (2009)
15. Traugott, S.: Why order matters: Turing equivalence in automated systems administration. In: 16th Conference on Systems Administration (LISA). (2002) 99–120
16. Zamboni, D.: Learning CFEngine 3: Automated system administration for sites of any size. O'Reilly Media, Inc. (2012)
17. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
18. Giurgiu, I., Castillo, C., Tantawi, A., Steinder, M.: Enabling efficient placement of virtual infrastructures in the cloud. In: 13th Int. Middleware Conf. (2012) 332–353
19. ChefSpec: https://github.com/acrmp/chefspec
20. Cucumber-puppet: http://projects.puppetlabs.com/projects/cucumber-puppet
21. Test Kitchen: https://github.com/opscode/test-kitchen
22. Pretschner, A.: Model-based testing. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. (may 2005) 722–723
23. Cadar, C., Godefroid, P., et al.: Symbolic execution for software testing in practice: preliminary assessment. In: 33rd Int. Conf. on Software Engineering (ICSE). (2011)
24. Navarro, L.D., Douence, R., Südholt, M.: Debugging and testing middleware with aspect-based control-flow and causal patterns. In: 9th Middleware. (2008) 183–202
25. Hummer, W., Raz, O., Shehory, O., Leitner, P., Dustdar, S.: Testing of data-centric and event-based dynamic service compositions. Softw. Test., Verif. & Reliab. (2013)
26. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: ACM EuroSys Conf. (2011) 183–198
27. Candea, G., Bucur, S., Zamfir, C.: Automated software testing as a service. In: 1st ACM Symposium on Cloud Computing (SoCC). (2010) 155–160
28. van der Burg, S., Dolstra, E.: Automating system tests using declarative virtual machines. In: 21st Int. Symposium on Software Reliability Engineering. (2010)
29. Casale, G., Kalbasi, A., Krishnamurthy, D., Rolia, J.: Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In: 10th International Middleware Conference. (2009) 20:1–20:20
30. Gambi, A., Hummer, W., Truong, H.L., Dustdar, S.: Testing Elastic Computing Systems. IEEE Internet Computing (2013)
31. Whitaker, A., Cox, R., Gribble, S.: Configuration debugging as search: finding the needle in the haystack. In: Symp. on Op. Sys. Design & Impl. (OSDI). (2004) 6–6
32. Su, Y.Y., Attariyan, M., Flinn, J.: AutoBash: improving configuration management with operating system causality analysis. In: SOSP. (2007)