Asserting Reliable Convergence for Configuration Management Scripts

Oliver Hanappi, Waldemar Hummer, Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, Austria
{ohanappi, hummer, dustdar}@dsg.tuwien.ac.at

Abstract

The rise of elastically scaling applications that frequently deploy new machines has led to the adoption of DevOps practices across the cloud engineering stack. So-called configuration management tools utilize scripts that are based on declarative resource descriptions and make the system converge to the desired state. It is crucial for convergent configurations to be able to gracefully handle transient faults, e.g., network outages when downloading and installing software packages. In this paper we introduce a conceptual framework for asserting reliable convergence in configuration management. Based on a formal definition of configuration scripts and their resources, we utilize state transition graphs to test whether a script makes the system converge to the desired state under different conditions. In our generalized model, configuration actions are partially ordered, often resulting in prohibitively many possible execution orders. To reduce this problem space, we define and analyze a property called preservation, and we show that if preservation holds for all pairs of resources, then convergence holds for the entire configuration. Our implementation builds on Puppet, but the approach is equally applicable to other frameworks like Chef, Ansible, etc. We perform a comprehensive evaluation based on real world Puppet scripts and show the effectiveness of the approach. Our tool is able to detect all idempotence and convergence related issues in a set of existing Puppet scripts with known issues as well as some hitherto undiscovered bugs in a large random sample of scripts.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.9 [*Management*]: Software Configuration Management; D.3.3 [*Programming Languages*]: Language Constructs and Features

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984000

Keywords Configuration Management, Idempotence, Convergence, DevOps, Testing, System Configuration Scripts, Declarative Language, Puppet

1. Introduction

In recent years, with the rise of cloud computing and large dynamically scaling distributed applications, the demand for automatic provisioning of IT infrastructures has grown steadily [34, 16]. This trend is accompanied by the rise of the DevOps movement [20], which emphasizes the reintegration of development and operations, to close the technical and mental gap between these two, traditionally separated, roles. To enable this transition, robust and repeatable software deployment processes are of paramount importance.

This state of affairs has led to the adoption of configuration management tools [29, 6], which automatically deploy and configure software systems. Such tools operate on a declarative description (denoted *configuration specification*) which represents different aspects (denoted *resources*) concerning the desired system state. The resource actions are repeatedly applied, trying to configure the system accordingly. They may fail temporarily but should eventually succeed, i.e., the system is said to *converge* to the desired state. Furthermore, once the system has been set up appropriately, it should no longer get changed by the configuration script. A crucial prerequisite for convergence is to execute idempotent actions [11], meaning that repeated executions do not fail or change the system once the action has succeeded.

Developing robust and reusable configurations is a challenging task, as they must be able to converge in various system environments under changing conditions, and should gracefully handle transient faults, e.g., network outages when downloading and installing software packages. Hence, systematic testing support for developers is essential.

In this paper, we apply a model based testing approach [31] and develop a formal framework upon which our test procedure is based. We define a property called *preservation* which can be tested pairwise between resources and which, assuming that it holds for all pairs of resources, implies convergence of the entire configuration. The implication is formally proven, thus the effectiveness of the test approach relies solely on the ability and reliability of testing the preservation property. Our implementation systematically executes configuration tests in various settings and automatically analyzes the state changes. We demonstrate that our approach effectively detects idempotence and convergence issues in a large sample of real-world Puppet [22] configuration scripts.

The remainder of this paper is structured as follows. Section 2 introduces an illustrative scenario to motivate the research problem. In Section 3 we discuss related work in the area of configuration management languages and tools. We provide a formal system model in Section 4, establish a formal notion of convergence in Section 5, and detail our testing approach in Section 6. Section 7 contains selected implementation details, and in Section 8 we summarize the results of the comprehensive evaluation we have conducted. Finally, Section 9 concludes the paper with outlook for future work.

2. Scenario

We introduce a short example to illustrate the subtle idempotence and convergence problems that can occur in declarative configuration scripts. Listing 1 shows a Puppet manifest that is based on a popular community script used to download and install a *Glassfish* application server.

```
exec { 'download':
  command => 'wget http://glassfish.org/latest.zip
              -O /tmp/gf.zip',
   creates => '/tmp/gf.zip',
   unless  => 'test -e /usr/bin/glassfish'
}
exec { 'unzip':
  command => 'unzip /tmp/gf.zip -d /opt/glassfish',
   onlyif  => 'test -e /tmp/gf.zip',
   require => Exec['download']
}
file { 'remove':
   path    => '/tmp/gf.zip',
   ensure  => absent,
   require => Exec['unzip']
}
exec { 'install':
  command => 'make -C /opt/glassfish install',
   creates => '/usr/bin/glassfish',
   require => Exec['unzip']
}
```

Listing 1. Exemplary Puppet Manifest (Installs Glassfish)

The Puppet manifest consists of the following resources:

Download fetches a zip archive containing the binaries from an HTTP server. If the file is already present or Glassfish is already installed, the download is skipped.

Unzip is executed after downloading and extracts the archive to a temporary directory if the archive is present.

Remove deletes the archive after `unzip` has been executed.

Install deploys Glassfish after `unzip` has finished. The resource is only applied if Glassfish is not already installed.

At first glance this Puppet script seems to be idempotent and therefore should let the system converge to a state in which Glassfish is installed and the archive is deleted. This is true as long as no failures occur during execution.

Now consider that the `install` step fails for some reason (e.g., because disk space is exhausted, or the system temporarily has too many open file descriptors). In this case the archive gets deleted, as there is no dependency to the failing `install` resource. When re-executing the script, the archive is downloaded again as the `download` resource checks whether Glassfish is installed rather than whether the archive has already been extracted. As long as the installation fails, the system constantly transitions between states in which the archive is present and not. There is no common desired state in which both resources, `download` and `delete`, can be satisfied because their goals are conflicting.

To make matters worse, the implementation of the `unzip` resource is not idempotent. Upon re-execution, `unzip` asks the user whether the existing files should be overwritten and interprets a missing user interaction as negative answer, which results in a non-zero exit code (indicating an error). Puppet will skip all successor actions, which leads to a stuck configuration that will never converge to the desired state.

These particular issues related to idempotence and convergence are hard to detect because it is often difficult to test for partial executions manually. Normally the script succeeds and the `unzip` action is thus skipped on re-executions, leaving such dormant bugs undetected by the developer.

In order to resolve the problems outlined above, the script can be modified as follows. First, the `unzip` action can be easily made idempotent by adding the parameter `-u`, which causes unzip to either create new or update changed files.

The conflict between `download` and `remove` can be resolved by slightly adjusting their semantics. Recall that these resources are conflicting with each other in any system state in which Glassfish is not installed: `download` ensures that the archive is present, whereas `remove` ensures the opposite.

One could adjust `remove` such that it only deletes the archive once Glassfish is installed. Hence, `remove` will only delete the archive in states in which it is no longer fetched by `download`. Vice versa, `download` will not restore the archive once deleted because Glassfish is already installed.

Alternatively, `download` could be adapted to only fetch the archive as long as it has not already been extracted. Observe that, due to the dependency relation, `remove` is only executed after successful extraction, such that there is no state in which both resources try to configure the system.

3. Related Work

The DevOps movement [20, 15] emphasizes the reintegration of development and operations and has therefore strongly influenced the adoption of automated configuration management tools. Agile methods commonly used in DevOps enable quick feedback loops in principle, however traditional IT operations may prevent these in practice if deployments are slow and occur infrequently. Automated configuration management is therefore identified as an important aspect which enables continuous delivery [24].

Configuration management is a large and diverse field. According to [6] it is the process of constraining a set of machines to adhere to a given policy while maximizing conformance and minimizing operational costs. The authors discuss the many diverse practices used in this field, and identify convergence as a key common element. Criticism is placed on the separation of configuration management and monitoring, and it is stated that an integration of both should be strived for in the next generation of tools and frameworks.

Surveys on popular tools can be found in [1, 13, 28]. The most popular ones are, amongst many others, CFEngine [5], Chef [30], Puppet [21] and Ansible [18].

To the best of our knowledge, there exists little work on automatic testing of configurations scripts focusing on idempotence and convergence. Some earlier work [19] has presented a blackbox approach for automatically testing Chef cookbooks for idempotence, by systematically executing subsequences of the contained actions. During execution, changes to the system are analyzed in order to conclude whether cookbooks are idempotent or not. While [19] assumes total order of configuration actions, we consider the more complex case of partially ordered actions.

Issues in convergent processes arise when actions are inconsistent or in conflict with each other, hence do not share a common desired target state. A statistical approach for ensuring consistency in a probabilistic way is presented in [9]. It is shown that if convergence cannot be reached within a certain time frame, then this situation is a probabilistic indicator for conflicts in a configuration specification.

A tool for statically verifying Puppet manifests for determinism is presented by Collard et al. [8]. Determinism states that a configuration has the same effect on any machine and thus is a complementary concept to convergence. The authors propose a simple imperative language for file system modification with clearly defined semantics. All resources are compiled and translated to logical formulas, which allow reasoning about determinism. Although the paper identifies and tackles very similar problems, both approaches are fundamentally different from each other: We focus on testing whereas the other approach implements static verification. Generally, verification can be considered superior to testing in terms of reliably detecting issues, but at the same time it constitutes a considerably more constrained method. The paper requires that all resources have clearly defined semantics, which, arguably, is difficult to achieve in real world scenarios. It is a common practice to utilize resources which invoke external binaries and shell scripts, whose formalization would presumably require a substantial amount of effort.

Couch and Sun [12] present a method for observing reproducibility, based on a formal model which distinguishes actual and observed system states. Latent preconditions are differences in actual system states that are not observed, which may lead to varying results when configuring hosts. Reproducibility on a single host can be statically verified when using a limited set of primitives, however, for multiple machines sharing the observed state this is more difficult. We build on their conclusion that configuration management needs to include state measuring as an intrinsic component. Any system aspects that are once managed need to be controlled in any future execution to avoid latent preconditions.

Erdweg et al. [14] study functional and non-functional properties of incremental build systems, which need to run different build tools on several files in the right order to create a desired output (all files properly compiled and assembled). The paper contributes a framework for provably sound and optimal incremental builds. Their model of dependency graphs and re-executions has similarities with our model, and the notion of idempotent build systems has inspired our approach for testing convergent configuration scripts.

4. System Model for Configuration Scripts

This section introduces a formal model for configuration scripts, discusses how they are executed within a system, and which properties are tested to conclude that the script models an idempotent and convergent configuration process.

4.1 States and Actions

The configured system is modeled as a possibly infinite set of *states* $\mathcal{S}$ which can transition between each other [2]. States can be compared by the $=$ equivalence relation. Our formal approach is agnostic of the particular system on which it is implemented as long as it is possible to recognize equivalent states or differences in states, respectively.

An action $a : \mathcal{S} \rightarrow \mathcal{S} \times \{\top, \bot\}$ is a transition function which accepts an input state and returns an output state as well as an exit code indicating success ($\top$) or failure ($\bot$).

Actions are assumed to be deterministic, that is, given the same input and under the same conditions (input state) they should always lead to the same outcome (output state). While it is true for real world scenarios that actions may behave non-deterministically based on external factors, e.g., network outage when downloading a file from a remote source, we are primarily interested in detecting bugs and issues whose logic is deterministic and reproducible.

Furthermore, actions are considered to be atomic. Obviously, under real world conditions, actions may execute partially or are composed of several atomic operations, e.g., deleting individual files when removing an entire directory. We do not explicitly model non-atomic actions, however, upon re-execution of an interrupted action additional state changes are expected due to the partial state, hence our approach can effectively detect non-atomic and partial actions.

Two actions are equal iff they behave exactly the same way, which is expressed in Definition 1 as matching output states and exit codes for any state on which they are applied.

Definition 1. Two actions a and b are *equivalent*, denoted $a = b$, iff $\forall s \in \mathcal{S} : a(s) = b(s)$.

Actions can be *composed* by the $\triangleright$ operator defined in Equation (1). A composed action $a \triangleright b$ first applies a on the input state and then b on the output state of a. The subsequent action b is only applied if the former action a succeeded, otherwise the composed action aborts and fails.

$$(a \triangleright b)(s) = \begin{cases} b(s') & \text{if } a(s) = (s', \top) \\ (s', \bot) & \text{if } a(s) = (s', \bot) \end{cases} \tag{1}$$

Note that action composition is associative, as shown in Appendix A.1. To improve readability we skip parentheses for action compositions in the remainder of this paper.

Definition 2 defines action *idempotence*: An action is idempotent iff it does not change the system state on re-execution once it has been successfully applied to the system. Put simply, applying the action twice (or multiple times) has the same effect as applying it only once.

Definition 2. An action a is *idempotent* iff $a = a \triangleright a$.

Observe that an idempotent action must not fail once it has been executed successfully, as discussed in [19]. This follows directly from the definition of idempotence.

Proof. Let a be an idempotent action and s a state such that $a(s) = (s', \top)$. It follows that a does not fail when applied to s'. Assume that a fails on s', hence $a(s') = (s'', \bot)$. It follows that $(a \triangleright a)(s) = a(s') = (s'', \bot) \neq (s', \top) = a(s)$, which contradicts with the idempotence of a. □

4.2 Action Sequences

An *action sequence* $z = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequential composition of $|z| = n$ actions which are applied in the order $a_1, a_2, \ldots, a_n$, hence a_1 is the first and a_n the last action to be applied to the system. The corresponding composed action $\overline{z}$ of z is defined in Equation (2).

$$\overline{z} = a_1 \triangleright a_2 \triangleright \cdots \triangleright a_n \tag{2}$$

When an action sequence z is applied to a state s, the system transitions several times between the intermediate states $\overline{z/i}(s)$ defined in (3). Observe that $\overline{z/n} = \overline{z}$ holds.

$$\begin{aligned} \overline{z/i} &= \begin{cases} a_1 & \text{if } i = 1 \\ \overline{z/i-1} \triangleright a_i & \text{if } i > 1 \end{cases} \\ &= a_1 \triangleright a_2 \triangleright \cdots \triangleright a_i \end{aligned} \tag{3}$$

The notion of action idempotence can be extended to action sequences, as shown in Definition 3.

Definition 3. Action sequence z is *idempotent* iff $\overline{z} = \overline{z} \triangleright \overline{z}$.

As discussed in [11], a sequence of idempotent actions is not necessarily idempotent as well. For instance, consider the following system: $\mathcal{S} = \mathbb{Z}$ (for illustration, the entire system state is represented as a single integer), $a_1(s) = (s, \top)$ if $s < 0$, $a_1(s) = (0, \top)$ if $s \geq 0$, and $a_2(s) = (|s|, \top)$. Although a_1 and a_2 are idempotent, $a_1 \triangleright a_2$ is not, e.g. for state -1: $(a_1 \triangleright a_2)(-1) = (1, \top)$, but $(a_1 \triangleright a_2)(1) = (0, \top) \neq (1, \top)$.

Therefore, the notion of *statelessness* [11] between two actions a and b expresses that the changes made by applying a do not depend on those made by b. Formally, a is stateless w.r.t. b iff $b \triangleright a \triangleright b = a \triangleright b$. It follows that any action sequence constructed from a set of idempotent and pairwise stateless actions is idempotent as well. Note that an action sequence that is idempotent according to Definition 3 may temporarily change the system state during the re-execution as long as the output state of the re-execution still equals its input state. This is also true if the actions are pairwise stateless.

For instance, consider the system $\mathcal{S} = \mathbb{Z}$ (again, the entire system state is described by a single integer), $a_1(s) = (1, \top)$ and $a_2(s) = (2, \top)$ in which a_1, a_2 and $z = \langle a_1, a_2 \rangle$ are idempotent and a_1, a_2 pairwise stateless. The output state of $\overline{z}$ for any input state is always state 2 and each re-execution also results in state 2, however $\overline{z/1}(2) = (1, \top) \neq (2, \top)$, hence state state 2 is temporarily left after executing action a_1.

This notion of idempotence usually does not accurately capture the intended meaning of reliable configuration in a real system, as it may lead to subtle issues like those outlined in Section 2. We therefore utilize the notion of *convergence* [10], which requires that there are no changes to the system at all, once the desired state has been reached.

Definition 4. An action sequence z is *convergent* iff for any $s \in \mathcal{S}$ such that $\overline{z}(s) = (s', \top)$ it holds that $\overline{z/i}(s') = (s', \top)$ is true for $1 \leq i \leq |z|$.

Definition 4 incorporates the idea that once the action sequence put the system into the desired state s' in which it exits successfully, it does not (even temporarily) leave this state on re-executions. Note that this definition particularly accounts for configuration scripts which are repeatedly executed with unsuccessful exit code ($\overline{z}(s_0) = (s_1, \bot), \overline{z}(s_1) = (s_2, \bot), \ldots, \overline{z}(s_{n-1}) = (s_n, \bot)$) until they eventually converge and successfully reach the target state $\overline{z}(s_n) = (s', \top)$.

As shown by Theorem 1, convergence is a stronger property than idempotence and thus implies it.

Theorem 1. A convergent action sequence is idempotent.

Proof. Let z be a convergent action sequence. We show that z is idempotent, hence $\overline{z}(s) = (\overline{z} \triangleright \overline{z})(s)$ holds for any $s \in \mathcal{S}$.

- Assume $\overline{z}(s) = (s', \bot)$. It follows from the definition of action composition that $(\overline{z} \triangleright \overline{z})(s) = (s', \bot)$, hence $\overline{z}(s) = (s', \bot) = (\overline{z} \triangleright \overline{z})(s)$.
- Assume $\overline{z}(s) = (s', \top)$. Observe that $\overline{z} = \overline{z/|z|}$ holds by definition. It follows from the convergence of z that $\overline{z/|z|}(s') = (s', \top) = \overline{z}(s')$ and by the definition of action composition that $(\overline{z} \triangleright \overline{z})(s) = \overline{z}(s') = (s', \top)$, hence $\overline{z}(s) = (s', \top) = (\overline{z} \triangleright \overline{z})(s)$. □

4.3 Configuration Specifications

A *configuration specification* $c = (R_c, \prec_c)$ is a set of resources $R_c = \{r_1, ..., r_n\}$ and their dependency relation $\prec_c$. It is typically encoded in a *configuration script* with domain-specific language constructs (e.g., Puppet, Chef, Ansible, ...).

A *resource* $r \in R_c$ represents a certain aspect to be configured, e.g., in our scenario the presence of a file (resource `download`) or the installation of a software program (resource `install`). A resource is applied to a system by its corresponding action $\overline{r}$. A resource is *satisfied* by a state $s \in \mathcal{S}$, denoted as $s \models r$, iff $\overline{r}(s) = (s, \top)$. Note that resource satisfaction is based on the assumption that the implementation of the resource's action is idempotent.

The *dependency relation* $\prec_c$ is a transitive, irreflexive relation that defines a strict partial order over R_c. In our scenario, the resource `install` depends on `unzip`, and transitively on `download`. Some configuration management tools assume a total order (e.g., Chef uses sequential execution of resources), which can also be captured by our model.

The *ancestors* and *successors* relations are defined in Equations (4) and (5), respectively. The ancestors A_r of a resource r are all resources which must be satisfied by the system prior to applying r (according to relation $\prec_c$).

$$A_r = \{r' \in R_c \mid r' \prec_c r\} \tag{4}$$
$$S_r = \{r' \in R_c \mid r \prec_c r'\} \tag{5}$$

A resource r fails when being applied to a state which does not satisfy one of the resource's ancestors in A_r. For instance, in our scenario it is expected that the `unzip` resource fails in a state not satisfying the `download` resource (i.e., when the archive has not been downloaded yet).

Finally, *non-related resources* I_r of a resource r are all resources whose execution is independent from r according to the dependency relation, see Equation (6). In our scenario, the resources `remove` and `install` are non-related.

$$I_r = \{r' \in R_c \mid r' \not\prec_c r \land r \not\prec_c r' \land r' \neq r\} \tag{6}$$

Observe that A_r, S_r and I_r are distinct sets and that $R_c = A_r \cup S_r \cup I_r \cup \{r\}$ is true for any resource $r \in R_c$.

4.4 Configuration Executions

A configuration specification c is applied to a system by applying all its resources, satisfying the dependencies in $\prec_c$.

An *execution order* $\pi : R_c \to \{1, 2, \ldots, |R_c|\}$ is a bijective (one-to-one) mapping from the set of resources to a set of execution positions. Position 1 is executed first, position 2 second, and position $|R_c|$ last. The inverse function π^{-1} maps each execution position to the corresponding resource.

The execution order defines an action sequence c_π, as shown in Equation (7), whose composed action $\overline{c_\pi}$ applies all resources of c corresponding to execution order π.

$$c_\pi = \langle \pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(|R_c|) \rangle \tag{7}$$

The set Π_c of valid execution orders of the configuration specification c is defined in Equation (8). B_c denotes the set of all bijections from R_c to $\{1, 2, \ldots, |R_c|\}$.

$$\Pi_c = \{\pi \in B_c \mid \forall r_1, r_2 \in R_c : r_1 \prec_c r_2 \Rightarrow \pi(r_1) < \pi(r_2)\} \tag{8}$$

Selecting a valid execution order corresponds to choosing a linear extension [3] of the partial order $\prec_c$. It follows that configuration executions are intrinsically non-deterministic as there are usually multiple possible linear extensions.

Note that the problem space of possible execution orders can be huge. Determining the number of all linear extensions for a given partial order is a #P-complete problem (#P is the class of counting problems which determine the number of solutions of decision problems in NP [32]). However, there are efficient polynomial time approximation algorithms [4].

5. Idempotence and Convergence

This section introduces the concepts of idempotent and convergent configuration specifications, which serve as the basis for our test approach discussed later in Section 6.

5.1 Definition of Idempotence and Convergence

Definition 5 is based on the definitions of idempotence for actions and action sequences but reflects the non-deterministic nature of execution orders. The repeated execution of c is not required to adhere to the same execution order π_1, but may be executed in arbitrary order π_2.

Definition 5. Configuration specification c is *idempotent* iff $\overline{c_{\pi_1}} = \overline{c_{\pi_1}} \triangleright \overline{c_{\pi_2}}$ for any pair of execution orders $\pi_1, \pi_2 \in \Pi_c$.

The same considerations about convergence of action sequences apply to configuration specifications as well, summarized by Definition 6. A convergent configuration specification does not change the desired state reached by execution order π_1 on re-execution with an arbitrary order π_2.

Definition 6. A configuration specification c is *convergent* iff for any pair of execution orders $\pi_1, \pi_2 \in \Pi_c$ and for any state $s \in \mathcal{S}$ s.t. $\overline{c_{\pi_1}}(s) = (s', \top)$ it holds that $\overline{c_{\pi_2}/i}(s') = (s', \top)$ is true for $1 \leq i \leq |R_c|$.

Analogous to action sequences, configuration specification convergence is a stronger property than idempotence as well, thus the former implies the latter (see Theorem 2).

Theorem 2. A convergent configuration specification is idempotent.

Proof. Let c be a convergent configuration specification. We need to show that $\overline{c_{\pi_1}}(s) = (\overline{c_{\pi_1}} \triangleright \overline{c_{\pi_2}})(s)$ for any execution orders $\pi_1, \pi_2 \in \Pi_c$ and any state $s \in \mathcal{S}$.

- Assume $\overline{c_{\pi_1}}(s) = (s', \bot)$. It follows from the definition of action composition that $(\overline{c_{\pi_1}} \triangleright \overline{c_{\pi_2}})(s) = (s', \bot)$, hence $\overline{c_{\pi_1}}(s) = (s', \bot) = (\overline{c_{\pi_1}} \triangleright \overline{c_{\pi_2}})(s)$.

- Assume $\overline{c_{\pi_1}}(s) = (s', \top)$. Note that $\overline{c_{\pi_2}}(\hat{s}) = \overline{c_{\pi_2}/|R_c|}(\hat{s})$ for any $\hat{s} \in \mathcal{S}$ by definition. It follows from convergence of c that $\overline{c_{\pi_2}/|R_c|}(s') = (s', \top) = \overline{c_{\pi_2}}(s')$ and by the definition of action composition that $(\overline{c_{\pi_1}} \triangleright \overline{c_{\pi_2}})(s) = \overline{c_{\pi_2}}(s') = (s', \top)$, hence $\overline{c_{\pi_1}}(s) = (s', \top) = (\overline{c_{\pi_1}} \triangleright \overline{c_{\pi_2}})(s)$.

□

5.2 Resource Preservation

The definitions provided in Section 5.1 are not well suited as formal foundation for generating test cases for configuration specifications because there are typically too many execution orders to test, as discussed in Section 4.4.

This section therefore introduces the notion of resource *preservation* in Definition 7, a property which can be tested pairwise between two resources. In Section 5.3, this definition is used to present Theorem 3, which shows that pairwise preservation implies convergence.

Definition 7. Resource b *preserves* resource a iff for any state $s \in \mathcal{S}$ satisfying a ($s \models a$), the state s' after applying b, $\overline{b}(s) \in \{(s', \top), (s', \bot)\}$, satisfies a as well ($s' \models a$).

Informally, a resource b is said to preserve another resource a if resource a is still satisfied after applying b. The resources thus manage different aspects of the system and do not conflict with each other or they agree on the way a shared asset shall be configured. Expressed differently, we can say that resource b preserves resource a iff $\overline{a} \triangleright \overline{b} \triangleright \overline{a} = \overline{a} \triangleright \overline{b}$.

5.3 Preservation Convergence Theorem

The concept of resource preservation is used to enable testing for idempotence and convergence in a feasible amount of time. In the following, we show that we can reduce the testing problem from the (infeasible) space of all possible execution orders to the (feasible) space of testing pairwise resource preservation. In particular, for a configuration specification to be convergent, each resource needs to preserve its respective ancestors and non-related resources, as shown by Theorem 3 in this section. First, we introduce Lemma 1 which is the basis for the proof of Theorem 3.

Lemma 1. Let c be a configuration specification whose resources are idempotent and preserve their respective ancestors and non-related resources. Any successful execution of c results in a state satisfying every resource.

Proof. Let c be a configuration specification whose resources preserve their respective ancestors and non-related resources. Let $s \in \mathcal{S}$ be an arbitrary state and $\pi \in \Pi_c$ a valid execution order of c such that $\overline{c_\pi}(s) = (s', \top)$. We need to show that s' satisfies any resource $r \in R_c$.

Let E_i denote the set of the first i applied resources by c_π, which is defined as $E_i = \{r_1, r_2, \ldots, r_i\}$ with $r_i = \pi^{-1}(i)$. Observe that $E_{|R_c|} = R_c$.

We show by induction over i that for any $1 \le i \le |R_c|$ with $\overline{c_\pi/i}(s) = (s_i, \top)$ it holds that $\forall r \in E_i : s_i \models r$. Note that $\overline{c_\pi/i}(s) = (s_i, \top)$ succeeds for any $1 \le i \le |R_c|$ because $\overline{c_\pi}(s) = (s', \top)$.

- Base case for $i = 1$: $\forall r \in E_1 : s_1 \models r$. Observe that $E_1 = \{r_1\}$ and by definition $\overline{c_\pi/i} = \overline{r_1}$, hence $\overline{c_\pi/i}(s) = \overline{r_1}(s) = (s_1, \top)$. It follows from the idempotence of $\overline{r_1}$ that $\overline{r_1}(s_1) = (s_1, \top)$, hence $s_1 \models r_1$ and thus $\forall r \in E_1 : s_1 \models r$.
- Inductive step for $i + 1$: $\forall r \in E_{i+1} : s_{i+1} \models r$. The induction hypothesis is $\forall r \in E_i : s_i \models r$. Observe that $E_{i+1} = E_i \cup \{r_{i+1}\}$. Note that by definition $\overline{c_\pi/i+1} = \overline{c_\pi/i} \triangleright \overline{r_{i+1}}$. It follows from $\overline{c_\pi/i}(s) = (s_i, \top)$ and by the definition of action composition that $\overline{c_\pi/i+1}(s) = (\overline{c_\pi/i} \triangleright \overline{r_{i+1}})(s) = \overline{r_{i+1}}(s_i) = (s_{i+1}, \top)$. It follows from the idempotence of $\overline{r_{i+1}}$ that $\overline{r_{i+1}}(s_{i+1}) = (s_{i+1}, \top)$, hence $s_{i+1} \models r_{i+1}$.
  It remains to show that $\forall r \in E_i : s_{i+1} \models r$. Observe that every resource in E_i is either an ancestor or a non-related resource but not a successor of r_{i+1}. A successor depends on r_{i+1} and therefore needs to be applied after r_{i+1}, otherwise π would not be a valid execution order. It follows that r_{i+1} preserves every $r \in E_i$. Furthermore, we know from the induction hypothesis that s_i satisfies every $r \in E_i$ and thus that s_{i+1} satisfies every $r \in E_i$ due to preservation.

Observe that $\overline{c_\pi/|R_c|} = \overline{c_\pi}$ and thus $s_{|R_c|} = s'$. It follows that s' satisfies every resource in $E_{|R_c|}$, hence it satisfies every resource in R_c. □

Theorem 3. A configuration specification whose resources are idempotent and preserve their respective ancestors and non-related resources is convergent.

Proof. Let c be a configuration specification whose resources preserve their respective ancestors and non-related resources, $\pi_1, \pi_2 \in \Pi_c$ any execution orders and $s \in \mathcal{S}$ any state such that $\overline{c_{\pi_1}}(s) = (s', \top)$.

We show that $\overline{c_{\pi_2}/i}(s') = (s', \top)$ for $1 \le i \le |R_c|$ by induction over i.

- Base case $i = 1$: $\overline{c_{\pi_2}/1}(s') = (s', \top)$. Note that $\overline{c_{\pi_2}/1}$ is defined as $\overline{r}$ with $r = \pi_2^{-1}(1)$. We know from Lemma 1 that $s' \models r$, hence $\overline{r}(s') = (s', \top)$ holds for the resource r. It follows that $\overline{c_{\pi_2}/1}(s') = \overline{r}(s') = (s', \top)$.
- Inductive step for $i + 1$: $\overline{c_{\pi_2}/i+1}(s') = (s', \top)$. The induction hypothesis is $\overline{c_{\pi_2}/i}(s') = (s', \top)$. Note that $\overline{c_{\pi_2}/i+1}$ is defined as $\overline{c_{\pi_2}/i} \triangleright \overline{r}$ with $r = \pi_2^{-1}(i + 1)$, hence $\overline{c_{\pi_2}/i+1}(s') = (\overline{c_{\pi_2}/i} \triangleright \overline{r})(s')$. It follows from the induction hypothesis and the definition of action composition that $(\overline{c_{\pi_2}/i} \triangleright \overline{r})(s') = \overline{r}(s')$. We know from Lemma 1 that $s' \models r$ and thus $\overline{r}(s') = (s', \top)$ holds for the resource r, hence $\overline{c_{\pi_2}/i+1}(s') = (\overline{c_{\pi_2}/i} \triangleright \overline{r})(s') = \overline{r}(s') = (s', \top)$.

□

Note that although preservation of ancestors and non-related resources implies idempotence and convergence, the inverse implications are not true. The respective theorems are added in Appendices A.2 and A.3. However, we believe that such cases are rare in practice. The proof of the two theorems builds upon resources which behave differently in states which cannot be reached by applying the configuration specification. We argue that such resources are hardly ever encountered in real world scenarios.

6. Test Approach

This section introduces our approach for testing the idempotence and convergence of a given configuration specification using model-based testing [31]. The overall goal of the test approach is to test whether or not a given configuration specification is convergent according to Definition 6.

As stated by Theorem 3, a configuration specification is convergent if its resources preserve their respective ancestors and non-related resources. Hence, our approach focuses on pairwise testing of preservation among resources.

Observe that resource actions are required to be idempotent by definition. Furthermore, as discussed above, testing preservation relies on the notion of resource satisfaction, which is based on the assumption of idempotent resource actions. Due to the fact that implementing idempotent actions in real systems is a challenging task [19, 34], we also test whether individual resource actions are idempotent or not.

In summary, our approach tests the following properties for each resource $r \in R_c$ of configuration specification c:

- $\overline{r}$ is idempotent
- for any ancestor $a \in A_r$: r preserves a
- for any non-related resource $r' \in I_r$: r preserves r'

6.1 Test Cases

A *test step* is an action associated with a resource and either applies that resource to the system or checks whether the system satisfies the resource. An execution step for a resource r is denoted as $exec\langle r\rangle$ and defined as (9), hence it simply executes the resource action and succeeds only if the resource action succeeds. It is expected that an execution step performs a configuration change.

An assertion step is denoted as $assert\langle r\rangle$ and defined as (10). Observe that an assertion step only succeeds on some state $s \in \mathcal{S}$ iff s satisfies r. Hence, it is expected that an assertion step does not reconfigure the system.

$$exec\langle r\rangle(s) = \overline{r}(s) \tag{9}$$

$$assert\langle r\rangle(s) = \begin{cases} (s', \top) & \text{if } \overline{r}(s) = (s', \top) \wedge s' = s \\ (s', \bot) & \text{if } \overline{r}(s) = (s', \top) \wedge s' \neq s \\ (s', \bot) & \text{if } \overline{r}(s) = (s', \bot) \end{cases} \tag{10}$$

Given a set of resources $R = \{r_1, r_2, \ldots, r_k\} \subseteq R_c$ ordered according to some execution order $\pi \in \Pi_c$. As a shorthand notation we will write $exec\langle R, \pi\rangle$ instead of $exec\langle r_1\rangle \triangleright exec\langle r_2\rangle \triangleright \cdots \triangleright exec\langle r_k\rangle$ and $assert\langle R, \pi\rangle$ instead of $assert\langle r_1\rangle \triangleright assert\langle r_2\rangle \triangleright \cdots \triangleright assert\langle r_k\rangle$.

A *test case* is a sequence of test steps (or test actions). If the action sequence of a test case fails, we further distinguish whether it was an execution or an assertion step that has failed. If an execution step terminates unsuccessfully, the test case is said to *abort*, otherwise, if an assertion step returns an unsuccesful result, the test case is said to *fail*.

In contrast to an action sequence defined by a valid execution order of a configuration specification, a resource action may be executed more than once in a test case. Furthermore, not all resources need to be executed. In general, to filter out action sequences that are undesirable as test cases, we require that a test case satisfies the following properties:

- for any assertion step there exists a corresponding execution step which is executed prior to the assertion step
- the test case contains an execution step for every ancestor of any executed resource
- the order of execution steps adheres to the dependency relation
- there is at least one assertion step
- the last test step is an assertion step
- there is at most one execution step for each resource

Finally, a *test suite* is a set of desirable test cases which satisfy the test goal.

6.2 STG Based Test Case Generation

A state transition graph (STG) [2, 33] is a graph which describes states and transitions between them for a given configuration specification. It can be used to generate valid execution sequences by enumerating paths within the STG. A specific subset of paths can be used to attest all properties required to conclude that the configuration specification from which the STG was built is convergent.

6.2.1 Partitioned State Transition Graphs

An STG is a directed graph whose nodes represent system states and whose edges represent transitions between them. A directed edge labeled with a resource r means that applying the resource action $\overline{r}$ to the source state s_1 (also denoted pre-state) results in the destination state s_2 (also denoted post-state), as defined in Equation (11).

$$\begin{aligned} STG^*_c &= (\mathcal{S}, T^*_c) \\ T^*_c &= \{(s_1, s_2, r) \in \mathcal{S} \times \mathcal{S} \times R_c \mid \overline{r}(s_1) = (s_2, \top)\} \end{aligned} \tag{11}$$

STGs are (near-)infinite in the general case, due to the huge number of possible concrete states a system can be in. For instance, consider the possible pre-states of resource `unzip` in our scenario. The files to be extracted may already exist in the system, either all of them, none of them, or some of them, with arbitrary file content, etc. Typically, we are not interested in all possible concrete system states, but we want to summarize states into classes (or partitions) of states.

An infinite STG can be reduced to a finite *partitioned STG* by applying a partitioning scheme for system states and considering transitions between state partitions instead of concrete states. Nodes in a partitioned STG do not represent single states $s_1, s_2, \ldots \in \mathcal{S}$, but state partitions, which are sets of states $S_1, S_2, \ldots, S_k \subseteq \mathcal{S}$. State partitions must be non empty, pairwise disjoint ($\forall 1 \leq i < j \leq k : S_i \cap S_j = \emptyset$) and exhaustive ($S_1 \cup S_2 \cup \cdots \cup S_k = \mathcal{S}$).

We partition system states into state partitions by resource statisfaction. Each subset of resources $R \subseteq R_c$ corresponds to a state partition $S(R)$ in which every state $s \in S(R)$ satisfies every resource $r \in R$, but not any other resource $r' \in R_c \backslash R$. Hence, for the remainder of this paper we consider partitioned STGs whose nodes are sets of satisfied resources rather than sets of states. Clearly, the system transitions between states of the corresponding state partitions $S(\cdot)$. P_c denotes the set of all resource sets whose state partitions are non empty according to this partitioning scheme.

A transition (R_1, R_2, r) between nodes R_1 and R_2 signifies that applying the resource r to any state $s \in S(R_1)$ of the source node R_1 results in a state $s' \in S(R_2)$, which is part of the state partition of destination node R_2.

The model of the partitioned STG discussed above is outlined in (12) for further reference.

$$\begin{aligned} STG_c &= (P_c, T_c) \\ P_c &= \{R \in \mathcal{P}(R_c) \,|\, S(R) \neq \emptyset\} \\ S(R) &= \{s \in \mathcal{S} \,|\, (\forall r \in R : s \models r) \wedge \\ &\qquad (\forall r' \in R_c \backslash R : s \not\models r')\} \\ T_c &= \{(R_1, R_2, r) \in P_c \times P_c \times R_c \,|\, \forall s \in S(R_1) : \\ &\qquad \overline{r}(s) = (s', \top) \wedge s' \in S(R_2)\} \end{aligned} \tag{12}$$

For the rest of this paper we will only consider partitioned STGs. The partitioned STG for the script presented in Section 2 is depicted in Figure 1 (resource names are abbreviated). For illustration purposes we omit any self-referencing edges and we write node label d, u for resource set $\{d, u\}$ and its corresponding state partition $S(\{d, u\})$.

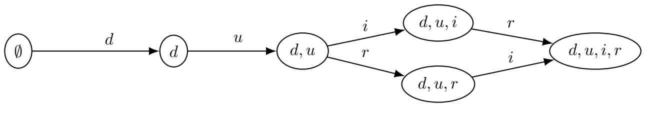

Figure 1. Partitioned STG of Scenario Script (assuming pairwise preservation)

6.2.2 Testing Properties

The generation of test cases is based on paths in the partitioned STG of a configuration specification whose resources are assumed to preserve their respective ancestors and non-related resources. A path $p = \langle R_1, \ldots, R_n \rangle$ is a sequence of distinct resource sets $R_1, \ldots, R_n$. Each transition $e_i = (R_i, R_{i+1}, r_i)$ between resource sets R_i and R_{i+1} is labeled with a resource r_i. Observe that $R_{i+1} = R_i \cup \{r_i\}$ holds due to the assumed resource preservation.

The path p can be executed on any state $s \in S(R_1)$ by applying all resources $r_1, \ldots, r_{n-1}$ according to the path transitions, as shown in Equation (13).

$$T_{exec}(p) \quad = \quad exec\langle r_1 \rangle \triangleright \cdots \triangleright exec\langle r_{n-1} \rangle \tag{13}$$

We aim at testing resource actions for idempotence and preservation along such path executions. Let e_i be a transition in p. Every execution of p will transition from some state $s_1 \in S(R_i)$ to $s_2 \in S(R_{i+1})$ such that $\overline{r_i}(s_1) = (s_2, \top)$.

We can test that r_i is idempotent by reapplying r_i to s_2 and check if the state is unchanged, hence executing $assert\langle r_i \rangle(s_2)$ after $exec\langle r_i \rangle(s_1)$. The idempotence attesting test case based on p is shown in Equation (14).

$$\begin{aligned} T_{idem}(p) \quad = \quad & exec\langle r_1 \rangle \triangleright assert\langle r_1 \rangle \triangleright \cdots \triangleright \\ & exec\langle r_{n-1} \rangle \triangleright assert\langle r_{n-1} \rangle \end{aligned} \tag{14}$$

In order to test preservation of any resource $r' \in R_i$ by r_i we can check whether r' is still satisfied in s_2, hence executing $assert\langle r' \rangle(s_2)$ after $exec\langle r_i \rangle(s_1)$. The test case which tests preservation for every resource in R_i (under some arbitrary order π) is shown in Equation (15). We write $assert\langle R_i, \pi \rangle$ for $assert\langle \pi^{-1}(1) \rangle \triangleright \cdots \triangleright assert\langle \pi^{-1}(i) \rangle$, where π solely determines the order of asserts in set R_i.

$$\begin{aligned} T_{pres}(p, \pi) \quad = \quad & exec\langle r_1 \rangle \triangleright assert\langle R_1, \pi \rangle \triangleright \cdots \triangleright \\ & exec\langle r_{n-1} \rangle \triangleright assert\langle R_{n-1}, \pi \rangle \end{aligned} \tag{15}$$

The test cases $T_{idem}(p)$ and $T_{pres}(p, \pi)$ are both derived from the execution test case $T_{exec}(p)$ and therefore share the same structure. Observe that $R_{i+1} = R_i \cup \{r_i\}$ holds, which allows the test cases to be merged as shown in Equation (16).

$$\begin{aligned} T_{STG}(p, \pi) \quad = \quad & exec\langle r_1 \rangle \triangleright assert\langle R_2, \pi \rangle \triangleright \cdots \triangleright \\ & exec\langle r_{n-1} \rangle \triangleright assert\langle R_n, \pi \rangle \end{aligned} \tag{16}$$

The test case $T_{STG}(p, \pi)$ attests idempotence of every executed resource along path p as well as the preservation of all ancestors of these resources. Furthermore, preservation of non-related resources which are executed prior to the resource itself in p is attested as well. Note that the execution order π determines only the order of assertion steps and not the order of resource execution, which is determined by p. Therefore, an arbitrary order may be used.

6.3 Test Path Selection Based on Minimal STG

The test goal describes resource properties which are tested pairwise, however when enumerating all possible paths in the partitioned STG, the same resource pairs and attested properties occur multiple times. For illustration, we consider a configuration specification c with less resource dependencies than in our scenario: $c = (\{r_1, r_2, r_3, r_4\}, \{(r_1, r_2)\})$.

The partitioned STG yields a path in which the resources are executed in order r_1, r_2, r_3, r_4 as well as a path with order r_3, r_4, r_1, r_2. We observe that the preservation of r_1 by r_2 is attested in both paths after executing r_2.

The *minimal* STG is a graph with the fewest state partitions and transitions such that 1) all relevant state partitions and transitions are present, and 2) there is a path from $\emptyset$ to any other state partition. There are potentially multiple minimal STGs, yet we expect the same test results among them.

Table 1 lists for each resource the state partitions and transitions required for testing. Figure 2 illustrates the graph constructed from these partitions and transitions, which is already the minimal STG. The dashed nodes and edges show the difference to the full partitioned STG.

Res.	Partitions and Transitions	Tested Properties
r_1	$\emptyset \xrightarrow{r_1} \{r_1\}$	idempotence of r_1
	$\{r_3\} \xrightarrow{r_1} \{r_1, r_3\}$	preservation of r_3 by r_1
	$\{r_4\} \xrightarrow{r_1} \{r_1, r_4\}$	preservation of r_4 by r_1
r_2	$\{r_1\} \xrightarrow{r_2} \{r_1, r_2\}$	idempotence of r_2, preservation of r_1 by r_2
	$\{r_1, r_3\} \xrightarrow{r_2} \{r_1, r_2, r_3\}$	preservation of r_3 by r_2
	$\{r_1, r_4\} \xrightarrow{r_2} \{r_1, r_2, r_4\}$	preservation of r_4 by r_2
r_3	$\emptyset \xrightarrow{r_3} \{r_3\}$	idempotence of r_3
	$\{r_1\} \xrightarrow{r_3} \{r_1, r_3\}$	preservation of r_1 by r_3
	$\{r_1, r_2\} \xrightarrow{r_3} \{r_1, r_2, r_3\}$	preservation of r_2 by r_3
	$\{r_4\} \xrightarrow{r_3} \{r_3, r_4\}$	preservation of r_4 by r_3
r_4	$\emptyset \xrightarrow{r_4} \{r_4\}$	idempotence of r_4
	$\{r_1\} \xrightarrow{r_4} \{r_1, r_4\}$	preservation of r_1 by r_4
	$\{r_1, r_2\} \xrightarrow{r_4} \{r_1, r_2, r_4\}$	preservation of r_2 by r_4
	$\{r_3\} \xrightarrow{r_4} \{r_3, r_4\}$	preservation of r_3 by r_4

Table 1. Required state partitions and transitions

Observe that only the two state partitions $\{r_1, r_3, r_4\}$ and $\{r_1, r_2, r_3, r_4\}$ are not part of the minimal STG. When dealing with larger, real-world configuration specifications, the difference between the minimal and full STG is much more significant and only the reduction to the minimal STG makes the approach feasible by drastically limiting the amount of generated test cases (see evaluation in Section 8.2).

6.4 Test Coverage

The STG based test case generation is built upon paths within the constructed STG. We restrict paths to start in state partition $S(\emptyset)$ because we assume that the "clean" test system (with no specified inital state) does not satisfy any resource upfront. An exact definition of a clean system highly depends on the configuration specification under test. However, given that most scripts are concerned with installing and configuring software, a freshly installed machine without any further configuration operations applied is a reasonable choice. Furthermore, we only consider cycle free paths, hence self-referencing edges are not taken into account.

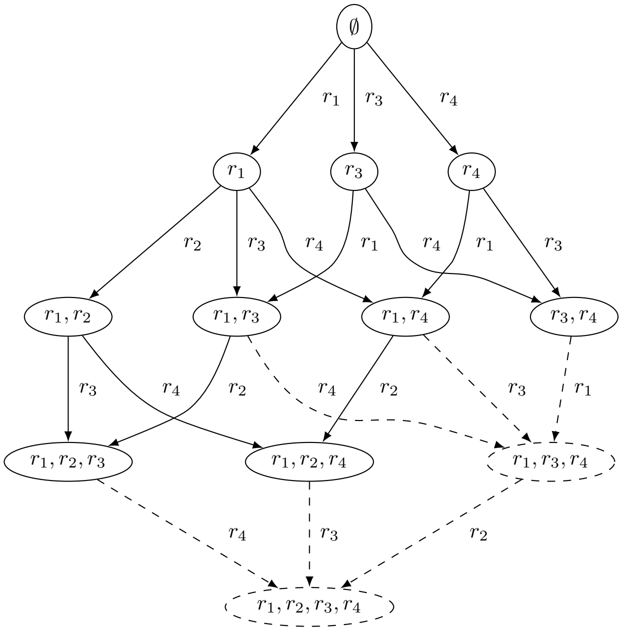

Figure 2. Minimal STG (solid) and Full STG (dashed) of c

The simplest approach is to enumerate all paths, which is known as *path coverage*. Paths start in the only source state partition $\emptyset$ and end in any sink state partition. A state partition is a sink iff it has no outgoing edges.

Another possible goal to drive test case generation is *edge coverage*. Each edge in the (partial) STG represents an idempotence and convergence related test, therefore every edge needs to be taken into account when generating test cases. A set of paths satisfies edge coverage, also known as edge path cover [25], iff every edge is visited in at least one path and each path starts in the only source state partition $\emptyset$ and ends in any sink state partition (without outgoing edges).

A refinement to edge coverage is *weak edge coverage*, which does not require paths to end in a sink node. This corresponds to transition coverage described in [26].

6.5 Summary of Test Generation Algorithm

In summary, the test case generation requires a configuration specification c, a path selection strategy $s \in \{path, edge, weakedge\}$ and a valid execution order $\pi \in \Pi_c$, as follows:

1. generate the minimal STG g for c
2. generate paths P using the path selection strategy s on g
3. return test cases $T = \{T_{STG}(p, \pi) \mid p \in P\}$ generated from the set of paths P

To illustrate the test case generation, consider the scenario script presented in Section 2 and its partitioned STG depicted in Figure 1, which is already the minimal STG.

We use weak edge coverage to select paths, which results in the paths and test cases listed in Table 2. Note that execution steps are abbreviated with $e\langle\cdot\rangle$ instead of $exec\langle\cdot\rangle$, and assertion steps with $a\langle\cdot\rangle$ instead of $assert\langle\cdot\rangle$.

#	Path / Test Case	#e	#a
1	$\emptyset \xrightarrow{d} \{d\} \xrightarrow{u} \{d, u\} \xrightarrow{i} \{d, u, i\} \xrightarrow{r} \{d, u, i, r\}$ $e\langle d\rangle \triangleright a\langle d\rangle \triangleright$ $e\langle u\rangle \triangleright a\langle d\rangle \triangleright a\langle u\rangle \triangleright$ $e\langle i\rangle \triangleright a\langle d\rangle \triangleright a\langle u\rangle \triangleright a\langle i\rangle \triangleright$ $e\langle r\rangle \triangleright a\langle d\rangle \triangleright a\langle u\rangle \triangleright a\langle i\rangle \triangleright a\langle r\rangle$	4	10
2	$\emptyset \xrightarrow{d} \{d\} \xrightarrow{u} \{d, u\} \xrightarrow{r} \{d, u, r\} \xrightarrow{i} \{d, u, i, r\}$ $e\langle d\rangle \triangleright a\langle d\rangle \triangleright$ $e\langle u\rangle \triangleright a\langle d\rangle \triangleright a\langle u\rangle \triangleright$ $e\langle r\rangle \triangleright a\langle d\rangle \triangleright a\langle u\rangle \triangleright a\langle r\rangle \triangleright$ $e\langle i\rangle \triangleright a\langle d\rangle \triangleright a\langle u\rangle \triangleright a\langle i\rangle \triangleright a\langle r\rangle$	4	10
		8	20

Table 2. Selected paths and generated test cases

The test goal defined at the beginning of this section defines that certain resource properties need to be tested. The construction of a minimal STG (which in our scenario equals the full STG) ensures that these properties are indeed attested by appropriate assertion steps. Table 3 lists the required properties and test cases in which they are checked.

Res.	Idem.	Pres. of d	Pres. of u	Pres. of i	Pres. of r
d	1, 2	-	-	-	-
u	1, 2	1, 2	-	-	-
i	1, 2	1, 2	1, 2	-	2
r	1, 2	1, 2	1, 2	1	-

Table 3. Test goal properties and attesting test cases

7. Implementation

This section presents the *citac* framework[1], which is an open-source prototype implementation of our approach.

7.1 Architecture

The prototype is implemented as a Ruby application, and currently focuses primarly on testing Puppet [21] manifests. The implementation is modular such that other configuration management tools like Chef or Ansible can be easily integrated. Although their implementations are very different from Puppet, most configuration management tools share the same conceptual model of declarative, idempotent resources and are thus applicable to our prototype.

The system architecture is depicted in Figure 3. The central component is the *test manager*, which manages the database containing the configuration specifications and controls the execution of test cases. The *configuration specification database* (depicted as *DB*) stores the model metadata, generated test suites and execution results. Furthermore, all files and Puppet modules required to run a configuration specification are also stored within the database.

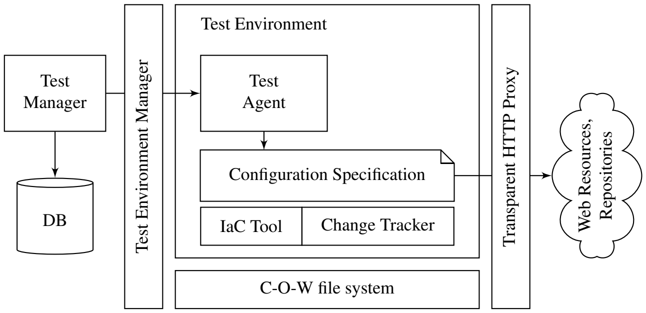

Figure 3. Architecture

The *test environment manager* is a component responsible for providing clean test environments for execution. The prototype builds upon lightweight Linux containers provided by Docker [23]. Docker containers share the kernel with the host operating system and provide an isolated file system, network stack, process space, etc. We use pre-built Docker images with a basic operating system environment. Writing persistent files is performed using an efficient copy-on-write (C-O-W) [27] file system mechanism.

The *test agent* is a component which executes a single test case. Our prototype ships with a patched version of Puppet which overwrites the mechanism responsible for scheduling resource executions. The implementation of the *change tracker* (see Section 7.2) uses *strace*[2] and other tools to monitor system changes performed by certain resource actions.

Executing configuration specifications typically involves downloading software packages from some external repositories. In order to enhance test process performance as well as to reduce workloads on external repositories, a *transparent HTTP proxy* is deployed, which caches downloaded files.

7.2 Change Tracking

A core requirement of our approach is to reliably track any system state changes effected by configuration scripts. In the following, we distinguish between tracking *persistent* state and *transient* state.

7.2.1 Tracking Changes in Persistent State

Persistent state is the part of the system state which survives system power-offs, for instance the contents of the hard disks, in particular the files stored on the file systems, including installed software packages. Moreover, master boot records, partition tables, BIOS / UEFI firmware settings and others count towards persistent state as well.

[1] `https://github.com/citac/citac`

[2] `http://strace.sourceforge.net/`

Running containers in Docker is based on a copy-on-write file system, which we use for tracking persistent state changes. When starting a container from an image, files are served directly from the stored image but file changes occurring in the running container are saved to a different location. Docker can be instructed to create another layer of indirection for a running container by creating an image from that container on the fly.

All file accesses performed by the resource action (and any descendant sub-processes) are monitored by tracing the corresponding system calls with the `ptrace` kernel library and the corresponding command line utility `strace`. After execution the current post-state of all accessed files is compared to the pre-state, which can be obtained by accessing the container snapshot that has been created just before applying the action. `strace` is used in order to distinguish file changes caused by the action from potential out-of-band changes caused by server daemons or other unrelated processes running in the background.

7.2.2 Tracking Changes in Transient State

Transient state on the other hand is the part of the system state which does not survive system power-offs, hence it is lost and reset on each restart.

The implemented prototype monitors the following aspects of transient state. Note that some items can be configured through files, however temporary changes held in memory can be made to the system as well.

- status of network interfaces
- network route configuration
- listening server sockets
- mounted file systems
- running processes

The transient state can be collected efficiently, therefore we capture pre- and post-states and compare them directly.

Depending on the configuration specification under test additional aspects may need to be included as well. The prototype was inspired by the data collected by Ohai [7], which produces a simple JSON file and allows for easy extension by small plugins written in Ruby. However, due to performance reasons, we decided not to build upon Ohai, but to use a custom lightweight implementation instead.

8. Evaluation

We have conducted a set of large-scale experiments to evaluate the proposed solution. The evaluation setup is discussed in Section 8.1, in Section 8.2 we report aggregated numbers and results, and Section 8.3 discusses in detail some selected bugs and issues that we have been able to reveal. The detailed evaluation results are available online[3].

8.1 Test Setup

We have manually selected 101 test modules from Puppet Forge[4] and automatically tested them in isolation for idempotence and convergence. The main selection criteria are number of downloads, number of resources, use of resources with custom shell commands (as those appear to be prone to programming bugs), and ability to run the module in Docker.

In addition, 11 configuration specifications with known bugs have been manually chosen, in order to ensure that our approach actually detects all bugs of different types (5 real world scripts with documented issues, 6 constructed ones). The set of scripts is exhaustive with regards to the issue taxonomy presented below (see Section 8.3, Figure 6).

For each Puppet module, we generated STG based test suites, which we then executed on a cluster with 6 virtual machines, each with 2GB RAM and 40GB disk space. Overall, the net execution of all 250.805 test steps took 9.15 days. Management overhead like transferring scripts and results between test hosts is excluded.

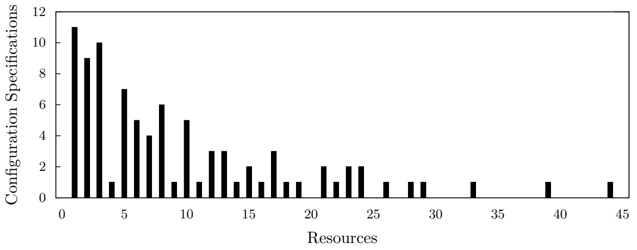

Figure 4. Resource Count Distribution

The analyzed configuration specifications consist of 9.7 resources on average, ranging from 1 to 44 resources. The distribution of resource counts is depicted in Figure 4.

Resource Type	Count	Percentage	Resource Type	Count	Percentage
file	457	53.3 %	file_line	10	1.2 %
package	162	18.9 %	yumrepo	9	1.0 %
exec	107	12.5 %	group	9	1.0 %
service	47	5.5 %	ini_setting	9	1.0 %
augeas	20	2.3 %	apt_key	9	1.0 %
user	11	1.4 %	others	8	0.9 %
				858	100.0 %

Table 4. Resource Types in Tested Puppet Modules

The analyzed Puppet modules consist of 858 resources in total. Table 4 lists in detail the distribution of resource types. The major resource types are `file` (create files with specific content), `package` (download software packages), `exec` (arbitrary shell commands), and `service` (start daemons).

8.2 Aggregated Test Statistics

In this section we briefly discuss some aggregated numbers and test statistics. Table 5 lists the average execution time

[3] `https://citac.github.io/eval/`

[4] `https://forge.puppetlabs.com/`

and standard deviation for all types of execution and assertion steps which have been executed during the evaluation.

Resource	**Execution Steps**			**Assertion Steps**		
Type	count	average	std.dev.	count	average	std.dev.
file	74504	0.8 s	3.0 s	65534	2.4 s	3.8 s
package	32605	10.8 s	51.2 s	30890	2.6 s	4.0 s
exec	15191	1.2 s	2.6 s	12757	4.1 s	12.4 s
apt_key	4037	1.5 s	4.5 s	3353	10.7 s	13.5 s
group	1267	0.2 s	0.3 s	1593	2.5 s	5.3 s
user	1272	0.2 s	0.1 s	1354	2.5 s	4.9 s
augeas	616	0.8 s	0.5 s	743	2.9 s	2.0 s
file_line	780	0.1 s	0.0 s	620	2.2 s	0.9 s
service	980	3.4 s	7.1 s	613	6.5 s	9.8 s
ini_setting	709	0.3 s	0.1 s	428	2.4 s	0.8 s
yumrepo	229	0.2 s	0.2 s	173	1.9 s	5.5 s
others	317	0.4 s	0.2 s	240	6.6 s	3.5 s
	132507	3.4 s	25.9 s	118298	2.9 s	6.1 s

Table 5. Performance of Execution and Assertion Steps

Looking at simple resource types like `file`, `user`, or `group`, the assertion overhead is approximately 1.6-2.3 seconds. This is caused by the change tracking mechanism, which involves intercepting system calls, creating snapshot Docker images, and capturing the transient system state. Please note that all resource types are treated equally without any knowledge about them, which requires that the full change tracking mechanism is applied even to simple ones with clear semantics like `file`.

The execution time of expensive resources like `package` varies greatly, with average around 10.8 seconds, but some outliers causing the standard deviation to be 51.2 seconds.

Surprisingly, the average execution time for assertion steps of type `apt_key` is much higher than the corresponding times for execution steps. An in-depth analysis revealed that this is due to high costs for creating snapshot images in some of the tested configuration specifications caused by large amounts of modified files.

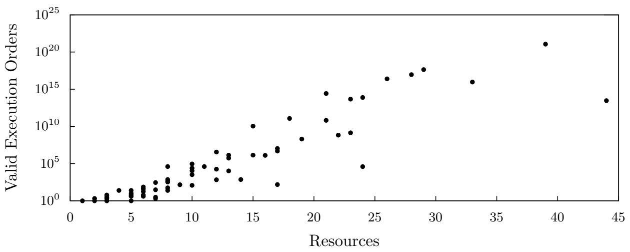

Figure 5. Valid Execution Orders by Resource Counts

A major motivation for creating the test approach presented in this paper is the possibility to overcome the problem of testing infeasibly many execution orders in partially ordered configuration specification such as Puppet manifests. The amount of valid execution orders depends on 1) the number of resources, and 2) on the dependency relation. The more resources are depending on each other, the less valid execution orders there are. Figure 5 correlates the resource count to the amount of valid execution orders (note the logarithmic scale on the y-axis).

Although it appears that there is a roughly exponential relation between resource count and number of valid execution orders, the actual numbers deviate quite significantly in some cases. For instance, the configuration specification with 39 resources has approximately 10^8 times more valid execution orders than the one with 44 resources, due to a lower number of resource dependencies.

8.3 Detected Issues

Our prototype was able to detect all issues in our set of scripts with known issues. In addition to that we discovered 5 hitherto *unknown* issues in our large set of tested scripts.

In the following we elaborate on the potential causes of non-idempotence and non-convergence. Based on our findings, we have derived the (potentially non exhaustive) taxonomy in Figure 6 which summarizes the main types of issues and their potential causes. In the rest of this section, we discuss details on some selected results along the different categories in the taxonomy.

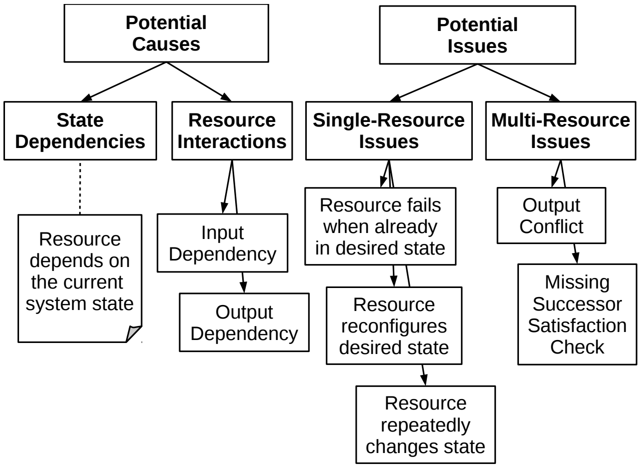
Potential Causes
State Dependencies
Resource Interactions
Resource depends on the current system state
Input Dependency
Output Dependency
Potential Issues
Single-Resource Issues
Multi-Resource Issues
Resource fails when already in desired state
Resource reconfigures desired state
Resource repeatedly changes state
Output Conflict
Missing Successor Satisfaction Check

Figure 6. Taxonomy of Idempotence/Convergence Issues

8.3.1 State and Resource Dependencies

State dependency. A resource has a state dependency and is called *stateful* [11] if its semantics are defined based on the current system state, e.g., if a resource writes the host name to a configuration file. In such cases modifications made by other resources may influence whether the stateful resource is satisfied or not.

Conflicting modifications are detected by our approach because in such cases resource preservation is violated.

Input dependency. An input dependency occurs if a resource requires the output of a previous resource in order to execute. A common pattern is the installation of a package which creates some directories by one resource and the

```
==================================
38. assert(Exec[unzip-downloaded])
==================================

Step result:    failure
Execution time: 4.923841802 seconds

############## OUTPUT START ##############
Archive: /tmp/glassfish-3.1.2.2.zip
replace [...]/catalog? [y]es, [n]o, [A]ll, [N]one:
[...] (EOF or read error, treating as "[N]one" ..)
Error: unzip returned 1 instead of one of [0]
##############  OUTPUT END  ##############
```

Listing 2. Failing assertion step of `fatmcgav/glassfish`

creation of configuration files in these directories by another resource, as documented in a Puppet practitioner's blog [17].

Such dependencies are detected by our approach as falsely classified non-related resources are executed in both orders, such that the dependent resource will fail in one case.

Output dependency. If multiple resources configure the same aspect of a system, an output dependency exists between them. For instance, configuring multiple host entries through multiple resources requires a coordinated access to the hosts file. If they fail to do so, preservation between the resources is violated, which is detected by our test approach.

8.3.2 Single-Resource Issues

Resource fails when already in desired state. State dependent resources may fail if they do not expect the system to be already configured properly, e.g., the `unzip` resource in our scenario fails on re-executions by asking if the already extracted files should be overwritten. We have detected this issue in the two popular Puppet scripts below.

- The execution of module `elasticsearch-logstash` version 0.5.1 fails when trying to remove logstash on subsequent runs once logstash is not present anymore[5]. This idempotence issue is caused by missing init scripts used to gather the service state. After the first execution, the init scripts are deleted, causing the Puppet configuration to fail upon re-execution.
- The module `fatmcgav/glassfish`[6] version 0.6.0 is the most popular Puppet Forge module to configure the GlassFish application server (>12.000 downloads at the time of writing). The module suffers from an issue in which extracting the downloaded binaries fails on subsequent executions. The corresponding resource does not check whether the downloaded binaries were already extracted but instead determines whether its successor is already satisfied. Listing 2 illustrates how our prototype reports this hitherto unknown issue.

  Manual testing would probably not reveal such a bug because it only occurs if the execution is aborted directly after executing the extracting resource and prior to the execution of its successor. This kind of script abortion may occur, e.g., during a power outage or if the disk space on the configured system is exhausted, however manually testing situations like these is not feasible.

Resource reconfigures desired state. A resource may fail to determine that the system is already properly configured and thus reconfigures it. This is problematic for various reasons such as continuous restarts of services which reload themselves once their configuration file is rewritten. In our scenario, the `download` resource continuously downloads the zip archive as long as the installation has not succeeded. The following scripts were found to be affected by this issue.

- The Puppet module `sensu-sensu` is known to rewrite its dashboard configuration file in old versions prior to 3/5/2014 when configured with empty user name and password[7]. The file rewrite triggers a restart of the dashboard service, which causes downtime and thus reduced availability. This issue was correctly detected by our prototype as part of the evaluation.
- We detected a new (unknown) issue in the Puppet module `echocat/kibana4`[8] in version 1.1.1, which installs Kibana, a data visualization platform. The download resource fails to properly check if the package already exists and re-fetches it on each execution.

  Listing 3 shows the prototype's output of the failing assertion step. In contrast to Listing 2 in which the resource execution failed, this listing reports system changes.

```
==================================
2. assert(Exec[Download Kibana4])
==================================

Step result:    failure
Execution time: 4.021931014 seconds

############## OUTPUT START ##############
##############  OUTPUT END  ##############

########## CHANGE SUMMARY START ##########
1 changes:
  file/changed: /opt/kibana-4.0.0-linux-x64
##########  CHANGE SUMMARY END  ##########"
```

Listing 3. Failing assertion step of `echocat/kibana4`

- The Puppet module `dwerder/graphite`[9] in version 5.9.0 installs the Python package `graphite-web` via the Python package manager `pip` for which there exists a native binding in Puppet. The package fails to register itself as installed in the package manager database, which causes a re-download and re-installation on every Puppet

[5] `https://github.com/elastic/puppet-logstash/issues/228`

[6] `https://forge.puppetlabs.com/fatmcgav/glassfish/0.6.0`

[7] `https://github.com/sensu/sensu-puppet/issues/205`

[8] `https://forge.puppetlabs.com/echocat/kibana4/1.1.1`

[9] `https://forge.puppetlabs.com/dwerder/graphite/5.9.0`

run. In addition to the redundant download, all source files are re-compiled upon repeated installation.

Resource repeatedly changes state. A state dependent resource may depend on itself or on a constantly changing system aspect, e.g., if it appends a line to a file but fails to check that the line was already written or if it writes the current time to a file. We were able to construct such issues and detect them via our prototype, however we did not encounter an existing real world script exhibiting this type of issue.

8.3.3 Multi-Resource Issues

Output conflict. Two output dependent resources may fail to agree on a common desired state and thus conflict with each other. In our scenario, the `download` and the `remove` resource cannot agree on whether the zip archive should exist or not. We also correctly detected a known issue [19] with an output conflict of the Chef cookbook `timezone` version 0.0.1. The cookbook sets the timezone by overwriting the file `/etc/timezone`, but then reconfigures the `tzdata` package, which rewrites the file with another content. We translated the cookbook directly into Puppet and were able to detect the convergence issue with our tool.

Missing successor satisfaction check. Many configuration scripts include temporary actions, like downloading and extracting the zip archive in our scenario. If such resources fail to check that the desired state is already reached, they may be in conflict with cleanup actions like the remove resource. We have detected the following instance of this problem.

- The Puppet module `oscerd/java`[10] in version 1.0.2 copies an archive to a temporary directory in order to install Java. The module cleans up the temporary data, however the archive is copied again on each execution because the corresponding resource fails to check that the desired version of Java has already been installed.

9. Conclusion

This paper discusses the difficulty of creating reliable automated system configurations which let the system converge to a desired state by continuous re-executions. Although such tools are based upon declarative descriptions, convergence to the target state requires performing concrete imperative actions on the system. We identified that the key component for achieving a reliable and convergent process is the idempotence of each applied resource action.

Problems arise if resources conflict with each other or depend on each other. If two or more resources manage the same aspect of the system, conflicts occur if they fail to coordinate themselves and do not agree on a shared desired state in which all participating resources are satisfied at the same time. On the other hand, if a resource depends on a certain aspect of the system which is managed by another resource, different execution orders may yield different outcomes.

We propose an automated model based test framework which determines whether a system configuration converges to a stable desired state. We assume configuration specifications whose resources are partially ordered. Testing all possible instantiations is usually infeasible due to the large amount of possible execution orders. To overcome this problem, we introduce the concept of pairwise resource preservation, which makes convergence testing feasible.

We have shown the effectiveness of our approach by means of a comprehensive evaluation of system tests that have generated some exciting and very encouraging insights. The prototype is able to detect all idempotence and convergence issues in a set of real world configuration specifications with known issues, as well as some hitherto undiscovered issues in a reasonably large set of public Puppet scripts.

In future work, we aim to extend and optimize our approach in multiple ways. First, we look into possibilities for static pruning, which may further reduce the testing effort while still achieving an acceptable coverage level. Our current proposal using pairwise tests is a blackbox approach; we plan to optimize this by incorporating knowledge about the specific resources which are executed. For instance, preservation test cases for file resources with non-overlapping file paths may be omitted. A similar idea is discussed in [8].

Second, we aim to extend the system model to support dynamic configuration specifications in which the declared resources change upon system state. This is necessary to cater for new and imperative language constructs, e.g., loops and lambda expressions in Puppet version 4.

Third, we envision that tests can be combined and further optimized via interleaving executions. As outlined in [10], it is possible to construct a sequence of resources with a length in $O(n^2)$ (in relation to the resource count n) in which all $O(n!)$ valid execution orders occur, although they may be interleaved with other resources. It is subject to future work to determine whether interleaved executions can possibly detect further idempotence and convergence issues.

Finally, we aim at analyzing whether supposedly independent configuration scripts could potentially interfere with each other. We anticipate that or approach is capable of detecting such issues because they will likely violate the preservation property.

Acknowledgments

We would like to thank the anonymous reviewers for their detailed and constructive feedback which helped to further improve the quality of the paper.

References

[1] J.-P. Arcangeli, R. Boujbel, and S. Leriche. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software*, 103, 2015.

[2] A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems.* Prentice Hall, 1994.

[10] `https://forge.puppetlabs.com/oscerd/java/1.0.2`

[3] G. Brightwell and P. Winkler. Counting Linear Extensions is #P-complete. In *23rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 175–181, 1991.

[4] R. Bubley and M. Dyer. Faster random generation of linear extensions. *Discrete Mathematics*, 201, 1999.

[5] M. Burgess. CFEngine: a site configuration engine. *Computing Systems*, 8(3), 1995.

[6] M. Burgess and A. Couch. Modeling Next Generation Configuration Management Tools. In *20th Int. Conference on Large Installation System Administration (LISA)*, 2006.

[7] Chef Software, Inc. Ohai. `https://docs.chef.io/ohai.html`, 2015.

[8] J. Collard, N. Gupta, R. Shambaugh, A. Weiss, and A. Guha. On Static Verification of Puppet System Configurations. *CoRR*, 2015.

[9] A. Couch and M. Chiarini. Dynamic Consistency Analysis for Convergent Operators. In *Resilient Networks and Services*. 2008.

[10] A. Couch and N. Daniels. The Maelstrom: Network Service Debugging via "Ineffective Procedures". In *15th USENIX Conference on Large Installation System Administration (LISA)*, pages 63–78, 2001.

[11] A. Couch and Y. Sun. On the Algebraic Structure of Convergence. In *Self-Managing Distributed Systems*, pages 28–40, 2003.

[12] A. Couch and Y. Sun. On observed reproducibility in network configuration management. *Science of Computer Programming*, 2004.

[13] T. Delaet, W. Joosen, and B. Vanbrabant. A Survey of System Configuration Tools. In *24th International Conference on Large Installation System Administration (LISA)*. USENIX Association, 2010.

[14] S. Erdweg, M. Lichter, and M. Weiel. A sound and optimal incremental build system with dynamic dependencies. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 89–106, 2015.

[15] F. Erich, C. Amrit, and M. Daneva. A Mapping Study on Cooperation between Information System Development and Operations. In *Product-Focused Software Process Improvement*. 2014.

[16] A. Gambi, W. Hummer, H.-L. Truong, and S. Dustdar. Testing Elastic Computing Systems. *IEEE Internet Computing*, 17(6):76–82, 2013.

[17] R. Harrison. How to Avoid Puppet Dependency Nightmares With Defines. `https://blog.openshift.com/how-to-avoid-puppet-dependency-nightmares-with-defines`, retrieved on 12/15/2015, 2013.

[18] L. Hochstein. *Ansible: Up and Running*. O'Reilly Media, Inc., 2014.

[19] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam. Testing Idempotence for Infrastructure as Code. In *14th ACM/IFIP/USENIX International Middleware Conference*. 2013.

[20] M. Hüttermann. *DevOps for developers*. Apress, 2012.

[21] S. Krum, W. Hevelingen, B. Kero, J. Turnbull, and J. McCune. *Pro Puppet*. Apress, 2013.

[22] J. Loope. *Managing Infrastructure with Puppet*. O'Reilly Media, Inc., 2011.

[23] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239), Mar. 2014.

[24] M. Miglierina. Application Deployment and Management in the Cloud. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2014.

[25] S. Ntafos and S. Hakimi. On Path Cover Problems in Digraphs and Applications to Program Testing. *IEEE Transactions on Software Engineering*, SE-5(5):520–529, 1979.

[26] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[27] H. Powell. ZFS and Btrfs: A Quick Introduction to Modern Filesystems. *Linux J.*, 2012(218), June 2012.

[28] V. Sobeslav and A. Komarek. OpenSource Automation in Cloud Computing. In *4th International Conference on Computer Engineering and Networks*, pages 805–812. 2015.

[29] D. Spinellis. Don't Install Software by Hand. *IEEE Software*, 2012.

[30] M. Taylor and S. Vargo. *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly Media, 2014.

[31] J. Tretmans. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing*, pages 1–38. Springer, 2008.

[32] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2), 1979.

[33] F. van Ham, H. van de Wetering, and J. van Wijk. Interactive visualization of state transition systems. *IEEE Transactions on Visualization and Computer Graphics*, 8(4):319–329, 2002.

[34] J. Wettinger, U. Breitenbücher, and F. Leymann. Compensation-Based vs. Convergent Deployment Automation for Services Operated in the Cloud. In *12th International Conference on Service-Oriented Computing (ICSOC)*, pages 336–350, 2014.

A. Appendix

A.1 Proof: Associativity of Action Composition

Proof that action composition is associative.. We need to show for arbitrary actions a, b and c and any state $s_0 \in \mathcal{S}$ that $((a \triangleright b) \triangleright c)(s_0) = (a \triangleright (b \triangleright c))(s_0)$ holds.

- Assume $a(s_0) = (s_1, \bot)$. It follows by the definition of $\triangleright$ that $(a \triangleright b)(s_0) = a(s_0) = (s_1, \bot)$ and $((a \triangleright b) \triangleright c)(s_0) = (a \triangleright b)(s_0) = (s_1, \bot)$. Further $(a \triangleright (b \triangleright c))(s_0) = a(s_0) = (s_1, \bot)$ holds, thus $((a \triangleright b) \triangleright c)(s_0) = (s_1, \bot) = (a \triangleright (b \triangleright c))(s_0)$.
- Assume $a(s_0) = (s_1, \top)$.

- Assume $b(s_1) = (s_2, \bot)$. It follows by the definition of $\triangleright$ that $(a \triangleright b)(s_0) = b(s_1) = (s_2, \bot)$ and $((a \triangleright b) \triangleright c)(s_0) = (a \triangleright b)(s_0) = b(s_1) = (s_2, \bot)$. Further $(b \triangleright c)(s_1) = b(s_1) = (s_2, \bot)$ and $(a \triangleright (b \triangleright c))(s_0) = (b \triangleright c)(s_1) = b(s_1) = (s_2, \bot)$ holds, thus $((a \triangleright b) \triangleright c)(s_0) = (s_2, \bot) = (a \triangleright (b \triangleright c))(s_0)$.
- Assume $b(s_1) = (s_2, \top)$. It follows by the definition of $\triangleright$ that $(a \triangleright b)(s_0) = b(s_1) = (s_2, \top)$ and $((a \triangleright b) \triangleright c)(s_0) = c(s_2)$. Further $(b \triangleright c)(s_1) = c(s_2)$ and $(a \triangleright (b \triangleright c))(s_0) = (b \triangleright c)(s_1) = c(s_2)$ holds, thus $((a \triangleright b) \triangleright c)(s_0) = c(s_2) = (a \triangleright (b \triangleright c))(s_0)$.

□

A.2 Theorem: Idempotence does not imply Preservation

Theorem 4. There are idempotent configuration specifications whose resources do not preserve their respective ancestors and non-related resources.

Proof. Let states be represented as single natural numbers ($\mathcal{S} = \mathbb{N}$). Consider the configuration specification $c = (\{r_1, r_2\}, \{(r_1, r_2)\})$ with $r_i(s) = (i, \top)$.

Observe that c is idempotent. The only valid execution order is $\pi(r_i) = i$. $\overline{c_\pi}(s) = (2, \top)$ holds for any state $s \in \mathcal{S}$ because $\overline{r_1}(s) = (1, \top)$, $\overline{r_2}(1) = (2, \top)$ and thus $\overline{c_\pi}(s) = (\overline{r_1} \triangleright \overline{r_2})(s) = (2, \top)$. It follows that $(\overline{c_\pi} \triangleright \overline{c_\pi})(s) = (2, \top) = \overline{c_\pi}(s)$ holds for any state s, thus $\overline{c_\pi} = \overline{c_\pi} \triangleright \overline{c_\pi}$.

Although r_1 is an ancestor of r_2, r_2 does not preserve r_1. Consider the state $s_1 = 1$ which obviously satisfies r_1 ($\overline{r_1}(1) = (1, \top)$). Applying r_2 to s_1 results in state $s_2 = 2$ because $\overline{r_2}(s_1) = (2, \top)$. s_2 does not satisfy r_1 anymore because $\overline{r_1}(s_2) = (1, \top) \neq (2, \top)$.

It follows that c is an idempotent configuration specification whose resource r_2 does not preserve its ancestor r_1. □

A.3 Theorem: Convergence does not imply Preservation

Theorem 5. There are convergent configuration specifications whose resources do not preserve their respective ancestors and non-related resources.

Proof. Consider the following system and configuration specification.

$$
\begin{aligned}
\mathcal{S} &= \mathbb{Z} \\
R_c &= \{r_1, r_2, r_3\} \\
\prec_c &= \{(r_1, r_2), (r_2, r_3), (r_1, r_3)\} \\
\overline{r_1}(s) &= (1, \top) \\
\overline{r_2}(s) &= \begin{cases} (1, \top) & \text{if } s > 0 \\ (0, \top) & \text{if } s \leq 0 \end{cases} \\
\overline{r_3}(s) &= \begin{cases} (1, \top) & \text{if } s > 0 \\ (-1, \top) & \text{if } s \leq 0 \end{cases}
\end{aligned}
$$

Observe that the only valid execution order π is $\pi(r_i) = i$, thus $\overline{c_\pi} = \overline{r_1} \triangleright \overline{r_2} \triangleright \overline{r_3}$. After applying r_1 the state is always 1 because $\overline{r_1}(1) = \overline{r_2}(1) = \overline{r_3}(1) = (1, \top)$, hence c is convergent.

However, r_3 does not preserve r_2. State 0 satisfies r_2 ($\overline{r_2}(0) = (0, \top)$), thus after applying r_3, r_2 must still be satisfied. Observe that the output state -1 ($\overline{r_3}(0) = (-1, \top)$) does not satisfy r_2 anymore ($\overline{r_2}(-1) = (0, \top) \neq (-1, \top)$), hence r_3 does not preserve r_2 although c is convergent. □