Rehearsal: A Configuration Verification Tool for Puppet

Michael Jank[a]

a Technical University of Vienna, Austria

Abstract Although configuration management tools like Puppet make configuring a lot of systems easier, they don't protect from misconfiguration. In some cases bugs are not caught in the testing environment because the configuration leads to an indeterministic state.

Puppet manifests are configuration files which represent the desired state of a system. Among other things, it contains information about which users to create, files and their desired content and packages that should be installed. Manifests can sometimes seem trivial and correct, but still violate the properties of determinism and idempotence.

Shambaugh et al. [10] tackle this problem with *Rehearsal*, a configuration verification tool for Puppet. The tool applies static verification to check whether Puppet manifests are deterministic and idempotent. For this purpose they create a semantics for Puppet and a domain specific language, namely FS, into which they transform the configuration files. A summary of their approach is presented in this paper.

Keywords configuration management, static verification, Puppet, determinism

The Art, Science, and Engineering of Programming

Perspective The Engineering of Programming

Area of Submission Configuration Management, Verification

1 Introduction

System administrators, that have to take care of more than only a handful of servers, often resort to some kind of tooling to help with the management of them. A few years ago this often resulted in them hacking together some scripts that would take care of that. Given decent programming abilities that could lead to successful management of big infrastructures. Still, without a framework to work in, every script would look and work a little bit different, depending on the background of the person that created it.

In recent years the growing need for management of big cloud infrastructures brought forward new tools and with them new domain specific languages (DSL) that help with exactly that. Puppet [7] and Ansible [4] are two examples for configuration languages that are widely used in the industry today. These tools, albeit being a great improvement, still don't protect from misconfiguration. It repeatedly happens that such errors are the cause for major service outages [6, 9, 12, 15]. In a study about bugs occurring in cloud systems by *Gunawi et al.* [2], configuration issues were the 4th biggest category they found.

One reason why this keeps happening is because bugs of the configuration are often times not caught in the development or testing environment. In other words: the configuration causes different behavior there, than in the production environment. To find this class of bugs in Puppet configurations, *Shambaugh et al.* [10] presents *Rehearsal*, a tool to verify configurations. The findings of that work are summarized in this paper. For this purpose at first Puppet is introduced, followed by sections about the semantics and analyses presented by *Rehearsal*. Lastly relevant related work and a conclusion is presented.

2 Puppet

Puppet is an open source project that provides tools to automate the management of infrastructure.

2.1 Architecture [8]

Puppet follows a *master/agent architecture* where the master node is in control of the configuration for several agent nodes. Along with that architecture the following communication pattern is used:

1. An agent node sends facts to the master and requests a catalog.
2. The master node responds with a catalog for the requesting node to specify how it should be configured, compiled using the facts from the agent node.
3. The agent node makes sure that the resources described in the catalog are in their desired state. To achieve this it applies the necessary changes.
4. The agent node then responds to the master with a report about those changes.

2.2 Manifests

A catalog consists, among other things, of so called *manifests*. A manifest is a collection of resources, their desired state and their dependencies.

In Listing 1 an example of a puppet manifest is given. The resource types in this case are *package*, *file* and *user*, which configure the respective instances *vim*, */home/carol/.vimrc* and *carol*. It shall be ensured that the package *vim* is installed, a file */home/carol/.vimrc* with the content syntax on exists and a user *carol* is present.

Listing 1 Puppet manifest [10]

```
package{'vim':
        ensure => present
}
file{'/home/carol/.vimrc':
        content => 'syntax on'
}
user{'carol':
        ensure => present,
        managehome => true
}
```

You may already have observed, that those three steps require to be executed in a certain order. Specifically the user account has to exist before a file can be created in it. For this purpose the property `require => User['carol']` has to be added for the file resource.

2.3 Problems using Puppet

Problems that commonly occur in Puppet manifests, as listed by *Shambaugh et al.* [10], are non-deterministic errors, over-constrained dependencies, silent failures and non-idempotence.

2.3.1 Non-Deterministic Errors

A regular task for Puppet is to install a package and then configure it. For this purpose, configuration files have to be created or overwritten after the package was installed. In Listing 2 there is no dependency between the creation of the configuration file and the installing of the package. Therefore it could happen, that the configuration file is created and then overwritten by the installation of the package. Determinism requires a Puppet manifest to always result in the same outcome.

Listing 2 Problematic Puppet manifest - non-deterministic error [10]

```
file {"/etc/apache2/sites-available/000-default.conf": content => ...}
package{"apache2": ensure => present}
```

2.3.2 Over-Constrained Dependencies

For the sake of determinism it sometimes happens that a strict order is enforced by introducing dependencies between packages that are actually independent from each other. In Listing 3 both modules *cpp* and *ocaml* require the packages *m4* and *make* to be present. The problem here is, that *cpp* requires *m4* to be installed before *make* while *ocaml* has it defined the other way around. Now if both modules are included in a manifest, Puppet will fail to apply the manifest and report a dependency cycle.

Listing 3 Problematic Puppet manifest - conflicting dependency declaration [10]

```
define cpp() {
        package{'m4': ensure => present }
        package{'make': ensure => present }
        package{'gcc': ensure => present }
        Package['m4'] -> Package['make']
        Package['make'] -> Package['gcc']
}
define ocaml() {
        package{'make': ensure => present }
        package{'m4': ensure => present }
        package{'ocaml': ensure => present }
        Package['make'] -> Package['m4']
        Package['m4'] -> Package['ocaml']
}
```

2.3.3 Silent Failure

In Listing 4 there's nothing wrong on first sight, but with the background knowledge that the *go* package in Ubuntu 14.04 depends on Perl this manifest can actually lead to two different states. The first possible state is that *Perl* is removed before *go* is installed (which causes the reinstallation of *Perl*) and the second possible state is *go* being installed first followed by the removal of *Perl*. The removal of *Pearl* in turn causes the removal of *go*. So either both packages are installed or both are being removed.

Listing 4 Problematic Puppet manifest - different success states [10]

```
package{'golang-go': ensure => present }
package{'perl': ensure => absent }
```

2.3.4 Non-Idempotence

Idempotence in the case of Puppet means, that it should be possible to apply a manifest various times without it changing the final outcome. Listing 5 constructs a manifest that is not idempotent, as the directory */src* is deleted after one run. A second run would therefore fail.

Listing 5 Problematic Puppet manifest - not idempotent [10]

```
file{"/dst": source => "/src" }
```

```
file{"/src": ensure => absent }
File["/dst"] -> File["/src"]
```

3 Semantics of Puppet

For *Shambaugh et al.* to perform analyses on manifests they presented semantics for Puppet. This was done in two stages:

1. Creation of a *resource graph*: Manifests are compiled to a directed acyclic graph of primitive resources. Primitive resources are the most basic types of resources available in Puppet. For example we have seen so far *file*, *package* and *user*. A non-primitive resource e.g. would be *cpp* in Listing 3, which in turn will be resolved to three *package* resources.
2. Modeling in *FS*: The primitive resources are then modeled as programs in *FS*, a newly defined imperative language of file system operations. *FS* is designed such that the semantics of resources can be expressed and at the same time static analysis can be applied.

The resource graph is then compiled into a sequence of *FS* programs. Specifically all possible permutations are generated that adhere to the order given by the edges of the graph.

3.1 FS Programs

The FS language is a simple imperative language that manipulate the file system. File systems are modeled as maps of paths to file contents. Each expression in FS consumes a file system and produces either a new one or aborts with an error. The language was deliberately kept simple to enable the static analyses explained in the next section. The exact restrictions and their implications are explained in [10].

4 Verifications

4.1 Determinism

The main contribution of *Rehearsal* is the introduction of an approach to check whether a resource graph is deterministic. The presented approach has the following steps:

1. **Reduce the number of paths:** As even small manifests can create a large number of paths (in the sense of file system paths), it is sought that the number of paths is reduced for the further analysis. This is done by removing *FS* operations from the analysis that operate on paths that are not accessed by any other operation. This reduction does not affect the result of the determinism-check.
2. **Reduce the number of permutations of the resource graph:** A permutation of the graph is eliminated, if the resources that changed places in two permutations

pass a commutativity check, i.e. the programs end up in the same state independent of whether the change of the resources happened or not. It is to note that such a check may include more than two resources, depending on the constellation.

3. **Encoding the semantics as a decidable formula:** The manifest is encoded as a decidable formula suitable for an SMT[1] solver. It is satisfiable if and only if the underlying program is non-deterministic.

4.2 Idempotence

The idempotence check can be done easily, after it is established that a resource graph is deterministic. It is now not necessary anymore to check each permutation, but a resource graph can then simply be treated as an expression e. This is because the property of determinism implies that each permutation leads to the same result. Idempotence can now simply be checked by verifying the equivalence between e and $e; e$ (i.e. the result of executing the expression one time is the same as executing it twice).

5 Related Work

5.1 Convergence in Configuration Management

In [3] *Hanappi et al.* introduce a framework for asserting reliable convergence in configuration management. They use state transition graphs to test if a system converges to the state desired by a script. To guarantee convergence they concluded that the idempotence of a script is key. The two approaches therefore tackle more or less the same problem. The main difference in the approaches is, that *Rehearsal* use static verification whereas *Hanappi et al.* focus on testing. They acknowledge that static verification in general is superior to testing but at the same time argue that static verification is more constrained, as every resource needs clearly defined semantics. Their implementation is also based on Puppet.

5.2 µPuppet

µPuppet [1] complements Rehearsal, as the authors say, on the understanding of the compilation stage of catalogs. They also formalize a subset of Puppet and its semantics. By doing so they implemented a reference implementation of *µPuppet*. This surfaced particularities in how Puppet is handling classes and scope.

[1] Satisfiability modulo theories

5.3 System Configuration Repair

Another recurring task of system administrators is to fix bugs on live systems and then reincorporate the changes into the Puppet manifest. Tortoise [14] presents an attempt to help with that. In their approach the original manifest is used along with a set of shell commands that were used to fix the bugs in order to generate a selection of patches. For this purpose soft constraints are derived from the manifest fails and hard constraints from the shell commands. The generated patch then proposes fixes that are consistent with the fix made through the shell.

5.4 Configuration languages

Conf Valley [5] is a generic framework to validate that a given configuration obeys a given configuration specification. To achieve the same checks about determinism and idempotence as *Rehearsal*, one would have to write a configuration specification that validates the generated resource graph.

The actual application area of *Conf Valley* is rather in the validation of application's configurations, checking whether a concrete specification conforms with the software vendor's intended way.

5.5 Assessing the quality of Puppet configurations

In [11] *Sharma et al.* analyzed more than a 100,000 Puppet files for known code smells. They intend to assess the quality of the code and make it easier to adopt best practices. Similarly *Van der Bent et al.* [13] defined a notion of code quality for puppet, developed a measurement model and implemented a software analysis tool. They verified the results of the tool with Puppet experts.

6 Conclusion

Puppet is a popular configuration framework, that is making it easy to configure a high number of systems at once. This also increases the potential to misconfigure a high number of systems at once. It is therefore highly desirable to have tools that point out possible configuration mistakes as early as possible. The violation of determinism and idempotence can lead to high frustration, as the consequences might only surface occasionally or under certain circumstances. So as long as these two properties are intact, it is more likely to already iron out bugs in the testing environment.

The realization of *Rehearsal* shows great potential. For the selected subset of Puppet it was possible to create a semantics that can check determinism and idempotence. To be useful in practice there is still a lot of work to do. Missing primitive resources would have to be modeled in *FS*. Another aspect that is not covered yet is software that is not installed fully by the operating system's package manager but additionally creates files through install scripts. Rehearsal needs to know about each single file that is created, modified or deleted by an entry in a manifest. Albeit it has to be

acknowledged that *Rehearsal* can even be used in such scenarios. It just wouldn't be a complete check.

Apart from that, the detection of any misconfiguration of applications or similar are not in the scope of *Rehearsal*. Promising work in that direction has been pointed out in Section 5.

References

[1] Weili Fu, Roly Perera, Paul Anderson, and James Cheney. "μPuppet: A Declarative Subset of the Puppet Configuration Language". In: *31st European Conference on Object-Oriented Programming*. 2017.

[2] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. "What bugs live in the cloud? a study of 3000+ issues in cloud systems". In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pages 1–14.

[3] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. "Asserting reliable convergence for configuration management scripts". In: *ACM SIGPLAN Notices* 51.10 (2016), pages 328–343.

[4] Red Hat. *Ansible*. URL: https://docs.ansible.com/.

[5] Peng Huang, William J Bolosky, Abhishek Singh, and Yuanyuan Zhou. "ConfValley: a systematic configuration validation framework for cloud services". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, page 19.

[6] David Oppenheimer, Archana Ganapathi, and David A Patterson. "Why do Internet services fail, and what can be done about it?" In: *USENIX symposium on internet technologies and systems*. Volume 67. Seattle, WA. 2003.

[7] *Puppet*. URL: https://puppet.com/docs/puppet/.

[8] *Puppet Documentation: Architecture Overview*. URL: https://puppet.com/docs/puppet/6.0/architecture.html.

[9] Ariel Rabkin and Randy Howard Katz. "How hadoop clusters break". In: *IEEE software* 30.4 (2013), pages 88–94.

[10] Rian Shambaugh, Aaron Weiss, and Arjun Guha. "Rehearsal: a configuration verification tool for puppet". In: *ACM SIGPLAN Notices*. Volume 51. 6. ACM. 2016, pages 416–430.

[11] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. "Does your configuration code smell?" In: *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE. 2016, pages 189–200.

[12] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. "Holistic configuration management at Facebook". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pages 328–343.

[13] Eduard Van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. "How good is your puppet? an empirically defined and validated quality model for puppet". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pages 164–174.

[14] Aaron Weiss, Arjun Guha, and Yuriy Brun. "Tortoise: interactive system configuration repair". In: *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE. 2017, pages 625–636.

[15] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. "An empirical study on configuration errors in commercial and open source systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pages 159–172.

About the author

Michael Jank is the author of this paper.
Contact him at michael.jank@student.tuwien.ac.at.