Rehearsal: A Configuration Verification Tool for Puppet

Rian Shambaugh Aaron Weiss Arjun Guha
University of Massachusetts Amherst, United States
{rian,aaronweiss,arjun}@cs.umass.edu

Abstract

Large-scale data centers and cloud computing have turned system configuration into a challenging problem. Several widely-publicized outages have been blamed not on software bugs, but on configuration bugs. To cope, thousands of organizations use system configuration languages to manage their computing infrastructure. Of these, Puppet is the most widely used with thousands of paying customers and many more open-source users. The heart of Puppet is a domain-specific language that describes the state of a system. Puppet already performs some basic static checks, but they only prevent a narrow range of errors. Furthermore, testing is ineffective because many errors are only triggered under specific machine states that are difficult to predict and reproduce. With several examples, we show that a key problem with Puppet is that configurations can be non-deterministic.

This paper presents Rehearsal, a verification tool for Puppet configurations. Rehearsal implements a sound, complete, and scalable determinacy analysis for Puppet. To develop it, we (1) present a formal semantics for Puppet, (2) use several analyses to shrink our models to a tractable size, and (3) frame determinism-checking as decidable formulas for an SMT solver. Rehearsal then leverages the determinacy analysis to check other important properties, such as idempotency. Finally, we apply Rehearsal to several real-world Puppet configurations.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords Puppet, system configuration, domain-specific languages, verification.

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908083

1. Introduction

Consider the role of a system administrator at any organization, from a large company to a small computer science department. Their job is to maintain computing infrastructure for everyone else. When a new software system, such as a Web service, needs to be deployed, it is their job to provision new servers, configure the firewall, and ensure that data is automatically backed up. If the Web service receives a sudden spike in traffic, they must quickly deploy additional machines to handle the load. When a security vulnerability is disclosed, they must patch and restart machines if necessary. All these tasks require the administrator to write and maintain system configurations.

Not too long ago, it was feasible to manage systems by directly running installers, editing configuration files, etc. A skilled administrator could even write shell scripts to automate some of these tasks. However, the scale of modern data centers and cloud computing environments has made these old approaches brittle and ineffective.

System configuration languages. System configuration is a problem that naturally lends itself to domain-specific languages (DSLs). In fact, the programming languages community has developed several DSLs for specifying system configurations that are used in practice. For example, NixOS [12] uses a lazy, functional language to describe packages and system configurations; Augeas [5] uses lenses [7] to update configuration files; and Engage [13] provides a declarative DSL that tackles issues such as inter-machine dependencies.

In the past few years, several system configuration languages have also been developed in industry. Puppet, Chef, and Ansible (recently acquired by Red Hat) are three prominent examples. This paper focuses on Puppet, which is the most popular of these languages, but these commercial languages have several features in common that set them apart from prior research. First, they support a variety of operating systems, tools, and techniques that systems administrators already know. Unlike NixOS, they don't posit new package managers or new Linux distributions, but simply use tools like `apt` and `rpm` under the hood. Second, these languages provide abstractions for managing several kinds of resources, such as packages, configuration files, user ac-

counts, and more. Therefore, they are broader in scope than Augeas, which only edits configuration files. Finally, these languages provide relatively low-level abstractions, compared to earlier work like LCFG [1]. For example, Puppet provides a simple and expressive DSLs that encourages average users to build their own abstractions.

Puppet. Puppet configurations (called *manifests*) are written in a expressive, yet constrained DSL, which makes them amenable to analysis. To a first approximation, a manifest specifies a collection of resources, their desired state, and their inter-dependencies. For example, the following Puppet manifest states that the `vim` package should be installed, that the user account `carol` should exist, and that she should have a `.vimrc` file in her home directory containing the single line `syntax on`:

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

It's tedious to describe every individual resource in this manner, so Puppet makes it easy to write parameterized modules. The official module repository, Puppet Forge, has nearly four thousand modules from over six hundred contributors.

Non-determinism and modularity. A key property of Puppet is that manifests should be deterministic [25]. Determinism is a critical property because it helps ensure that a manifest has the same effect in testing and in production. Similarly, if one manifest is applied to several machines, which is common in large deployments, determinism helps to ensure that they are replicas of each other.

Unfortunately, it is easy to write manifests that are not deterministic. Puppet can install resources in any order, unless the manifest explicitly states inter-resource dependencies.[1] Therefore, the example manifest above is non-deterministic: there will be a runtime error if Puppet tries to create the file `/home/carol/.vimrc` before Carol's account. We can fix this bug by making the dependency explicit:

```
User['carol'] -> File['/home/carol/.vimrc']
```

The fundamental problem is that Puppet manifests specify a partial-order on resources, thus resources can be installed in several orders. However, when some dependencies are missing, applying the manifest can go wrong: the system may signal an error or may even fail silently by transitioning to an unexpected state. These bugs are very hard to detect with testing, since the number of valid permutations of resources becomes intractable very quickly.

Surprisingly, a manifest can also have too many dependencies and be over-constrained. Imagine two manifests A and B that both install the resources R_1 and R_2. Suppose that R_1 and R_2 do not depend on each other, but the manifest authors take a conservative approach and add a false dependency to avoid non-determinism issues. If A picks R_1 `->` R_2 and B picks R_2 `->` R_1 then A and B cannot be composed.

Therefore, manifests must be deterministic to be correct, but must only have essential dependencies to be composable. Without composability, manifests cannot be decomposed into reusable modules, which is one of the key features of Puppet. However, when a manifest is only partially-ordered, we may need to check an intractably large number of orderings to verify determinism.

A further complication is that the Puppet has a diverse collection of resource types, which makes it hard to determine how resources interact with each other. For example, a file may overwrite another file created by a package, a user account may need the `/home` directory to be present, a running service may need a package to be installed, and so on. We could try to side-step this issue by building a dynamic determinacy analysis [8, 24]. However, a purely dynamic approach could only identify a problems when two replicas diverge, whereas a static determinacy analysis helps ensure that a manifest behaves correctly on any machine regardless of its initial state.

Idempotency. Determinism is not a sufficient condition to ensure that Puppet behaves predictably. In a typical deployment, the Puppet background process periodically reapplies the manifest to ensure that the machine state is consistent with it. For example, if a user modifies the machine (*e.g.*, manually editing configurations), re-applying the manifest will correct the discrepancy. Thus if the machine state has not changed, reapplying the manifest should have no effect. Like determinacy, this form of idempotence is also believed to be a key property of Puppet [20]. However, it is also trivial to construct manifests that are not idempotent.

Our approach. To the best of our knowledge, this is the first paper to develop programming language techniques for Puppet (or a related language such as Chef and Ansible). We first present a core fragment of Puppet with several small examples that illustrate its problems (section 2). We develop a formal semantics of Puppet that models manifests as programs in a simple, non-deterministic imperative language of filesystem operations called FS (section 3).

Our main technical result is a sound, complete, and scalable determinacy analysis (section 4). To scale to real-world examples, we use three different analyses to shrink the size of models. The first two analyses dramatically reduce the number of paths that the determinism-checker needs to reason about by eliminating resources that do not affect determinism and eliminating other side-effects that are not observed by the rest of the program. The third analysis is an unusual commutativity check that accounts for the fact that resources are mostly idempotent. Finally after leveraging the aforementioned analyses, the determinacy checker encodes the semantics as effectively-propositional formulas for an SMT solver.

[1] Puppet calculates dependencies automatically only in some trivial cases, *e.g.*, files "auto-require" their parent directory.

Types	*rtype* ::=	`file` | `package` | ⋯	
Strings	*str* ::=	"⋯`$x`⋯`$y`⋯"	
Identifiers	x ::=	`$x` \| `$y` | ⋯	
Titles	t ::=	*str* | x	
Values	v ::=	*str*	String
		n	Number
		`[`$v_1 \cdots v_n$`]`	Array
		x	Variable
Attributes	attr ::=	*str* `=>` v	
Resources	R ::=	*rtype*`{`t`:` *attr*$_1 \cdots$ *attr*$_n$`}`	
Manifests	m ::=	R	Resource
		`define` *rtype*`(`$x_1 \cdots x_n$`)` `{` m `}`	Type
		rtype$_1$`[`t_1`]` `->` *rtype*$_2$`[`t_2`]`	Dependency
		m_1 m_2	Composition

Figure 1: Syntax of Puppet fragment used in this paper.

```
define myuser($title) {
  user {"$title":
    ensure => present,
    managehome => true
  }
  file {"/home/${title}/.vimrc":
    content => "syntax on"
  }
  User["$title"] -> File["/home/${title}/.vimrc"]
}

myuser {"alice": }
myuser {"carol": }
```

Figure 2: A user-defined resource type and its instantiations.

We argue that our determinacy analysis enables several other higher-level properties to be checked (section 5), and show this is the case by developing a simple idempotence checker that leverages determinism in a fundamental way.

We implement our algorithms in a tool called Rehearsal, which we evaluate on several real-world examples (section 6). Finally, we discuss related work (section 7), summarize the limitations of our approach (section 8), and conclude (section 9). The Rehearsal source code, benchmarking scripts, and a technical appendix are available online [15].

2. Introduction to Puppet

This section introduces the fragment of Puppet that we use in the exposition of this paper. We also illustrate the kinds of problems that Rehearsal solves.

2.1 A Core Fragment of Puppet

The Puppet DSL is quite sophisticated. It has typical features such as functions, loops, and conditionals, and several domain-specific features that make it easy to specify resources and their relationships. Rehearsal can parse and process a significant subset of Puppet, but, for clarity, we constrain our examples to the fragment of Puppet shown in figure 1. A manifest, m, is composed of resources, resource type declarations, and inter-resource dependencies. A resource, R, has a type, a title, and a map of attributes. The resource type determines how the attribute-map is interpreted. For example, a `file` resource must have an attribute called `path`, a `user` resource must have an attribute called `name`, and so on. The resource title can be any descriptive string, but is often used as the default value for an essential attribute. For example, if a `file` resource does not have the required `path` attribute, the title is used as the `path`. A manifest can declare several resources by juxtaposition, but the order in which resources appear is not significant. Instead, manifests must specify dependencies explicitly. To state that the resource t_2 depends on the resource t_1, we write *rtype*$_1$`[`t_1`]` `->` *rtype*$_2$`[`t_2`]`.[2] In addition to a few dozen built-in resource types, Puppet allows manifests to define their own types. A type definition is essentially a function that consumes named attributes as arguments and produces a manifest as a result. For example, if all users in an organization use the same default environment, we can create a new type called `myuser` and instantiate it for several users, as shown in figure 2.

2.2 Common Puppet Problems

There are a number of problems that can easily occur in Puppet manifests.

Non-deterministic errors. A common Puppet idiom is to first install a package and then overwrite its default configuration. For example, the `apache2` package installs a web server and several configuration files. To host a website, at least the default site configuration file, `000-default.conf`, has to be replaced (figure 3a). If the dependency between the package and the file is accidentally omitted, Puppet may try to create the configuration file first which would signal an error because the file is in a directory that the package has yet to create.

Over-constrained dependencies. Consider a strawman solution to the non-determinism problem: we could add false dependencies so that all resources are totally ordered. Unfortunately, this approach makes it difficult to write independent modules which is one of the main features of Puppet. For example, figure 3b shows two simple types that configure C++ and OCaml development environments.[3] Both modules install `make` and `m4` because they are commonly used by C++ and OCaml projects. To force determinism, both modules in the figure have false dependencies between `make` and `m4`. However, each has picked a different order which can easily occur when the modules have different authors. Therefore, if we try to instantiate both modules simultaneously, Puppet will fail and report a dependency cycle. [4] This heavy-handed approach to determinism sacrifices composability.

[2] The first letter of a type name is capitalized in resource references.

[3] Idiomatic Puppet would use the `class` keyword.

[4] Readers familiar with Puppet may know that shared resources have to be guarded with `defined`. Some people consider `defined` to be an anti-pattern, but a simple search shows that it is used in over 1/3rd of the packages on Puppet Forge to enable the kind of modularity that we discuss.

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

package{"apache2": ensure => present }
```

(a) Signals an error nondeterministically.

```
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']
}
```

(b) Cannot be composed due to false dependencies.

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

(c) Leads to two different success states.

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

(d) Not idempotent.

Figure 3: Several problematic manifests.

Silent failure. In addition to non-deterministic errors, it is also possible to write a manifest that nondeterministically leads to two distinct states without Puppet reporting an error. For example, the manifest in figure 3c states that Perl should be removed and the Go compiler should be installed. Surprisingly, on Ubuntu 14.04, the Go compiler depends on Perl[30], so this state is not realizable. Puppet cannot detect this problem, but simply dispatches to the native package manager (*e.g.*, `apt` or `yum`) to actually install and remove packages. For this manifest, Puppet issues two low-level commands to remove Perl and install Go. Since there are no dependencies, they may execute in either order. If Perl is first removed, the command to install Go installs Perl too, but if Perl is removed after Go is installed, that command will remove Go too. This kind of error is more insidious than a nondeterministic error, since there isn't an obvious fix.

Non-idempotence. Another key property of Puppet manifests is that they should be idempotent: applying a manifest twice should be the same as applying it once. However, Puppet does not enforce this property, which makes it easy to produce manifests that are not idempotent. For example, we can make the non-deterministic manifest in figure 3c deterministic by removing Perl before Go is installed:

```
Package['perl'] -> Package['golang-go']
```

However, this manifest is not idempotent. Puppet checks which packages are installed before it issues any commands to install or remove packages. In this example, if both packages are already installed, Puppet will remove Perl and take no further action, even though removing Perl removes Go. If we apply the manifest again (*i.e.*, when neither package is installed), Puppet installs Go and takes no further action, even though Perl is implicitly installed. The real issue is that this manifest is fundamentally inconsistent and cannot be fixed by adding dependencies. A system cannot have Perl removed and Go installed, so the manifest should be rejected.

An even simpler example of non-idempotence is the manifest in figure 3d, which copies `src` to `dst` and then deletes `src`. The second run of this manifest will always fail, because the first run removes `src`. This example shows that even though primitive resources are designed to be idempotent, they can be composed in ways that break idempotence.

Summary. We've introduced a small fragment of Puppet and used it to illustrate several kinds of bugs that can occur in Puppet manifests. We've argued that the root cause of these bugs is that Puppet does not ensure that manifests are deterministic and idempotent. Before we describe how Rehearsal checks these properties, we present the semantics of Puppet that Rehearsal uses.

Vertices	V	$::=$	$v_1 \mid \cdots \mid v_k$
Edges	E	$\subseteq$	$V \times V$
Vertex Labels	L	$\in$	$V \to R$
Resource Graphs	G	$\in$	$V \times E \times L$

Figure 4: Resource graphs.

3. Semantics of Puppet

This section presents a semantics for Puppet, which we develop in two stages. (1) We compile manifests to a directed acyclic graph of primitive resources, which we call a *resource graph*. The compilation process involves several passes to eliminate features that inject dependencies, change attributes, and so on. We also substitute instantiations of user-defined types with their constituent resources until only primitive resources remain. (2) Next, we model the semantics of individual resources as programs in a small imperative language of file system operations called FS. We carefully design FS so that it is expressive enough to describe the semantics of resources, yet restrictive enough to enable the static analyses we present in subsequent sections.

3.1 From Puppet to Resource Graphs

A *resource graph* G is a directed acyclic graph with vertices labeled by primitive resources. An edge exists from V_1 to V_2 if V_2 depends on V_1. At a high-level, we compile manifests

into resource graphs by converting primitive resources to nodes and dependencies to edges. To do this, we employ a number of passes to simplify manifests.

Puppet has several abstractions that allow manifests to succinctly describe dependencies. For example, user-defined types can be used to abstract over a collection of other resources and dependencies. We reduce user-defined abstractions to their constituent resources by repeatedly substituting their definitions until only primitive resources remain. In order to preserve ordering, this pass must introduce new edges between resources within instances of abstractions. In addition, resources can also be assigned to a *stage*, and stages are ordered independently of resources. To deal with this, we implement a stage elimination pass that adds edges between the constituent resources of each stage.

Certain Puppet features have non-local side effects. For example, the following expression uses a *resource collector* to update all file-resources owned by `carol` to be unreadable by others, regardless of where they are defined:

```
File<| owner == 'carol' |> { mode => "go-rwx" }
```

Unfortunately, resource collectors are not modular and make separate compilation impossible. In general, it is not possible to know the attributes of a resource until all user-defined types (which may define collectors) are eliminated as described above. The passes that tackle these kinds of expressions are necessarily global transformations.

Our compiler tackles the details described above and some other features of Puppet that we don't belabor here.

3.2 From Resources to FS Programs

Puppet has dozens of different primitive resource types that can interact with each other in subtle ways. Moreover, some resources have flags that dramatically change their behavior. To deal with this diversity, we model resources as small programs in a low-level language called FS that captures their essential effects and possible interactions. The advantage of using FS is that we can quickly add support for additional resource types and new versions of Puppet without rebuilding the rest of our analysis toolchain. In this paper, FS is an imperative language with simple operations that affect the filesystem. However, it also would be straightforward to enrich the language in several ways.

***Syntax and semantics of* FS.** The FS language, defined in figure 5 is a simple imperative language of programs that manipulate the filesystem. We model filesystems (σ) as maps from paths (p) to file contents. A file may be a regular file with some content ($\mathsf{File}(str)$) or the value Dir that represents a directory. Expressions in FS denote functions that consume filesystems and produce either a new filesystem or error (err). FS has primitive expressions to create directories ($\mathsf{mkdir}(p)$), create files ($\mathsf{creat}(p, str)$), remove files and empty directories ($\mathsf{rm}(p)$), and copy files ($\mathsf{cp}(p_1, p_2)$). Sequencing ($e_1; e_2$) and conditionals (if (a) e_1 else e_2) behave in the usual way. Predicates include the usual boolean connectives and primitive tests to check if a path is a file ($\mathsf{file?}(p)$), a directory ($\mathsf{dir?}(p)$), an empty directory ($\mathsf{emptydir?}(p)$), or contains nothing ($\mathsf{none?}(p)$).

Since FS has no loops, its programs always terminate. This is a reasonable restriction since applying Puppet resources must terminate too. FS does not have procedures or variables, but their omission doesn't affect programmers, since FS code is generated from a host language (in our case, Scala). A more important restriction is that FS programs work with a finite set of paths and file contents, so FS programs are finite state. At first glance, it appears that a program would not be affected by the state of paths that do not appear in the program text. However, the semantics of $\mathsf{rm}(p)$ and $\mathsf{emptydir?}(p)$ are affected by subpaths of p, even if they don't appear in the program. Finally, FS programs only work with a finite set of file contents. In fact, there are no operations that allow programs to read the contents of files, but this is not an essential property.

FS can easily be extended in several ways to produce higher-fidelity models of Puppet resources. *e.g.*, it is easy to imagine adding timestamps, file-permissions, and so on. Notably, these extensions would not affect the finiteness of FS programs, so we believe our analysis approach would work with these higher-fidelity models too.

Notation. We write $e_1 \equiv e_2$ when both expressions produce the same output (or error) for all input filesystems. For brevity, we use if (e_1) e_2 as shorthand for if (e_1) e_2 else id.

3.3 Modeling Resources as FS Programs

Now that we have a language of filesystem operations, we define a compilation function $\mathcal{C} : R \to e$ that maps resources to FS expressions. The actual definition has several hundred lines of code and is quite involved, but the high-level idea is to model each resource as an FS program. Even for simple resources, $\mathcal{C}$ needs to validate attributes, fill-in values for optional attributes, and produce programs that check several preconditions before applying the desired action. We now illustrate how $\mathcal{C}$ models several key resource-types.

Files and directories. Individual files and directories are the simplest resource in Puppet. The `file` resource type manages both and has several attributes that determine (1) whether it is a file or directory, (2) if it should be created or deleted, (3) if parent directories should be created, (4) the contents of a file, or (5) a source file that is copied over. Moreover, all combinations of these attributes are not meaningful, and most are optional. The $\mathcal{C}$ function addresses these details in full.

SSH keys. Some Puppet resources edit the contents of configuration files. For example, the `ssh_authorized_key` resource manages a user's public keys, where each resource is an individual line of a single file. Rather than increase the complexity of FS by including detailed file-editing commands, we model the logical structure of these

Syntax

Paths	$p ::=$	/	*Root directory*
		p/str	*Sub-path*
File Contents	$v ::=$	Dir	*Directory*
		File(str)	*File*
File Systems	$\sigma ::=$	$\langle p_1 = v_1 \cdots p_k = v_k \rangle$	
Predicates	$a ::=$	none?(p)	*Does not exist*
		file?(p)	*Is a file*
		dir?(p)	*Is a directory*
		emptydir?(p)	*Is an empty dir.*
		true	*True*
		false	*False*
		$a_1 \vee a_2$	*Disjunction*
		$a_1 \wedge a_2$	*Conjunction*
		$\neg a$	*Negation*
Expressions	$e ::=$	id	*No op*
		err	*Halt with error*
		mkdir(p)	*Create directory*
		creat(p, str)	*Create file*
		rm(p)	*Remove file/empty dir.*
		cp(p_1, p_2)	*Copy file*
		$e_1; e_2$	*Sequencing*
		if (a) e_1 else e_2	*Conditional*

Semantics

$$[\![a]\!] \in \sigma \to \mathsf{bool}$$

$$[\![\mathsf{file?}(p)]\!]\,\sigma \triangleq \exists str.\sigma(p) = \mathsf{File}(str)$$

$$[\![\mathsf{dir?}(p)]\!]\,\sigma \triangleq \sigma(p) = \mathsf{Dir}$$

$$[\![\mathsf{none?}(p)]\!]\,\sigma \triangleq p \notin dom(\sigma)$$

$$[\![\mathsf{emptydir?}(p)]\!]\,\sigma \triangleq \sigma(p) = \mathsf{Dir} \text{ and } \neg\exists str.p/str \in dom(\sigma)$$

$$\cdots$$

$$[\![e]\!] \in \sigma \to \sigma + \mathsf{err}$$

$$[\![\mathsf{id}]\!]\,\sigma \triangleq \sigma$$

$$[\![\mathsf{err}]\!]\,\sigma \triangleq \mathsf{err}$$

$$[\![\mathsf{mkdir}(p/str)]\!]\,\sigma \triangleq \begin{cases} \sigma[p/str := \mathsf{Dir}] & [\![\mathsf{dir?}(p) \wedge \mathsf{none?}(p/str)]\!]\,\sigma \\ \mathsf{err} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{creat}(p/str, str')]\!]\,\sigma \triangleq \begin{cases} \sigma[p/str := \mathsf{File}(str')] & [\![\mathsf{dir?}(p) \wedge \mathsf{none?}(p/str)]\!]\,\sigma \\ \mathsf{err} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{rm}(p)]\!]\,\sigma \triangleq \begin{cases} \sigma - p & [\![\mathsf{file?}(p) \vee \mathsf{emptydir?}(p)]\!]\,\sigma \\ \mathsf{err} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{cp}(p_1, p_2/str)]\!]\,\sigma \triangleq \begin{cases} \sigma[p_2/str := \mathsf{File}(str')] & [\![\mathsf{none?}(p_2/str) \wedge \mathsf{dir?}(p_2)]\!]\,\sigma \\ & \text{and } \sigma(p_1) = \mathsf{File}(str') \\ \mathsf{err} & \text{otherwise} \end{cases}$$

$$[\![e_1; e_2]\!]\,\sigma \triangleq \begin{cases} [\![e_2]\!]\,\sigma' & [\![e_1]\!]\,\sigma = \sigma' \\ \mathsf{err} & [\![e_1]\!]\,\sigma = \mathsf{err} \end{cases}$$

$$[\![\mathsf{if}\ (a)\ e_1\ \mathsf{else}\ e_2]\!]\,\sigma \triangleq \begin{cases} [\![e_1]\!]\,\sigma & [\![a]\!]\,\sigma \\ [\![e_2]\!]\,\sigma & \text{otherwise} \end{cases}$$

Figure 5: FS syntax and semantics.

resources in a portion of the filesystem disjoint from other files. However, this alone disguises a certain kind of determinacy bug. Consider a manifest with two resources: an `ssh_authorized_key` and a `file` that overwrites the key-file. Clearly, these resources do not commute, but by placing ssh keys in their own disjoint directory, the compiled program would be deterministic. To address this issue, our model for `ssh_authorized_key` also creates a key-file and sets its content to a unique value, enabling us to catch this kind of determinacy bug.

Packages. A package resource creates (or removes) a large number of files and directories, so we need this file list to model packages. Fortunately, there are simple command-line tools that do exactly this: *e.g.*, `apt-file` for Debian-based systems, `repoquery` for Red Hat-based systems, and `pkgutil` for Mac OS X.[5] The $\mathcal{C}$ function invokes the aforementioned tool and builds a (potentially very large) program that first creates the directory tree and then issues a sequence of creat(p, str) commands to create the files. In our model, we simply give every file p in a package a unique content str. This model is sound but conservative: some equivalences can be lost. For example, suppose a manifest has two resources: a package that creates a file p and a file resource that overwrites p with exactly the same contents as the package. This manifest would be deterministic without any dependencies, but our tool would report it as nondeterministic, due to our conservative package model. However, this situation is unlikely to arise in practice, but if it does it may indicate another mistake: it's more likely that the author meant to overwrite p with some other contents.

[5] We've tested with `apt-file` and `repoquery`.

$$[\![G]\!] \in \sigma \to 2^{\sigma + \mathsf{err}}$$

$$[\![G]\!]\,\sigma \triangleq \{[\![\mathcal{C}(L(v_1)); \cdots; \mathcal{C}(L(v_n))]\!]\,\sigma \mid \langle v_1 \cdots v_n \rangle \in \mathsf{perms}(G)\}$$
$$\text{where } G = (V, E, L)$$

Figure 6: Semantics of resource graphs.

Other resource types. We model several other resource types, including cron jobs, users, groups, services, and host-file entries. Puppet has several resources types that are only applicable to Mac OS X or Windows systems that we have not modeled. However, if we wished to analyze a manifest for these platforms, it should be easy to extend the resource compiler to support these resources. Notably, the rest of our toolchain would be unchanged as it is agnostic to the actual set of resources since it operates over FS programs.

3.4 Semantics of Resource Graphs

Now that we have a compiler from resources to FS, it is straightforward to give a semantics to resource graphs. Informally, a resource graph denotes a function from filesystems to a set of filesystems and the error state. To define this function, we take all sequences of resources that respect the order imposed by the edges, compile each resource-sequence to a sequence of FS programs, apply each program to the input state, and take the union of the results (figure 6).

A pleasant feature of this definition is that the resource graph and resource compiler abstract away the peculiarities of Puppet. We can extend $\mathcal{C}$ to support new resource types or the Puppet compiler to support even more Puppet features without changing the methods that will be discussed in the rest of this paper.

4. Determinacy Analysis

This section presents our main technical result which is a sound, complete, and scalable approach to check that resource graphs (produced from manifests) are deterministic.

Definition 1 (Determinism)**.** *A resource graph G is deterministic, if for all filesystems σ, $|[\![G]\!]\,\sigma| = 1$.*

This property does not preclude a manifest from always producing an error on some or even all inputs. Any non-trivial manifest makes assumptions about the initial state (*e.g.*, the operating system in use) and thus will raise an error if it is applied to a machine that is not in the right initial state. Determinism simply guarantees that successes and failures will be predictable.

Our approach has three major steps:

1. The first step is to reduce the number of paths that we need to reason about. Even a small manifest may manipulate several hundred paths and tracking their state over hundreds of intermediate states can be intractable. We observe that resources often modify paths p that are not accessed by any other resource, thus operations on these paths can be safely eliminated without affecting the result of the determinism-check.
2. The next step is to reduce the number of permutations of the resource graph, which can grow exponentially with the number of resources. The natural approach is to use partial-order reduction with a fast, commutativity check. However, the obvious approach, based on calculating read- and write-sets is not effective because many resources may create overlapping directories (*e.g.*, `/usr` and `/etc`). We observe that this is a form of false sharing and develop a commutativity check that accounts for idempotent directory creation.
3. The final step is to encode the semantics of the manifest as a decidable formula for an SMT solver that is satisfiable if and only if the program is non-deterministic. Our encoding relies on the fact that programs manipulate a finite set of paths that are statically known. However, the result of some operations may be affected by the state of paths that do not appear in the program itself. We carefully bound the domain of paths to ensure our approach is complete.

We first present our encoding of manifests as formulas.

4.1 From Resource Graphs to Formulas

The function $\Phi(e)$ produces a collection of formulas that encode the semantics of the expression e (figure 7). In these formulas, two boolean variables determine whether the initial and final states are non-error states and every path is modeled by two variables that describe their initial and final state. These path-state variables are only meaningful in a non-error state. More concretely, a logical state (Σ) is a record of two components: (1) $\Sigma.\mathsf{ok}$ is a formula that is true if the current state is not the error state and (2) $\Sigma.\mathsf{fs}$ maps paths to formulas that describe their state. We could employ McCarthy's theory of arrays [21] to encode this map, but it's more efficient to encode it directly with one formula per path. To encode resource graphs G as formulas, we use the function $\Phi_G(G)$, defined in the same figure, which maps the input logical state to a set of output logical states by evaluating each expression on the fringe with $\Phi(e)$ and recurring on the subgraph that has e removed.

To prove this encoding sound and complete, we need to relate concrete states returned by the evaluator to logical states. This is mostly routine, but the domain of logical filesystems has to be large enough: if a program reads or writes to a path p, then there must be a formula $p \in \mathit{dom}(\Sigma.\mathsf{fs})$. For example, note that `mkdir(/a/b)` reads `/a` and writes `/a/b`.

Lemma 1 (Soundness and completeness)**.** *For all σ and G:*

1. *$\sigma' \in [\![G]\!]\,\sigma$ iff there exists $\Sigma \in \Phi_G(\langle \mathsf{ok} = \mathsf{true}, \mathsf{fs} = \sigma \rangle, e)$ such that $\Sigma \vdash \langle \mathsf{ok} = \mathsf{true}, \mathsf{fs} = \sigma' \rangle$.*
2. *$\mathsf{err} \in [\![G]\!]\,\sigma$ iff there exists $\Sigma \in \Phi_G(\langle \mathsf{ok} = \mathsf{true}, \mathsf{fs} = \sigma \rangle, e)$ such that $\Sigma.\mathsf{ok} \vdash \mathsf{false}$.*

4.2 Checking Determinism

With resource graphs encoded as formulas, it should be straightforward to use a theorem prover to check determinism (though we have yet to address scalability issues). Since $\Phi_G(G)$ maps an input logical state to a set of output logical states, the resource graph should be deterministic, if and and only if there does not exist an input logical state that produces two different logical states. *i.e.*, the following formula should be unsatisfiable:

$$\exists \Sigma_1, \Sigma_2, \Sigma_3 . \Sigma_2 \in \Phi_G(G)\Sigma_1 \wedge \Sigma_3 \in \Phi_G(G)\Sigma_1 \wedge \Sigma_2 \neq \Sigma_3$$

The subtlety here is that the domain of $\Phi_G(G)$ to be large enough to find a counterexample when G is a non-deterministic resource graph.

To understand the issue, consider the simpler problem of checking whether two expressions are inequivalent, $e_1 \not\equiv e_2$, which is the essence of checking non-determinism. At first glance, it appears that expressions only read and write to the paths that appear in it and the result of an expression is not affected by the state of any other paths. That is, if we have a state σ such that $[\![e_1]\!]\,\sigma \not\equiv [\![e_2]\!]\,\sigma$ then for paths p that do not appear in either e_1 or e_2, $([\![e_1]\!]\,\sigma)(p) = ([\![e_2]\!]\,\sigma)(p)$. But, this equation is wrong.

The $\mathsf{emptydir?}(p)$ predicate poses a problem, since it depends on the state of the immediate children of p, including those that may not appear in the program. Consider the following inequality, where the only difference between the programs is that one checks if the directory is empty and the other only checks that it is a directory:

$$\begin{aligned} &\mathsf{if}\ (\mathsf{emptydir?}(\texttt{/a}))\ \mathsf{id}\ \mathsf{else}\ \mathsf{err} \\ \not\equiv\ &\mathsf{if}\ (\mathsf{dir?}(\texttt{/a}))\ \mathsf{id}\ \mathsf{else}\ \mathsf{err} \end{aligned}$$

Any input filesystem that demonstrates the inequality must have a file (or directory) within `/a`. However, if we construct

$$
\begin{array}{rl}
\text{Logical Formulas} & \phi ::= \cdots \\
\text{Logical Filesystems} & \hat{\sigma} ::= \langle p_1 = \phi_1 \cdots p_k = \phi_k \rangle \\
\text{Logical States} & \Sigma ::= \langle \mathsf{ok} = \phi, \mathsf{fs} = \hat{\sigma} \rangle
\end{array}
$$

$$encPred(\hat{\sigma}, b) \in \phi$$

$$
\begin{array}{rl}
ok(e) \in \hat{\sigma} \to \mathsf{bool} \\
ok(\mathsf{id})\hat{\sigma} \triangleq \mathsf{true} \\
ok(\mathsf{err})\hat{\sigma} \triangleq \mathsf{false} \\
ok(\mathsf{mkdir}(p/str))\hat{\sigma} \triangleq \hat{\sigma}(p) = \mathsf{dir} \wedge \hat{\sigma}(p/str) = \mathsf{dne} \\
ok(\mathsf{creat}(p/str, str'))\hat{\sigma} \triangleq \hat{\sigma}(p/str) \triangleq \mathsf{dne} \wedge \hat{\sigma}(p) = \mathsf{dir} \\
ok(\mathsf{rm}(p))\hat{\sigma} \triangleq \exists c.\hat{\sigma}(p) = \mathsf{file}(c) \wedge \\
\qquad \forall str.p/str \in dom(\hat{\sigma}) \Rightarrow \hat{\sigma}(p/str) = \mathsf{dne} \\
ok(\mathsf{cp}(p_1, p_2/str))\hat{\sigma} \triangleq \exists str'.\hat{\sigma}(p_1) = \mathsf{File}(str') \wedge \\
\qquad \hat{\sigma}(p_2) = \mathsf{Dir} \wedge \hat{\sigma}(p_2/str) = \mathsf{none?} \\
ok(e_1; e_2)\hat{\sigma} \triangleq ok(e_1)\hat{\sigma} \wedge ok(e_2)(f(e_1)\hat{\sigma}) \\
ok(\mathsf{if}\ (b)\ e_1\ \mathsf{else}\ e_2)\hat{\sigma} \triangleq \mathsf{if}\ (encPred(\hat{\sigma}, b))\ ok(e_1)\hat{\sigma}\ \mathsf{else}\ ok(e_2)\hat{\sigma}
\end{array}
$$

$$
\begin{array}{rl}
f(e) \in \hat{\sigma} \to \hat{\sigma} \\
f(\mathsf{id})\hat{\sigma} \triangleq \hat{\sigma} \\
f(\mathsf{err})\hat{\sigma} \triangleq \hat{\sigma} \\
f(\mathsf{mkdir}(p/str))\hat{\sigma} \triangleq \hat{\sigma}[p/str := \mathsf{Dir}] \\
f(\mathsf{creat}(p/str, str'))\hat{\sigma} \triangleq \hat{\sigma}[p/str := \mathsf{file}(str')] \\
f(\mathsf{rm}(p))\hat{\sigma} \triangleq \hat{\sigma}[p := \mathsf{none?}] \\
f(\mathsf{cp}(p_1, p_2/str))\hat{\sigma} \triangleq \hat{\sigma}[p_2/str := \hat{\sigma}(p_1)] \\
f(e_1; e_2)\hat{\sigma} \triangleq f(e_2)(f(e_1)\hat{\sigma}) \\
f(\mathsf{if}\ (b)\ e_1\ \mathsf{else}\ e_2)\hat{\sigma} \triangleq \mathsf{if}\ (encPred(\hat{\sigma}, b))\ f(e_1)\hat{\sigma}\ \mathsf{else}\ f(e_2)\hat{\sigma}
\end{array}
$$

$$
\begin{array}{rl}
\Phi(e) \in \Sigma \to \Sigma \\
\Phi(e)\langle \mathsf{ok} = b, \mathsf{fs} = \hat{\sigma} \rangle \triangleq \langle \mathsf{ok} = b \wedge ok(e)\hat{\sigma}, \mathsf{fs} = f(e)\hat{\sigma} \rangle
\end{array}
$$

$$
\begin{array}{rl}
\Phi_G(G) \in \Sigma \to 2^{\Sigma} \\
\Phi_G((\emptyset, E))\Sigma \triangleq \{\Sigma\} \\
\Phi_G(G)\Sigma \triangleq \bigcup \Phi_G(G - e)(\Phi(e)\Sigma) \\
\qquad \text{where } inDegree(e) = 0
\end{array}
$$

Figure 7: Encoding FS as logical formulas.

$$
\begin{array}{rl}
dom(a) \in 2^p \\
dom(\mathsf{file?}(p)) \triangleq \{p\} \\
dom(\mathsf{emptydir?}(p)) \triangleq \{p, p/str\}\ str \text{ is fresh} \\
\cdots \\
dom(e) \in 2^p \\
dom(\mathsf{mkdir}(p/str)) = \{p, p/str\} \\
dom(\mathsf{creat}(p/str, str')) = \{p/str, p\} \\
dom(\mathsf{rm}(p)) = \{p/str\}\ str \text{ is fresh} \\
dom(\mathsf{cp}(p_1, p_2/str)) = \{p_1, p_2, p_2/str\} \\
dom(e_1; e_2) = dom(e_1) \cup dom(e_2) \\
dom(\mathsf{if}\ (a)\ e_1\ \mathsf{else}\ e_2) = dom(a) \cup dom(e_1) \cup dom(e_2)
\end{array}
$$

Figure 8: Bounding the domain of FS programs.

a logical filesystem using only the paths that appear in the program text, we will not find this counterexample. A similar problem affects $\mathsf{rm}(p)$. The function in figure 8 addresses this problem by adding fresh files in directories that are removed or tested for emptiness to avoid this bug. We can now prove that equivalence-checking is complete.

Lemma 2 (Completeness—equivalence)**.** *If:*

- $[\![e_1]\!]\,\sigma \neq [\![e_2]\!]\,\sigma$ *and*
- $dom(\hat{\sigma}') = dom(e_1) \cup dom(e_2)$

then $\Phi(\langle \mathsf{ok} = \mathsf{true}, \mathsf{fs} = \hat{\sigma}' \rangle, e_1) \neq \Phi(\langle \mathsf{ok} = \mathsf{true}, \mathsf{fs} = \hat{\sigma}' \rangle, e_2)$.

Soundness is straightforward. A model for the formula can be easily transformed into a counterexample filesystem.

Lemma 3 (Soundness—equivalence)**.** *If:*

- $\Phi(\langle \mathsf{ok} = \mathsf{true}, \mathsf{fs} = \hat{\sigma} \rangle, e_1) \neq \Phi(\langle \mathsf{ok} = \mathsf{true}, \mathsf{fs} = \hat{\sigma} \rangle, e_2)$ *and*
- $\hat{\sigma} \vdash \sigma$

then $[\![e_1]\!]\,\sigma \neq [\![e_2]\!]\,\sigma$.

We use these lemmas to prove that that determinism checking is sound and complete. In the theorem below, $dom(e)$ is lifted to $dom(G)$ in the obvious way.

Theorem 1. *(Determinism) G is deterministic, if and only if there exists* Σ_1*,* Σ_2*, and* Σ_3 *such that* $\Sigma_2 \in \Phi_G(G)\Sigma_1 \wedge \Sigma_3 \in \Phi_G(G)\Sigma_1 \wedge \Sigma_2 \neq \Sigma_3$ *is unsatisfiable, where* $dom(\Sigma_1) = dom(\Sigma_2) = dom(\Sigma_3) = dom(G)$.

4.3 Commutativity and Directory Creation

Modeling all valid permutations of resources can produce formulas that are intractably large. For example, suppose a resource graph G has exactly two nodes a and b that do not have any ancestors. The naive approach considers evaluating either node first and then recurs on the two subgraphs $G - a$ and $G - b$. When the sub-graphs also have several nodes without any ancestors, the size of the generated formula grows intractably large. A significantly better approach is to use a fast, syntactic commutativity check to rule out permutations that don't need to be explored, similar to partial-order reduction. Note that it is not sufficient to check that a and b commute. For example, $b; a; c$ and $b; c; a$ are valid permutations in the following graph:

$$a \qquad b \longrightarrow c$$

We can only conclude that they are equivalent if we know that a commutes with both b and c. Therefore, to avoid recurring on both $G - a$ and $G - b$, we need to prove that a (or b) commute with all nodes that are not ancestors of a, as shown in figure 9a.

Next, we need a fast, syntactic commutativity check, which should be straightforward to do for FS. Surprisingly, the natural approach does not work. A typical commutativity check works as follows: to check if e_1 and e_2 commute, calculate the set of locations that each reads and writes. If the expressions don't have any overlapping writes and e_1 does not read any locations that e_2 writes (and vice versa), then they do commute. If not, they may or may not commute and we need to semantically check both orderings.

This approach is not effective for Puppet, due to the semantics of packages. Typical packages install files to shared directories (*e.g.* `/usr/bin`, `/etc`, and so on) and will create these directories if necessary. Therefore, the conventional

$$\Phi_G(G) \in \Sigma \to 2^\Sigma$$
$$\Phi_G((\emptyset, E))\Sigma \triangleq \{\Sigma\}$$
$$\Phi_G(G)\Sigma \triangleq \Phi_G(G-e)(\Phi(e)\Sigma) \quad \text{where } \mathit{inDegree}(e) = 0 \quad \forall e' \in G.\neg \mathit{ancestor}(e', e) \Rightarrow e'; e \equiv e; e'$$
$$\Phi_G(G)\Sigma \triangleq \bigcup \Phi_G(G-e)(\Phi(e)\Sigma) \quad \text{where } \mathit{inDegree}(e) = 0$$

(a) Incorporating the commutativity-check.

Abstract Values $\tilde{v} ::= \bot \mid R \mid W \mid D$

Abstract State $\tilde{\sigma} ::= \langle p_1 = \hat{v}_1 \cdots p_k = \hat{v}_k \rangle$

$\bot \sqsubset R, D \sqsubset W$

$$[e]_C \in \tilde{\sigma} \to \tilde{\sigma}$$
$$[\mathsf{if}\ (\neg\mathsf{dir?}(p/str))\ \mathsf{mkdir}(p/str)]_C\tilde{\sigma} \triangleq \begin{cases} \tilde{\sigma}[p/str := D] & \hat{\sigma}(p/str) \sqsubseteq D \\ & \text{and } \tilde{\sigma}(p) = D \\ \tilde{\sigma}[p/str := W] & \text{otherwise} \end{cases}$$
$$[\mathsf{mkdir}(p)]_C\hat{\sigma} \triangleq \tilde{\sigma}[p := W]$$
$$[\mathsf{creat}(p, str)]_C\hat{\sigma} \triangleq \tilde{\sigma}[p := W]$$
$$[e_1; e_2]_C\tilde{\sigma} \triangleq [e_2]_C([e_1]_C\tilde{\sigma})$$
$$\cdots$$

(b) Checking commutativity.

Figure 9: Commutativity checks eliminate the number of permutations that need to be generated.

approach cannot prove that packages commute. Manifests that installs several packages typically do not specify any dependencies between them, so this issue arises frequently.

To address this issue, we use an abstract interpretation that maps each path p to the abstract values $\bot$, R, W, and D (figure 9b). These values indicate that the expression either does not affect p ($\bot$), reads from p (R), writes to p (W), or ensures that p is a directory (D). A $\mathsf{mkdir}(p)$ expression that doesn't first check if p already exists is simply a write (W). Only a guarded $\mathsf{mkdir}(p)$ can ensure p is a directory, such as these expressions:

$$\begin{array}{ll} & \mathsf{if}\ (\neg\mathsf{dir?}(\mathtt{p}))\ \mathsf{mkdir}(\mathtt{p}) \\ \equiv & \mathsf{if}\ (\mathsf{none?}(p))\ \mathsf{mkdir}(p)\ \mathsf{else\ if}\ (\mathsf{file?}(p))\ \mathsf{err\ else\ id} \end{array}$$

In addition, the analysis ensures that expressions create directory trees in a reasonable order. For example, an expression that creates `/a` before `/a/b` is not equivalent to an expression that tries to create `/a/b` before `/a`. However, two expressions that create sibling directories do commute. To ensure that these properties hold, we map p/str to D, only if p is already mapped to D.

We can use the result of this analysis to check that expressions commute, even if they create overlapping directory trees.

Lemma 4. *For all e_1 and e_2, if:*

1. $\{p \mid [e_1]_C\bot(p) = R\} \cap \{p \mid [e_2]_C\bot(p) = W\} = \emptyset$,
2. $\{p \mid [e_1]_C\bot(p) = W\} \cap \{p \mid [e_2]_C\bot(p) = R\} = \emptyset$,
3. $\{p \mid [e_1]_C\bot(p) = D\} \cap \{p \mid [e_2]_C\bot(p) \in \{R, W\}\} = \emptyset$, *and*
4. $\{p \mid [e_1]_C\bot(p) \in \{R, W\}\} \cap \{p \mid [e_2]_C\bot(p) = D\} = \emptyset$

then $e_1; e_2 \equiv e_2; e_1$.

4.4 Pruning Files from Resources

The syntactic commutativity check mentioned above eliminates the need to explore different permutations of resources that are obviously equivalent to each other. However, even a single permutation that installs several large resources can make formulas needlessly large. For example, suppose a manifest installs a large package (*e.g.*, `git`, which has over 500 files) and then doesn't read or write to any of the files that the package creates. Intuitively, we should be able to completely *eliminate* resources that are not observed by the rest of the manifest.

However, there are situations where resources must interfere. It is quite common for a manifest to update a default configuration file created by a package. For example, the manifest in figure 3a installs the Apache web server and supplies a site-specific configuration file that should overwrite the default configuration. Even in this situation, the manifest does not update most of the other 200+ files that the Apache package creates. Intuitively, we should be able to *shrink* resources so that we don't have to track the state of files that cannot affect the outcome of the determinism-check.

In this section, we formalize these two observations using two simple analyses.

Eliminating Resources. Notice that a determinism-check is essentially a conjunction of equivalence-checks between all valid permutations of resources. For example, the following resource graph has eight valid permutations of the four resources shown:

$$a \longrightarrow c \longleftarrow b \qquad d$$

A naive determinism-check would generate all permutations and verify that they are equivalent:

$$\begin{array}{r} a; b; c; d \equiv a; b; d; c \equiv a; d; b; c \equiv b; a; c; d \\ \equiv b; a; d; c \equiv b; d; a; c \equiv d; a; b; c \equiv d; b; a; c \end{array}$$

However, suppose we use our commutativity check to determine that c and d commute. We could then rewrite all the permutations that end with $c; d$ to instead end with $d; c$, which gives us a series of permutations that all end in c. In general, $e_1; e \equiv e_2; e$, if and only if $e_1 \equiv e_2$. Therefore, we can completely eliminate c without changing the result of the equivalence check.

In general, if a resource commutes with all other resources that may be evaluated after it in the resource graph, then that resource can be eliminated without affecting the result of the determinism-check. Moreover, eliminating one resource often allows their parents to be eliminated. For example, suppose that b commutes with a and d. Eliminating

c, as discussed above, allows us to then eliminate b by the same argument. However, trying to eliminate b first would fail, since it does not commute with c, which may-succeed b. In practice, a true dependency $a \rightarrow b$ indicates that b truly depends on the effects of a and thus the two resources do not commute. In our experience, we've found it most effective to eliminate resources by starting with resources at the fringe of the dependency graph that are not required by any other resources.

Shrinking Resources. There are several cases where large resources cannot be completely eliminated. However, they can be shrunk as follows. In general, if a resource writes to a path p such that (1) other resources in the manifest do not observe the state of p and (2) other resources in the manifest do not affect the state of p then we can eliminate writes to p without changing whether the manifest is deterministic or not. Moreover, the encoding of FS programs as formulas can then exploit the fact that p is read-only and use a single variable to represent the state of p, instead of using new variables for each state. This can dramatically reduce the number of variables needed to encode the program.

Consider the problem of shrinking two expressions e_1 and e_2 to e_1' and e_2', such that $e_1 \equiv e_2$ if and only if $e_1' \equiv e_2'$. If both expressions leave a path p in the same state, it should be possible to shrink both expressions by removing their writes to p. However, to implement idempotent operations, resources tend to have a complex series of reads and writes (section 3.3). Nevertheless, a resource that writes to p typically ensures that p is either placed in a definite state or signals an error if it cannot do so. We say that these resources make *definitive writes* to p. Therefore, if both expressions make the same definitive write to p, then we can eliminate writes to p.

We detect definitive writes using the abstract interpretation sketched in figure 10b, which produces an abstract heap, $\hat{\sigma}$ that maps paths p to abstract values that characterize the effect of an expression on p over all input states:

- If $\hat{\sigma}(p) = \mathsf{dir}$, the expression ensures that p is a directory (or signals an error).
- If $\hat{\sigma}(p) = \mathsf{file}(str)$, the expression ensures that p is a file with contents str(or signals an error).
- If $\hat{\sigma}(p) = \mathsf{dne}$, the expression ensures that p does not exist (or signals an error).
- If $\hat{\sigma}(p) = \bot$, the expression does not read or write p.
- If $\hat{\sigma}(p) = \top$, the expression has an indeterminate effect on p.

Lemma 5. *If* $(\widehat{[\![e]\!]}\bot)(p) \sqsubset \top$ *then for all states* σ_1 *and* σ_2, $([\![e]\!]\,(\sigma_1))(p) = ([\![e]\!]\,(\sigma_2))(p)$.

If the abstract interpretation determines that e_1 and e_2 set a path p to the same definite value, we should be able to prune writes to p from both expressions. However, consider the two equivalent expressions below:

$$\mathsf{mkdir}(p/str); \mathsf{if}\ (\mathsf{dir?}(p/str))\ \mathsf{id}\ \mathsf{else}\ \mathsf{err} \equiv \mathsf{mkdir}(p/str)$$

The expressions on either side ensure that p/str is a directory. However, if we naively replace $\mathsf{mkdir}(p/str)$ with id, we get the following wrong result:

$$\mathsf{id}; \mathsf{if}\ (\mathsf{dir?}(p/str))\ \mathsf{id}\ \mathsf{else}\ \mathsf{err} \neq \mathsf{id}$$

To correctly eliminate writes to p, we need to also transform expressions that read from p to account for the effect that the write would have had. In our example, the test $\mathsf{dir?}(p/str)$ will always be true, since it follows $\mathsf{mkdir}(p/str)$. The insight is that when writes to p are eliminated, we need to transform all expressions that subsequently read or write to p. In our example, we need to transform $\mathsf{dir?}(p/str)$ to true.

The pruning function, *prune*(p, e), eliminates writes to p by preserving reads in this manner (figure 10a). The function correctly handles programs where a write to p is followed by other reads and writes to p by partial evaluation.

The following lemma states that the same definitive write from e_1 and e_2 doesn't change their (in-)equivalence.

Lemma 6. *If* $(\widehat{[\![e_1]\!]}\bot)(p) = (\widehat{[\![e_2]\!]}\bot)(p) = \hat{v}$ *and* $\hat{v} \sqsubset \top$ *then* $e_1 \equiv e_2$ *if and only if* *prune*$(p, e_1) \equiv$ *prune*(p, e_2).

Although pruning eliminates writes to p, it does not eliminate reads from p. However, eliminating writes ensures that p is a read-only path. When we encode the expression as a logical formula, the encoding can optimize for read-only paths by using a single variable to represent the initial state of the path, which then remains unchanged.

Pruning for determinism checking. Since a determinism check encodes equalities between all permutations of resources, we could also apply the abstract interpretation to all permutations, but this would be intractable. Instead, we apply the abstract interpretation to each resource in isolation to find paths that are definitively written by exactly one resource and only prune these paths. This conservative approach works well in practice.

4.5 Summary

These are the three major techniques that Rehearsal uses to make determinism-checking scale. We've also outlined how each step preserves (in-)equivalences, so the approach is sound and complete.

Other approaches. We have tried two other techniques for checking determinism that are less effective than the methodology discussed in this section.

1. We developed a dynamic analysis that simply installed resources in different valid permutations within independent Docker containers. The Docker API makes it easy to see how a container has updated its filesystem. However, installing resources takes time and it took our prototype several hours to verify small manifests with less than ten resources. (We fully utilized a four-core machine with 16 GB RAM.) In contrast, our static analysis checks determinism in seconds.

$$
\begin{aligned}
P[\![\cdot]\!] &\in e \times p \times \sigma \to e \times \sigma \\
P[\![\mathsf{id}]\!]\ p\ \sigma &= (\mathsf{id}, \sigma) \\
P[\![\mathsf{err}]\!]\ p\ \sigma &= (\mathsf{err}, \sigma) \\
P[\![\mathsf{mkdir}(p)]\!]\ p\ \sigma &= (\mathsf{err}, \sigma) \text{ if } \sigma(p) = \mathsf{Dir} \text{ or } \sigma(p) = \mathsf{File}(str) \\
P[\![\mathsf{mkdir}(p/str)]\!]\ p/str\ \sigma &= (\mathsf{if}\ (\mathsf{none?}(p/str) \wedge \mathsf{dir?}(p))\ \mathsf{id}\ \mathsf{else}\ \mathsf{err}, \sigma[p/str := \mathsf{Dir}]) \\
P[\![\mathsf{mkdir}(p')]\!]\ p\ \sigma &= (\mathsf{mkdir}(p'), \sigma) \text{ if } p \neq p' \\
P[\![\mathsf{creat}(p, str)]\!]\ p\ \sigma &= (\mathsf{err}, \sigma) \text{ if } \sigma(p) = \mathsf{Dir} \text{ or } \sigma(p) = \mathsf{File}(str') \\
P[\![\mathsf{creat}(p/str, str')]\!]\ p/str\ \sigma &= (\mathsf{if}\ (\mathsf{none?}(p/str) \wedge \mathsf{dir?}(p))\ \mathsf{id}\ \mathsf{else}\ \mathsf{err}, \sigma[p/str := \mathsf{File}(str')]) \\
P[\![\mathsf{creat}(p', str)]\!]\ p\ \sigma &= (\mathsf{creat}(p', str), \sigma) \text{ if } p \neq p' \\
P[\![\mathsf{rm}(p)]\!]\ p\ \sigma &= (\mathsf{err}, \sigma) \text{ if } p \notin dom(\sigma) \text{ or } \exists str.p/str \in dom(\sigma) \\
P[\![\mathsf{rm}(p)]\!]\ p\ \sigma &= (\mathsf{if}\ (\mathsf{file?}(p) \vee \mathsf{emptydir?}(p))\ \mathsf{id}\ \mathsf{else}\ \mathsf{err}, \sigma - p) \\
P[\![\mathsf{rm}(p')]\!]\ p\ \sigma &= (\mathsf{rm}(p'), \sigma) \text{ if } p \neq p' \\
&\cdots \\
P[\![\mathsf{if}\ (a)\ e_1\ \mathsf{else}\ e_2]\!]\ p\ \sigma &= (e_1, \sigma) \text{ if } [\![a]\!]\ \sigma = \mathsf{true} \\
&\cdots
\end{aligned}
$$

$$
\begin{aligned}
&prune \in p \times e \to e \\
&prune(p, e) = p' \text{ where } (p', \sigma) = P[\![e]\!]\ p\ \cdot
\end{aligned}
$$

(a) Pruning definitive writes.

Abs. Values $\hat{v} ::= \bot \mid \top \mid \mathsf{dir} \mid \mathsf{file}(str) \mid \mathsf{dne}$

Abs. State $\hat{\sigma} ::= \langle p_1 = \hat{v}_1 \cdots p_k = \hat{v}_k \rangle$

$$\bot \sqsubset \mathsf{dir}, \mathsf{file}(str), \mathsf{dne} \sqsubset \top$$

$$
\begin{aligned}
\widehat{[\![e]\!]} &\in \hat{\sigma} \to \hat{\sigma} \\
\widehat{[\![\mathsf{id}]\!]}\hat{\sigma} &= \hat{\sigma} \\
\widehat{[\![\mathsf{err}]\!]}\hat{\sigma} &= \hat{\sigma} \\
\widehat{[\![\mathsf{mkdir}(p)]\!]}\hat{\sigma} &= \hat{\sigma}[p := \mathsf{dir}] \\
\widehat{[\![\mathsf{creat}(p, str)]\!]}\hat{\sigma} &= \hat{\sigma}[p := \mathsf{file}(str)] \\
\widehat{[\![\mathsf{rm}(p)]\!]}\hat{\sigma} &= \hat{\sigma}[p := \mathsf{dne}] \\
\widehat{[\![\mathsf{if}\ (a)\ e_1\ \mathsf{else}\ e_2]\!]}\hat{\sigma} &= \widehat{[\![e_1]\!]}\hat{\sigma} \sqcap \widehat{[\![e_2]\!]}\hat{\sigma} \\
\widehat{[\![e_1; e_2]\!]}\hat{\sigma} &= \widehat{[\![e_2]\!]}(\widehat{[\![e_1]\!]}\hat{\sigma})
\end{aligned}
$$

(b) Detecting definitive writes.

Figure 10: Shrinking resources.

2. Instead of using an SMT solver, we tried to encode FS programs as binary-decision diagrams (BDDs) by exploiting the natural hierarchy of paths to pick a good variable order. (*e.g.*, `a` $<$ `a/b`.) In our experience, the SMT solver was faster and significantly easier to use. For example, properties such as "all paths must be distinct" are very easy to express using `distinct` constraints in an SMT solver.

5. Beyond Determinism

After we've checked that a manifest is deterministic, we can treat it as an expression rather than a resource graph: we can pick any valid ordering of the resources (determinism ensures that they are all equivalent) and sequence them to form a single expression e. We emphasize that while resource graphs denote relations, FS expressions denote functions. This lets Rehearsal check several properties quickly and easily.

Invariants. We've seen that Puppet is actually very imperative. A manifest that declares a `file` resource may overwrite it using some other resource, which is typically undesirable. Rehearsal checks for this issue using the following formula, which is unsatisfiable if e ensures that p is always a file with content str:

$$\exists \hat{\sigma}.ok(e)\hat{\sigma} \wedge f(e)\hat{\sigma}(p) \neq \mathsf{file}(str)$$

It is easy to imagine checks for several other invariants.

Idempotence. We discussed in section 2 that idempotence is a critical property of Puppet manifests. To test if a manifest is idempotent, we simply check if $e \equiv e; e$ holds.

We emphasize that these checks are efficient because they do not have to consider all permutations of resources. Moreover, these simple checks would be unsound if applied to non-deterministic manifests.

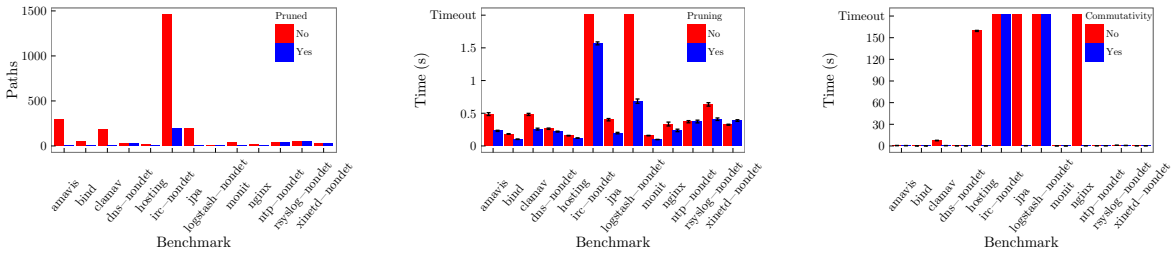
6. Evaluation

Rehearsal is implemented in Scala and uses the Z3 Theorem Prover [11] as its SMT solver. The majority of the codebase is the frontend that turns manifests into FS expressions. To model packages, Rehearsal needs to query an OS package manager. For portability, we've built a web service for Rehearsal that can query the package manager for several operating systems. The service returns the package listing in a standardized format and stores the result in a database to speedup subsequent queries. Our current deployed service has Ubuntu and CentOS running in containers and it is easy to add support for other operating systems.

Note that the times reported in this section do not include the time required to fetch package listings. The package querying tools (`apt-cache` and `repoquery`) can take several seconds to run, which is why our web service caches their results.

Third-party benchmarks. We benchmark the determinism checker on a suite of 13 Puppet configurations gleaned from GitHub and Puppet Forge. We specifically chose benchmarks that did not use `exec` resources as detailed further in section 8. We manually verified that six of them have determinism bugs and that seven do not. For each non-deterministic program, we developed a fix and verified that Rehearsal reports that it is deterministic and idempotent. We repeat all timing experiments ten times and report the average. We perform all experiments on a quad-core 3.5 GHz Intel Core i5 with 8GB RAM.

Figure 11 shows the effect of pruning on Rehearsal's determinacy analysis. In the figure, the non-deterministic manifests are marked *-nondet*. Without commutativity checking, four benchmarks do not complete in over ten minutes and one takes over two minutes to run (figure 11c). Even with commutativity-checking, two benchmarks timeout after ten minutes. However, when commutativity-checking is coupled



(a) Paths per state. (b) Shrinking and eliminating resources. (c) Checking commutativity.

Figure 11: Benchmarking determinacy analysis.

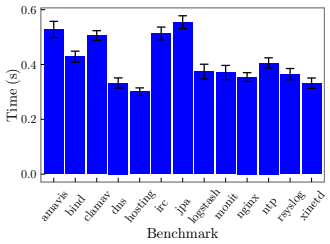


Figure 12: Benchmarking idempotence checking.

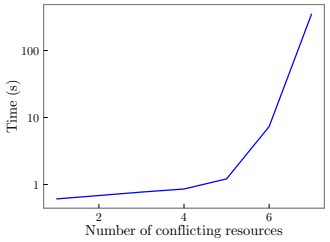


Figure 13: Scalability with n interfering resources.

with file pruning (section 4.4), all benchmarks complete in less than two seconds (figure 11b). Figure 11a shows the number of files in each manifest, with and without pruning. (Note that commutativity-checking does not affect the number of files.) As expected, the runtime of benchmarks corresponds to the number of files that need to be modeled.

Figure 12 reports the time required by the idempotence check is less than one second on all benchmarks. In practice, the idempotence check would be preceded by a determinism check, which typically takes more time to complete.

Synthetic benchmarks. The benchmarks above suggest that commutativity checking and pruning are effective in practice. However, it is quite straightforward to construct an artificial scenario where the commutativity check is ineffective. The natural way to construct this benchmark is to have n unordered file-resources that write to the same path. This renders the commutativity-check useless, so Rehearsal is forced to explore all $n!$ paths through the resource graph. Moreover, the file cannot be pruned either. Figure 13 shows the running time grows non-linearly with n. In fact, even with $n = 6$, the running time exceeded two minutes.

Although the simple benchmark described above can be constructed using FS, it is not a valid Puppet manifest, since Puppet does not allow multiple file-resources to affect the same path. A working alternative is to find n conflicting packages that all create the same file p and try to install all of them simultaneously. Even in this scenario, Rehearsal can determine that the manifest is non-deterministic relatively quickly (*i.e.*, that the formula is satisfiable). However, we can force the manifest to be deterministic by using a single file resource that updates the contents of p after all the packages are installed:

```
# All packages create a file /a
package{'A-1': before => File['/a'] }
package{'A-2': before => File['/a'] }
package{'A-3': before => File['/a'] }
...
file{'/a': content => 'x' }
```

The final file-resource makes the manifest deterministic, which forces the solver to construct a proof of unsatisfiability instead of terminating early with a satisfying assignment. We believe that this kind of scenario is very unlikely to arise in practice.

Bugs found. Rehearsal found determinism bugs in six benchmarks (including a previously undiscovered bug). The bugs are of the kind we described in section 2. Specifically, several benchmarks omitted a necessary dependency between a package and a configuration file. In addition, one benchmark omitted a dependency between a user account and SSH keys for the user. Broadly speaking, resource-types such as files and packages have a well-understood semantics, but users may not understand their interactions.

7. Related Work

Other system configuration languages. Several system configuration languages have been developed over decades

of research, many of which are surveyed by Delaet and Joosen [10]. To the best of our knowledge, the kind of verification tools we have developed for Puppet have not been developed for these languages. Instead, we highlight how several languages differ from Puppet and consider what it would take to adapt our approach for them.

Hagemark and Zadeck's Site tool [16] has a DSL that is closely related to Puppet. A Sitefile describes bits of configurations in "classes" that can be composed in several ways. Site traverses these classes in topological order and can also suffer missing dependencies, which our techniques detect.

LCFG [1] provides built-in components for configuring common applications. However, while new LCFG components have to be authored in Perl, Puppet encourages average users to build their own abstractions using the Puppet DSL. An inter-component dependency in LCFG requires coordination between the configuration file and Perl code (using "context variables"). Rehearsal leverages Puppet's high-level DSL which makes all dependencies manifest. Building similar tools for LCFG would be difficult due to Perl.

Anderson and Herry [2] develop a denotational semantics for the SmartFrog configuration language that faithfully models its non-deterministic semantics. They show that their model helps resolve several implementation issues, though ordering issues remain. They argue that system configuration languages need formal models and warn that popular languages gain features faster than formal models can be developed. Our work shows that it is possible to model and analyze a significant fraction of a large system configuration language, but we don't disagree with their conclusions.

Engage [13] is a system for deploying and configuring distributed applications that can specify complex, inter-machine dependencies, where values computed by one resource at runtime can be used as inputs to another resource on a different machine. Puppet is more limited and does not support orchestration. To manage the life-cycle of a resource, Engage users have to write drivers in Python. Although the Engage type-checker ensures that resources are composed correctly, it assumes that these Python drivers are error-free. In contrast, the Puppet DSL performs operations similar to Engage drivers and our tools can check this code.

NixOS [12] takes a radically different approach to package and configuration management than a typical Linux distribution. NixOS places every package and configuration in a unique location (determined during configuration) and ensures that they are immutable. This design forces NixOS policies to make all dependencies explicit. Puppet bring some of the advantages of NixOS to traditional operating systems and Linux distributions, but our paper shows that it doesn't provide the same guarantees of NixOS. Instead of proposing a radical, new architecture, we show that program verification techniques can be employed to provide strong guarantees for Puppet configurations.

Tucker and Krishnamurthi [29] argue that Racket's unit system could be adapted to build a better package manager. The benefits of their design are similar to the benefits of NixOS (discussed above).

Testing and verification of configurations. CLOUDMAKE is a cloud-based build system in use at Microsoft that has important features such as artifact caching, parallel builds, etc. CLOUDMAKE commands make all inputs and outputs explicit. Christakis, et al. [9] have a mechanized proof that CLOUDMAKE scripts are race-free, which justifies parallel builds. Our paper shows that it's not possible to prove such a theorem for all Puppet configurations. Instead, Rehearsal verifies that individual manifests are deterministic.

Hummer et al. [18] systematically test Chef configurations and find that several configurations are not idempotent. Their test-based approach cannot ensure complete coverage and can take several days. By contrast, we use static analysis to prove determinacy and idempotency, which would be more difficult to do for Chef as it is a Ruby-embedded DSL.

Although Puppet uses native package managers to implement package resources, Puppet doesn't leverage the rich information that packages provide, such as their direct dependencies and conflicts, which leads to the kind of errors described in section 2. It should be possible to leverage package metadata to build more useful verification tools, perhaps using the SAT-based encoding of Opium [28]. Unlike `apt-get`, Opium's algorithm for calculating installation/uninstallation is complete for a given distribution. The analogous problem for Puppet would be to calculate the installation profile for a resource, given a universe of resources, such as modules on Puppet Forge. To do so, one would need to calculate and verify dependencies. Rehearsal does the latter and could be augmented to do the former.

Rehearsal uses a straightforward model of the filesystem, partly because Puppet's model hides many platform-specific filesystem details for portability (*e.g.*, Puppet doesn't support hard links). Others have developed filesystem models that are much richer than ours (*e.g.*, [4, 22, 23]). The program logic of Gardner et al. [14] is particularly interesting because it enables modular reasoning about filesystem-manipulating programs. In contrast, the verification techniques in our paper are not modular because we support Puppet features that have global effects on the resource graph. If these features were ignored, a modular analysis would be attractive.

Cloud services such as Microsoft Azure contain large configurations with many components in various representations (*e.g.*YAML, XML, INI, etc.). ConfValley [17] unifies these configurations into a single representation and validates them with respect to user-written predicates about the configuration. The predicates may describe desirable properties for a particular cloud service configuration such as ensuring that a particular variable has the correct type or a certain file has the appropriate permissions. Rehearsal verifies two specific properties about the effects of a Puppet config-

uration on a machine, rather than properties of the configuration itself. We consider every possible input and execution path in order to prove or disprove idempotence and determinism. A ConfValley-style verification of Puppet would involve writing predicates about the structure of the resource graph, which should be straightforward to do with our tools.

Determinacy checkers. In the past few years, several tools have been developed that use static [6, 19, 31] and dynamic [8, 24] techniques to check that multi-threaded programs are deterministic. Rehearsal is a static determinacy checker for Puppet and leverages an SMT solver, thus is most closely related to Liquid Effects [19]. Liquid Effects establishes determinism by showing that concurrent effects are disjoint, but there are common examples of deterministic Puppet programs that do not have disjoint effects. Instead, Rehearsal has a commutativity check that accounts a pattern of false sharing that is common to Puppet (section 4.3). Rehearsal and Liquid Effects address determinism in two very different domains. Liquid Effects proves determinism for multi-threaded C programs with pointers, aliasing, and functions that are tackled in a modular way with types. In contrast, Puppet manifests have no aliasing, loops, or procedures. Since our problem is simpler, we are able to build a scalable, sound, and complete determinacy checker that requires no annotations by the programmer.

When Rehearsal reports that a Puppet manifest is *deterministic*, the manifest may still yield different outputs for different inputs. *i.e.*, Rehearsal only verifies that a manifest maps each input state to a single output state. In contrast, Andreasen and Møller [3], have developed techniques to infer that program expressions *determinate*, *i.e.*, that an expression produces the same value in all executions. They exploit determinate expressions to improve the precision of their JavaScript Type Analyzer. In contrast, Puppet expressions are always determinate, but Puppet manifests can be non-deterministic.

Alternate uses of configuration management. Finally, we note that configuration management is an overloaded term in the literature. This paper addresses an issue that arises in software configuration and deployment. However, the term configuration management is also used to refer to version-control systems (*e.g.*, CVS and Git) and to application configuration [26, 27], which is not the subject of this work.

8. Limitations

The primary limitation of this work is that Puppet manifests support embedded shell scripts (using the `exec` resource type). Shell scripts are often an anti-pattern, but they are indispensable for certain tasks. For example, they are often used to setup software that has not made its way into sanctioned software repositories. The main challenge with shell scripts is that they can have arbitrary effects on the filesystem, unlike the other resource-types that have a clearer semantics and lend themselves to formal models.

Another limitation of our work is that our analyses rely on models of system resources, which can be inaccurate. For example, to model packages, we need to know the files that a package creates. At present, we assume that packages only create the files returned by `apt-file` (on Debian) and `repoquery` (on Red Hat). However, many packages use "post-install scripts" to create additional files, which our approach will miss. Therefore, although our algorithms are sound and complete with respect to our model of system resources, our models have known limitations. A more precise alternative would be to actually install packages in a sandboxed environment and check what files get written to disk.

Finally, as suggested above, our analysis is platform-dependent. In fact, the choice of operating system determines how packages are modeled. Although Puppet has several platform-neutral features, it also exposes the platform name and version as program variables that a manifest can use to specialize for a particular platform. Rehearsal takes the platform name as a command-line flag and so a manifest can be re-verified for several platforms. However, it would be more useful to check that a manifest has similar effects on different platforms.

9. Conclusion

This paper presents Rehearsal, the first verification tool for Puppet, a popular system configuration language. Specifically, Rehearsal checks that `exec`-free Puppet manifests are deterministic and idempotent, which are both fundamental properties of correct Puppet manifests. To build Rehearsal, we developed a simple semantics for Puppet that we hope will be useful to other researchers. We believe that our approach to modeling Puppet will enable several other tools, *e.g.*, manifest repair and synthesis, and security auditing.

Acknowledgments

We thank the PLDI'16 reviewers, our shepherd Manu Sridharan, Daniel Barowy, Emery Berger, Shriram Krishnamurthi, Robert Powers, John Vilk, and Jean Yang for their thoughtful feedback and suggestions. We thank Joseph Collard and Nimish Gupta for their work on a preliminary version of Rehearsal. This work is supported by the U.S. National Science Foundation under grants CNS-1413985 and CCF-1408745 and by a Google Research Award.

References

[1] Paul Anderson. Towards a High-Level Machine Configuration System. *USENIX Large Installation System Administration Conference (LISA),* 1994.

[2] Paul Anderson and Herry Herry. A Formal Semantics for the SmartFrog Configuration Language. *Journal of Network and Systems Management,* 24(2):309–345, 2016.

[3] Esben Andreasen and Anders Møller. Determinacy in Static Analysis for jQuery. *ACM SIGPLAN Conference on Ob-*

ject Oriented Programming, Systems, Languages and Applications (OOPSLA), 2014.

[4] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. *International Conference on Formal Engineering Methods (ICFEM),* 2004.

[5] Augeas. Retrieved Apr 15 2016 from `http://augeas.net`.

[6] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. A Type and Effect System for Deterministic Parallel Java. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA),* 2009.

[7] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful Lenses for String Data. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL),* 2008.

[8] Jacob Burnim and Koushik Sen. Asserting and Checking Determinism for Multithreaded Programs. *Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE),* 2009.

[9] Maria Christakis, K. Rustan M. Leino, and Wolfram Schulte. Formalizing and Verifying a Modern Build Language. *International Symposium on Formal Methods (FM),* 2014.

[10] Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. A survey of system configuration tools. *USENIX Large Installation System Administration Conference (LISA),* 2010.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS),* 2008.

[12] Eelco Dolstra, Andreas Löh, and Nicholas Pierron. NixOS: A Purely Functional Linux Distribution. *Journal of Functional Programming,* 20(5–6):577–615, 2010.

[13] Jeffery Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: A Deployment Management System. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* 2012.

[14] Philippa Gardner, Gian Ntzik, and Adam Wright. Local Reasoning about POSIX File Systems. *European Symposium on Programming (ESOP),* 2014.

[15] Arjun Guha, Rian Shambaugh, and Aaron Weiss. Rehearsal. Retrieved Apr 15, 2016 from `http://plasma.cs.umass.edu/rehearsal`.

[16] Bent Hagemark and Kenneth Zadeck. Site: A Language and System for Configuring Many Computers as One Computing Site. *USENIX Large Installation System Administration Conference (LISA),* 1989.

[17] Peng Huang, William J. Bolosky, and Abhishek Singh Yuanyuan Zhou. ConfValley: A Systematic Configuation Validation Framework for Cloud Services. *European Conference on Computer Systems (EuroSys),* 2015.

[18] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing Idempotence and Convergence for Infrastructure as Code. *ACM/IFIP/USENIX International Middleware Conference,* 2013.

[19] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic Parallelism via Liquid Effects. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* 2012.

[20] Puppet Labs. Puppet Features: Idempotency. Retrieved Apr 15, 2016 from `http://docs.puppetlabs.com/guides/introduction.html#idempotency`.

[21] John McCarthy. Towards a Mathematical Science of Computation. *IFIP Congress,* 1962.

[22] Carroll Morgan and Bernard Sufrin. Specification of the UNIX Filing System. *IEEE Transactions on Software Engineering (TSE),* 10(2):128–142, 1984.

[23] Tom Ridge, David Sheets, Thomas Tuerk, Anil Madhavapeddy, Andrea Giugliano, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. *ACM Symposium on Operating Systems Principles (SOSP),* 2015.

[24] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. *European Symposium on Programming (ESOP),* 2009.

[25] Eric Shamow. Inside Puppet: About Determinism. Retrieved Apr 15, 2016 from `http://puppetlabs.com/blog/inside-puppet-about-determinism`.

[26] Alex Sherman, Philip A. Lisiecki, Andy Berkheimer, and Joel Wein. ACMS: The Akamai Configuration Management System. *USENIX Symposium on Networked System Design and Implementation (NSDI),* 2005.

[27] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chandler, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. *ACM Symposium on Operating Systems Principles (SOSP),* 2015.

[28] Chris Tucker, David Shuffleton, Ranjit Jhala, and Sorin Lerner. OPIUM: Optimal Package Install/Uninstall Manager. *International Conference on Software Engineering (ICSE),* 2007.

[29] David B. Tucker and Shriram Krishnamurthi. Programming Languages for Software Configuration. *International Workshop on Software Configuration Management (SCM),* 2001.

[30] Ubuntu. Details of package `golang-go` in trusty. Retrieved Apr 15, 2016 from `http://packages.ubuntu.com/trusty/devel/golang-go`.

[31] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic Verification of Determinism for Structured Parallel Programs. *International Static Analysis Symposium (SAS),* 2010.