Rehearsal: A Configuration Verification Tool for Puppet

Rian Shambaugh
University of Massachusetts Amherst
rian@cs.umass.edu

Aaron Weiss
University of Massachusetts Amherst
aaronweiss@cs.umass.edu

Arjun Guha
University of Massachusetts Amherst
arjun@cs.umass.edu

Abstract

Large-scale modern data centers and cloud computing have turned system configuration into a challenging and urgent problem. Several widely-publicized outages have been blamed not on software bugs, but on software configuration issues. To cope, thousands of organizations now use system configuration languages to manage their computing infrastructure. Of these, Puppet is the most widely used with over $18,000$ paying customers and many more open-source users. The heart of Puppet is a declarative 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.

This paper presents Rehearsal, a verification tool for Puppet configurations. We first illustrate the kinds of errors that occur in Puppet with several examples. We show that the key issue is that configurations can easily be non-deterministic, thus Rehearsal implements a sound, complete, and scalable determinacy analysis for Puppet. To develop it, we (1) present a formal model for Puppet configurations, (2) use two different program analyses to prune our models to a tractable size, and (3) verify manifest determinacy by framing determinacy-checking as an SMT problem. Rehearsal then leverages the determinacy analysis to check other important properties, such as idempotency. Finally, we apply Rehearsal to several real-world Puppet configurations.

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.

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 [9] uses a lazy, functional language to specify packages and system configurations; Augeas [3] uses lenses [5] to update configuration files on mainstream operating systems; and Engage [10] provides a declarative DSL that tackles issues such as inter-machine dependencies.

In the past few years, several system configuration languages have been developed in industry. Puppet, Chef, and Ansible (recently acquired by RedHat) are three prominent examples. Puppet, which is the most popular of these languages, has over $18,000$ paying customers (with many more that use the open-source version) including half of the Fortune 100 companies [17]. This paper focuses on Puppet, 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 `apt`, `rpm`, etc. under the hood. Second, these languages provide abstractions for managing several kinds of resources, such as packages, configuration files, user accounts, 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]. Instead, they provide simple and expressive DSLs that encourage 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 Pup-

pet manifest states that the `vim` package should be installed, the user account `carol` should exist, and she should have a `.vimrc` file in her home directory:

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

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. An often-cited property of Puppet is that manifests are deterministic. In practice, determinism means that a system administrator knows that a manifest will have the same effects in testing and production. Similarly, if one manifest is applied to several machines, which is common in large deployments, determinism ensures 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 only 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 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 may 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 [6, 19]. However, a purely dynamic approach would be weaker here as it could only identify a problem when two replicas have diverged, whereas a static determinacy analysis can ensure 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. 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 languages techniques for Puppet or any other system configuration language of its kind. We first present a core fragment of Puppet and several small examples that illustrate the problems with the language (section 2). We develop a formal semantics for Puppet that models manifests as programs in a simple, nondeterministic imperative programming language for filesystem manipulation 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 two different program analyses to prune the size of models. The first analysis dramatically reduces the number of paths that the determinism-checker needs to reason about by finding and eliminating side-effects that are unobserved by the rest of the program. The second 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 logical formulas in an SMT solver.

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) and future directions (section 9).

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.

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

Types	*rtype* ::=	`file` | `package` | $\cdots$	
Strings	*str* ::=	"$\cdots$`$x`$\cdots$`$y`$\cdots$"	
Identifiers	x ::=	`$x` \| `$y` | $\cdots$	
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 Core Puppet

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

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

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

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 the core subset of Puppet features that are sufficient for most Puppet tasks, 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`]`. 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,

```
file {"/etc/apache2/sites-enabled/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

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 (figure 2).

2.2 Common Puppet Problems

Since Puppet affords programmers a great deal of freedom, there are a number of common problems that occur in manifests.

Non-deterministic error. 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. In fact, if the package had been previously *removed* or *purged*, more subtle errors can occur.

Overconstrained dependencies. Consider a strawman solution to the non-determinism problem: we could add false dependencies so that all resources are totally ordered. Unfor-

tunately, 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.[2] 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. [3] This heavy-handed approach to determinism sacrifices composability.

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, the Go compiler depends on Perl, so this state is not realizable, but Puppet cannot detect this problem. Puppet 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 correct state.

Non-idempotence. A desired property of Puppet manifests is that they are idempotent: applying a manifest twice should be the same as applying it once. However, Puppet does not enforce this property making it easy to produce manifests that are not idempotent. For example, we can make the nondeterministic manifest in figure 3c deterministic by requiring Perl to be removed 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 because it specifies an inconsistent state.

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 breaks idempotence.

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

[3] 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.

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 into nodes and dependencies into edges. To do this, we employ a number of passes to reduce manifests into a more concrete form.

In particular, Puppet has several constructs to allow succinct descriptions of dependencies and abstractions over primitive resources. In order for our resource graph to be correct, 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" }
```

It breaks modularity and makes 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. [We don't parse resource collectors. –Rian]

Our compiler tackles all the details of these and several other features that we don't belabor here. This work is necessary to apply our toolchain to unmodified third-party code.

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 holds files ($\mathsf{file?}(p)$), directories ($\mathsf{dir?}(p)$), empty directories ($\mathsf{emptydir?}(p)$), or nothing ($\mathsf{none?}(p)$). Predicates also include the usual boolean connectives ($\vee$, $\wedge$, and $\neg$).

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, FS, 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 capture the semantics of each resource type. 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. In addition, resources (thus programs) must be idempotent, which further complicates the definition of $\mathcal{C}$. 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 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, and as such 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.[4] 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 $\mathsf{creat}(p, str)$ 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 appears unlikely to arise in practice, and its occurrence may be indicative of another mistake, *e.g.*, perhaps the author meant to overwrite p with some other contents.

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

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

Syntax

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

[FS does not have emptydir? (Not important?) –Rian]

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.

$$[\![G]\!] \in \sigma \to 2^{\sigma + \mathsf{err}}$$
$$[\![G]\!]\,\sigma \triangleq \{[\![\mathcal{C}(L(v_1)); \cdots; \mathcal{C}(L(v_n))]\!]\,\sigma \mid \langle v_1 \cdots v_n \rangle \in \mathsf{perms}(G)\}$$
$$\text{where } G = (V, E, L)$$

Figure 6: Semantics of resource graphs

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 would 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 tools described 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 and would raise an error on certain inputs. 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 the 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. Using an abstract interpretation, we determine if p is the same file or a directory over all input filesytems. If so, we carefully prune writes to p.
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 shar-

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

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

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

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

$$
\begin{array}{ll}
\Phi_G(G) & \in \Sigma \to \Sigma \\
\Phi_G((\emptyset, E))\Sigma & \triangleq \Sigma \\
\Phi_G(G)\Sigma & \triangleq \\
\multicolumn{2}{c}{\Phi_G(G - e_1)(\Phi(e_1)\Sigma) \bigvee \cdots \bigvee \Phi_G(G - e_n)(\Phi(e_n)\Sigma)} \\
\multicolumn{2}{l}{\text{where } \{e_1 \cdots e_n\} = \{e \in G \mid \mathit{inDegree}(e) = 0\}} \\
\multicolumn{2}{l}{\quad \Sigma_1 \bigvee \Sigma_2 = \mathsf{if}\ (b)\ \Sigma_1\ \mathsf{else}\ \Sigma_2 \text{ where } b \text{ is a fresh variable}}
\end{array}
$$

Figure 7: Encoding of programs to SMT formulas.

ing 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 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

$\Phi(e)$ produces a formula that encodes that semantics of the expression e (figure 7). The formula maps an input logical state (Σ) to an output logical state. Unlike the concrete evaluator, which either produces an output state or signals an error, the logical state is a record of two components. (1) $\Sigma.\mathsf{ok}$ is a formula that indicates if the current state is not the error state. (2) $\Sigma.\mathsf{fs}$ is a map that describes the state of the file system and is only meaningful if $\Sigma.\mathsf{ok}$ is true. We could employ McCarthy's theory of arrays [15] to encode

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

Figure 8: Calculating the domain bound for FS expressions

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. The formula non-deterministically chooses each expression e on the fringe of G, evaluates it with $\Phi(e)$ and then recurs.

To prove this encoding sound and complete, we need to relate concrete filesystems to logical filesystems. 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 $\mathsf{mkdir}(\texttt{/a/b})$ reads `/a` and writes `/a/b`.

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

1. $\Phi_G(\langle \mathit{ok} \texttt{=} \mathsf{true}, \mathit{fs} \texttt{=} \sigma \rangle, e) \vdash \langle \mathit{ok} \texttt{=} \mathsf{true}, \mathit{fs} \texttt{=} \sigma' \rangle$ *iff* $\sigma' \in [\![e]\!]\,\sigma$
2. $\Phi_G(\langle \mathit{ok} \texttt{=} \mathsf{true}, \mathit{fs} \texttt{=} \sigma \rangle, e) \vdash \langle \mathit{ok} \texttt{=} \mathsf{false}, \mathit{fs} \texttt{=} \sigma' \rangle$ *iff* $\mathsf{err} \in [\![e]\!]\,\sigma$

4.2 Checking Non-determinism

With resource graphs encoded as formulas, it should be straightforward to use a theorem prover to check non-determinism (though we have yet to address scalability issues). Recall that each invocation of $\Phi_G(G)$ allocates fresh boolean variables that can be interpreted as the order in which resources are chosen. Therefore, a resource graph should be non-deterministic, if and only if the following formula is satisfiable:

$$\exists \Sigma. \Phi_G(G)\Sigma \neq \Phi_G(G)\Sigma$$

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

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 the directory is empty and the other only checks that it is a directory:

$$\text{if (emptydir?(/a)) id else err}$$
$$\not\equiv \text{if (dir?(/a)) id else err}$$

Any input filesystem that demonstrates the inequality must have a file (or directory) within /a. However, if we construct 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 $\Phi(\langle \mathit{ok} = \mathsf{true}, \mathit{fs} = \hat{\sigma}' \rangle, e_1) \neq \Phi(\langle \mathit{ok} = \mathsf{true}, \mathit{fs} = \hat{\sigma}' \rangle, e_1)$.

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

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

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

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

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

Theorem 1. *(Nondeterminism) G is non-deterministic, if and only if there exists Σ, such that $dom(G) \subseteq dom(\Sigma)$ and $\Phi_G(G)\Sigma \neq \Phi_G(G)\Sigma$ is satisfiable.*

4.3 Pruning Definitive Writes

Even small manifests can affect hundreds of files, which makes the representation of individual states (Σ) intractably large. Large packages are the most obvious culprits. For example, on Ubuntu 14.04, the `git` package has over 500 files, but it is unlikely that the rest of the manifest will affect them. Similarly, figure 3a showed a manifest that installs the Apache web server package and overwrites one of its configuration files, but it does not read or write any of the other 200 files that the package creates. In general, it is safe to ignore side-effects that are not observable by other resources in a manifest. This section formalizes this intuition to shrink the domain of logical states.

Detecting Definitive Writes. Since a non-determinism check is essentially a conjunction of equivalence checks, we first consider the simpler problem of shrinking 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 eliminate 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. Therefore, if both expressions have leave p in all inputs (or error), we can eliminate writes to p.

$$\begin{aligned}
\text{Abstract Values}\quad \hat{v} &::= \bot \mid \top \mid \mathsf{dir} \mid \mathsf{file}(\mathit{str}) \mid \mathsf{dne} \\
\text{Abstract State}\quad \hat{\sigma} &::= \langle p_1 = \hat{v}_1 \cdots p_k = \hat{v}_k \rangle \\
&\bot \sqsubset \mathsf{dir}, \mathsf{file}(\mathit{str}), \mathsf{dne} \sqsubset \top \\
\widehat{[\![e]\!]} &\in \hat{\sigma} \to \hat{\sigma} \\
\widehat{[\![\mathsf{mkdir}(p)]\!]}\hat{\sigma} &= \hat{\sigma}[p := \mathsf{dir}] \\
\widehat{[\![\mathsf{creat}(p, \mathit{str})]\!]}\hat{\sigma} &= \hat{\sigma}[p := \mathsf{file}(\mathit{str})] \\
&\cdots
\end{aligned}$$

Figure 9: Abstract interpretation to detect definite writes.

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

Figure 10: Pruning writes to expressions

We detect these *definitive writes* using the abstract interpretation sketched in figure 9, 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 $\hat{\sigma}(p) = \mathsf{file}(\mathit{str})$, 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.

Pruning Writes. If the abstract interpretation determines that e_1 and e_2 have the definitely write to p in the same way, we can prune those writes from expressions. To do so safely, consider the following inequality:

$$\mathsf{mkdir}(\texttt{/a}); \mathsf{creat}(\texttt{/a/b}, \mathit{str}) \not\equiv \mathsf{creat}(\texttt{/a/b}, \mathit{str})$$

We should be able to eliminate the write to /a/b. But, a naive approach that simply replaces $\mathsf{creat}(\texttt{/a/b}, \mathit{str})$ with id would change the inequality. The insight is that to create the file we also reading from from /a and /a/b and these reads have to be preserved:

$$\text{if } (\mathsf{dir?}(\texttt{/a}) \wedge \mathsf{none?}(\texttt{/a/b})) \text{ id else err}$$

$$\begin{array}{l}\text{Abstract Values } \tilde{v} ::= \bot \mid R \mid W \mid D\\ \text{Abstract State } \tilde{\sigma} ::= \langle p_1 = \hat{v}_1 \cdots p_k = \hat{v}_k \rangle\\ \bot \sqsubset R, D \sqsubset W\end{array}$$

$$[\![e]\!]_C \in \tilde{\sigma} \to \tilde{\sigma}$$

$$[\![\mathsf{if}\ (\neg\mathsf{dir?}(p/str))\ \mathsf{mkdir}(p/str)]\!]_C\ \tilde{\sigma} \triangleq \begin{cases}\tilde{\sigma}[p/str := D] & \hat{\sigma}(p/str) \sqsubseteq D \\ & \text{and } \tilde{\sigma}(p) = D \\ \tilde{\sigma}[p/str := W] & \text{otherwise}\end{cases}$$

$$[\![\mathsf{mkdir}(p)]\!]_C\ \hat{\sigma} \triangleq \tilde{\sigma}[p := W]$$

$$[\![\mathsf{creat}(p, str)]\!]_C\ \hat{\sigma} \triangleq \tilde{\sigma}[p := W]$$

$$[\![e_1; e_2]\!]_C\ \tilde{\sigma} \triangleq [\![e_2]\!]_C\ ([\![e_1]\!]_C\ \tilde{\sigma})$$

Figure 11: Checking commutativity when expressions create overlapping directory trees

The pruning function, $\mathit{prune}(p, e)$, eliminates writes to p by preserving reads in this manner (figure 10). 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 4. *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* $\mathit{prune}(p, e_1) \equiv \mathit{prune}(p, e_2)$.

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.

4.4 Commutativity and Directory Creation

Modeling all valid permutations of resources can be intractably large, so the natural solution is to employ partial-order reduction. The key to partial-order reduction is to have 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 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 11). 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:

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

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 abstract interpretation to check that expressions commute, even if they create overlapping directory trees.

Lemma 5. *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 $e_1; e_2 \equiv e_2; e_1$.

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.

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, 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. The check can be optimized further by pruning writes (section 4.3).

We emphasize that these checks are efficient because they do not have to consider all permutations of resources.

6. Evaluation

Rehearsal is implemented in Scala and uses the Z3 Theorem Prover [8] 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 running in virtual machines. 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 VMs running and it is easy to add support for other operating systems.

Benchmark suite. We benchmark Rehearsal 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.

Rehearsal effectiveness. We have proved that Rehearsal will always find and report nondeterminism errors for the fragment of Puppet that we can parse (theorem 1) and that our optimizations preserve determinism (lemmas 4 and 5).

Rehearsal scalability. Without the commutativity check, the majority of benchmarks do not complete within ten minutes and so we don't report those running times. As a result, all of our results have the check enabled. Figure 12b shows the running time of the determinacy checker with and without pruning. In the figure, the non-deterministic manifests are marked *-nondet*. The figure shows that pruning has a negligible effect on small manifests, but a larger impact on manifests that manipulate several files. In fact, the `irc` and `hosting` benchmarks timed out after ten minutes without pruning but completed in less than thirty seconds with it.

Figure 12a shows the number of files in each manifest, before and after pruning. As expected, the number of pruned paths corresponds to amount of speedup in figure 12b.

Figure 12c 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 takes much longer. Therefore, checking idempotence with determinism has a negligible overhead.

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

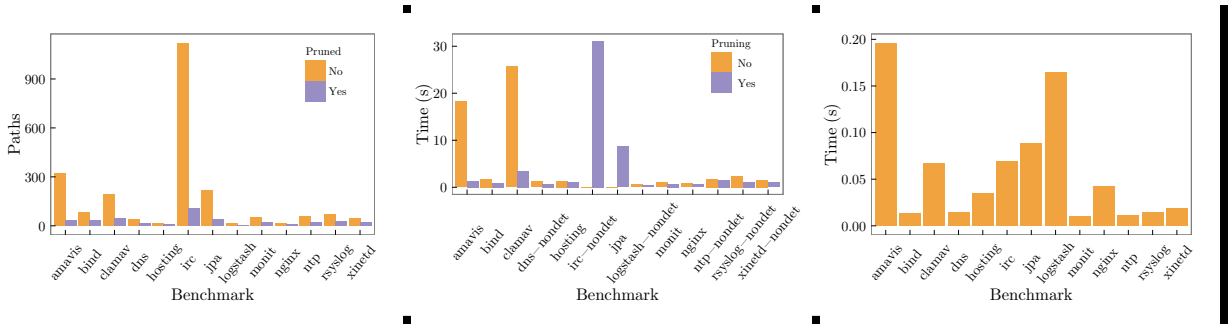
Other system configuration languages. Several system configuration languages have been developed over decades of research. To the best of our knowledge, the kind of verification tools we have developed for Puppet have not been developed for the languages listed below. Instead, we highlight how they differ from Puppet and consider what it would take to adapt our approach to these other languages.

Hagemark and Zadeck's Site tool [12] 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.

Engage [10] 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 lifecycle 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 itself performs operations similar to Engage drivers and our tools can check this code.

NixOS [9] 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



(a) Paths per state. (b) Checking determinism. (c) Checking idempotence.

Figure 12: Benchmarks

verification techniques can be employed to provide strong guarantees for Puppet configurations.

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. [7] 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. [13] 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 [20]. 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.*, [2, 16, 18]). The program logic of Gardner et al. [11] 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.

Determinacy checkers. In the past few years, several tools have been developed that use static [4, 14, 21] and dynamic [6, 19] techniques to check that multithreaded programs are deterministic. Rehearsal is a static determinacy checker for Puppet and leverages an SMT solver and is most closely related to Liquid Effects [14]. 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 4.4). Rehearsal and Liquid Effects address determinism in two very different domains. Liquid Effects proves determinism for multithreaded 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.

8. Limitations

The primary limitation of work is that manifests can support embedded shell scripts using the `exec` resource type. Shell scripts are often an anti-pattern, but can be indispensable for certain tasks. In practice, they appear most often for installing software that has not made its way into official software repositories, but they have other uses as well. The problem arising from `exec` stems from the fact that shell scripts can run arbitrary programs making filesystem effects hard to predict. To deal with this, we are actively exploring the idea of a hybrid static-dynamic analysis that infers effect models from shell scripts in `exec` resources.

9. Conclusion

This paper presents Rehearsal, the first verification tool for Puppet, a popular system configuration language. Rehearsal

includes a simple semantics for Puppet called FS that we hope will be useful to other researchers. Rehearsal itself features both a determinism and an idempotency checker for Puppet, both fundamental tools for creating correct manifests. In addition, we believe that our approach to modeling Puppet will enable several other tools, *e.g.*, manifest repair and synthesis, and security auditing.

References

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

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

[3] Augeas. `augeas.net`.

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

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

[12] 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.

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

[14] 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.

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

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

[17] PuppetConf 2014 Press Release. `puppetlabs.com/about/press-releases/puppetconf-2014-it-automation-event-year-opens-record-atte`

[18] 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.

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

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

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