Tortoise: Interactive System Configuration Repair

Aaron Weiss
Northeastern University
Boston, MA, USA 02115
weiss@ccs.neu.edu

Arjun Guha
University of Massachusetts, Amherst
Amherst, MA, USA 01003
{arjun, brun}@cs.umass.edu

Yuriy Brun

***Abstract*—System configuration languages provide powerful abstractions that simplify managing large-scale, networked systems. Thousands of organizations now use configuration languages, such as Puppet. However, specifications written in configuration languages can have bugs and the shell remains the simplest way to debug a misconfigured system. Unfortunately, it is unsafe to use the shell to fix problems when a system configuration language is in use: a fix applied from the shell may cause the system to *drift* from the state specified by the configuration language. Thus, despite their advantages, configuration languages force system administrators to give up the simplicity and familiarity of the shell.**

This paper presents a synthesis-based technique that allows administrators to use configuration languages and the shell in harmony. Administrators can fix errors using the shell and the technique automatically repairs the higher-level specification written in the configuration language. The approach (1) produces repairs that are *consistent* with the fix made using the shell; (2) produces repairs that are *maintainable* by minimizing edits made to the original specification; (3) ranks and presents multiple repairs when relevant; and (4) supports all shells the administrator may wish to use. We implement our technique for Puppet, a widely used system configuration language, and evaluate it on a suite of benchmarks under 42 repair scenarios. The top-ranked repair is selected by humans 76% of the time and the human-equivalent repair is ranked 1.31 on average.

I. Introduction

Modern computing systems are large, complex, and need to be reconfigured frequently to address changing threats and requirements. The job of a *system administrator* is to perform these tasks. For example, if a web server is under attack, she may reconfigure a firewall; if a new security patch is available, she may deploy it; if an intrusion detection system is needed, she may set it up and ensure it does not interfere with normal operations. System administration is a difficult task and the majority of large organizations use *system configuration languages* to make the job easier. For example, Puppet [42] is deployed at over 33,000 companies, Chef [8] has over 40 million downloads [59], and Ansible [45] was quickly bought by Red Hat a few years after its release.

Unfortunately, updating system configurations is a surprisingly difficult task and several recent, high-profile computing failures have been caused by configuration updates gone wrong. For example, in 2016, some Google App Engine customers suffered a two-hour service outage due to a configuration error that was triggered during an application server update [18]. In 2015, the New York Stock Exchange suffered an outage that halted trading for four hours because a software update went awry [54]. In 2010, Facebook suffered a 2.5 hour outage that was again caused by a faulty configuration update [16]. In that incident, a system for verifying system configurations actually exacerbated the problem.

This paper focuses on Puppet, the most widely deployed system configuration language [58], but our work generalizes to other configuration languages (see Section VIII). Puppet configurations (known as *manifests*) have the following key features. First, manifests are declarative, parameterizable, and support modular composition. For example, Puppet has an online repository of nearly 5,000 community-supported manifests. Second, manifests make systems reproducible. For example, if a new web server is needed, a system administrator can quickly set it up if she already has a web server manifest. Finally, manifests support centralized management. Puppet uses a client-server model, where all manifests are maintained on a centralized server and propagated to client machines.

Manifests may have bugs and even bug-free manifests need to be updated to address changing requirements. However, there are many cases where manifests make changes harder to apply than they should be. A small change, such as creating a new user, adding a firewall rule, or starting a service, is easy to perform with the command-line shell, using commands such as `useradd`, `iptables`, and `service` that are familiar to administrators. The shell also lets the administrator explore the state of the system and, unlike a manifest, typically provides immediate feedback when the administrator makes a mistake. By contrast, editing a manifest is much harder. First, in a large manifest that uses high-level, user-defined abstractions, it can be difficult to find where and how an update should be made. Second, the only way to test an update is to redeploy it, which can take anywhere from minutes to hours. Finally, an update may have unintended effects, especially if the update is in a function that is called from multiple contexts.

The natural solution to this problem is to use a manifest and the shell simultaneously. For example, a manifest could specify the state of the machine while small updates are made using the shell. Unfortunately, it is not safe to make changes from the shell when a manifest is in use, because the actual state of the system will no longer match the state specified in the manifest — a phenomenon known as *configuration drift*.

Our approach. We present a new approach to repairing system configurations, called *imperative configuration repair*, which bridges the gap between the shell and system configuration lan-

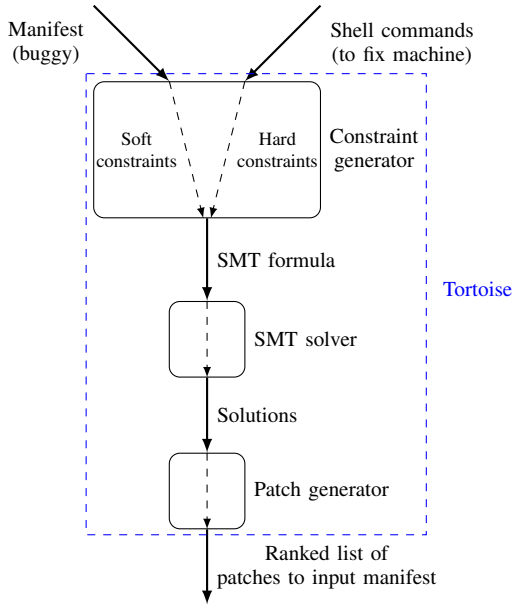


Fig. 1: The user deploys a buggy Puppet manifest and uses the shell to fix the machine state. Tortoise automatically produces a ranked list of manifest repairs that preserve the changes made in the shell.

guages. Imperative configuration repair is a program synthesis-based technique that allows a manifest to be automatically repaired given a sequence of shell commands to guide the desired system state. Therefore, instead of running the risk of configuration drift, we allow system administrators to use a familiar shell to fix errors with the confidence that the manifest will be updated automatically. Our approach has several important properties:

1) Our repair procedure is *consistent*: the synthesized repair is guaranteed to preserve all changes made in the shell (that are within the scope of the system model), so there is no configuration drift.
2) Our repair procedure preserves the structure and abstractions in the manifest. Therefore, we synthesize *maintainable* patches.
3) When multiple consistent repairs exist, we rank them and present several alternatives in a comprehensible manner.
4) Our approach places no restrictions on how the shell is used and works with *all* existing shells.

We have implemented our approach in a tool called Tortoise, which is outlined in Figure 1 and works as follows:

1) Suppose the administrator needs to update a machine that is managed by a Puppet manifest. Tortoise allows her to directly update the machine using ordinary shell commands. Behind the scenes, Tortoise uses ptrace to record all system calls and file system changes made from the shell.
2) When she is done, Tortoise builds a model of the original manifest and the updates from the shell in a language called ΔP. The model treats the shell updates as hard constraints and the original manifest as soft constraints.
3) Tortoise translates the ΔP model into logical formulae for an SMT solver (specifically, Z3-str [63]). These formulae produce $\exists\forall$-queries, which we solve using CEGIS [51].
4) Tortoise interprets each solution produced by the SMT solver as a patch to the Puppet manifest, ranks the patches in an intuitive way, and presents the most likely patches to the user.
5) Finally, the user selects the patch she wishes to apply. Although different patches have different effects, Tortoise guarantees that all patches are *consistent* and preserve the changes that the user made from the shell in step 1.

```
1 package{"apache2": ensure => present }
2
3 service{"apache2": ensure => running }
4
5 file{"/etc/apache2/sites-enabled/piedpiper.conf":
6   content => "<VirtualHost www.piedpiper.com:80>
7   DocumentRoot /var/sites/piedpiper
8   </VirtualHost>" }
9
10 file{"/var/sites/piedpiper:
11   ensure => "directory",
12   source => "puppet://sites/piedpiper",
13   owner => root,
14   mode => 0700,
15   recurse => "remote" }
16
17 Package["apache2"]
18   -> Service["apache2"]
19 Package["apache2"]
20   -> File["/etc/apache2/sites-enabled/piedpiper.conf"]
21 File["/etc/apache2/sites-enabled/piedpiper.conf"]
22   ~> Service["apache2"]
```

Fig. 2: Managing a single website.

We evaluate Tortoise on an existing suite of Puppet manifests [47]. We identify a total of 42 scenarios where the manifests would need repair. However, instead of repairing the manifests directly, as a system administrator would normally have to do, we directly update the system using the shell and use Tortoise to synthesize the repair to the manifest. The highest-rank repair Tortoise synthesized was the correct repair 76% of the time, and the correct repair was in the top five Tortoise-synthesized repairs 100% of the time. Tortoise and our benchmarks are available at http://plasma.cs.umass.edu/tortoise.

The rest of this paper is organized as follows. Section II, motivates our approach with an example. Section III details Tortoise's expressiveness. Section IV presents the ΔP language and describes the translation from Puppet manifests and system call traces to ΔP. Section V describes how Tortoise converts ΔP constraints to logical formulae for an SMT solver, how models returned by the solver are interpreted as repairs, and how these repairs are ranked. Section VI evaluates Tortoise and Section VII summarizes our work's limitations. Finally, Section VIII discusses related work and Section IX concludes.

II. A Configuration Repair Scenario

To motivate the need for imperative configuration repair, we first present a simple manifest that sets up a web server (Figure 2). This manifest has a bug and we describe how a system administrator would find and fix the bug using the shell, and then how Tortoise automatically synthesizes the repair to the manifest by observing the administrator's shell commands.

A manifest is a declarative specification of the expected system configuration. A manifest specifies a set of resources, their state, and their interdependencies. Each resource consists

of (1) a *type*, for example **package**, **file**, or **service**; (2) a *title*, interpreted based on the type, for example, the title of a **file** is the path of the file, whereas the title of a **package** is the name of the package; and (3) a dictionary of *attributes* to configure the resource, such as specifying that a **package** should be present or absent, or that a **file** refers to a directory.

Figure 2 shows a manifest with four resources: (1) the `apache2` package, which must be present on the system; (2) the `apache2` service, which must be running; (3) the file `piedpiper.conf` that sets up www.piedpiper.com; and (4) the directory containing the site's files, which are copied from the puppet master server (indicated by the `puppet://` prefix). The manifest also specifies three dependencies: (1) the `apache2` package must be installed before the service; (2) the `apache2` package must be installed before `piedpiper.conf` is created; and (3) the `apache2` service is "notified" (restarted) when `piedpiper.conf` changes.

This manifest will successfully deploy, but that does not guarantee that the resulting system configuration is correct. In fact, the manifest in Figure 2 has a bug: if we visit www.piedpiper.com, we will get an HTTP 403 Forbidden error.

Repairing the configuration in the shell. When the system administrator discovers this problem, she considers that a 403 error may either indicate that the client does not have permission to access the requested resource or that the server is misconfigured and cannot access a needed file. For security reasons, the server does not send a detailed error message to the client. Therefore, the only way to debug the problem is to inspect the web server log. The administrator runs the following command:

```
tail /var/log/apache2/error.log
```

The log contains the line "`(13) permission denied`", which indicates that the permissions on the site directory may be incorrect. To investigate this, the administrator now runs the following command:

```
stat /var/sites/piedpiper
```

The result of this command returns the directory's owner, which is `root`, and its permissions, which is `0700`. This indicates that the directory is not readable by others, including website visitors. The fix is to make the directory readable by all:

```
chmod 755 /var/sites/piedpiper
```

Now, the administrator can refresh the page and verify that the error is fixed.

Unfortunately, the state of the machine has now *drifted* from state specified in the manifest. Using this manifest to configure a second web server will lead to the same problem and the administrator will have to manually apply the same fix. Worse, Puppet itself will undo the fix on this server! Any changes to the manifest, e.g., to install more software, will cause Puppet to re-apply all resources and revert the permissions back to their original broken state. When using Puppet, it is safe to perform read-only actions to explore the machine state using the shell, e.g., to view logs or inspect permissions. However, it is unsafe to use the shell to perform updates.

```
1 package{"apache2": ensure => present }
2 service{"apache2": ensure => running }
3
4 define website($title,$root) {
5   file{"/etc/apache2/sites-enabled/$title.conf":
6     content => "
7     <VirtualHost $title:80>
8     DocumentRoot /var/sites/$root
9     </VirtualHost>" }
10
11  file{"/var/sites/$root":
12    ensure => "directory",
13    source => "puppet://sites/$root",
14    owner => root,
15    mode => 0700,
16    recurse => "remote" } }
17
18 website{"www.piedpiper.com": root => "piedpiper" }
19 website{"piperchat.com": root => "piperchat" }
```

Fig. 3: A website abstraction. For exposition, the inter-resource dependencies are elided.

Tortoise solution. Tortoise allows the administrator to fix the bug using the shell without the risk of configuration drift. To do so, Tortoise first translates the manifest into a program, written in ΔP, that models all the effects that the manifest has on the file system. The model is lengthy, but only a small fragment is relevant to this repair:

```
1 rlet title = "/etc/sites/piedpiper" from str;
2 rlet mode = 0700 from int(9);
3 ...
4 chmod(title,mode);
```

The code uses the `chmod` command to set the mode. However, instead of using constants for the mode and the directory name, the command refers to the *repairable variables* on the first two lines. Each repairable variable specifies an original value and a *repair space* of alternate values. For example, on line 2, the original mode is `0700` but can be repaired to any 9-bit integer, if needed. After producing this model, Tortoise translates the observed system calls issued by the shell into an assertion, also expressed in ΔP. In this case, the assertion is as follows:

```
5 assert(mode("/var/sites/piedpiper") == 0755);
```

The only way for this assertion to hold is if the value of `mode` is repaired to `0755` and the value of `title` is left unchanged. This repair to the ΔP model corresponds to changing line 14 of Figure 3 to `mode => 0755`. This is exactly the change the administrator would have made herself. However, in more sophisticated manifests, there may be several alternative repairs that Tortoise ranks and presents to the user as patches.

User-defined abstractions in Puppet. We now consider a more sophisticated example manifest that uses Puppet's abstractions to manage a second website, piperchat.com. The naïve approach is to duplicate and tweak the configuration for piedpiper.com. But, a better approach is to create a custom `website` type (known as a *defined type* in Puppet) for managing a website that is parameterized by the domain name and site directory. This custom type allows websites to be configured with just one line each (lines 18 and 19 in Figure 3).

Suppose the system administrator built this abstraction before fixing the permissions problem, thus both websites produce the same error. She discovers the error by visiting piedpiper.com, as she did earlier, witnesses the 403 error, and then uses the shell to diagnose and fix the problem as before, using the `chmod` command. Without Tortoise, there are two problems:

```
1 define website($title,$root,$https = false) {
2   if ($https) { ... lots of configuration ... }
3   else { ... same as before ... } }
```

Fig. 4: Optional support for HTTPS.

(1) the configuration has drifted from the manifest as before and (2) the other website remains broken.

Tortoise solution. There are two ways to correct the manifest to be consistent with the system state:

1) Change line 15 to be consistent with the shell and affect both websites:

```
mode => 0755
```

2) Change line 15 to be exactly consistent with the shell and leave the other website unaffected:

```
$title == "piperchat" ? 0755 : 0700
```

There are situations where either type of repair may be desired. The first repair generalizes a change made to one instance to all instances of the same type, whereas the second kind of repair is necessary to specify special-case behavior. In general, Tortoise cannot know which kind of repair is desired, so it presents both repairs to the administrator. Since special-cases are the exception, Tortoise ranks the repairs in the order shown above. Notice that both repairs update a line of code that is within a defined type, so Tortoise is not limited to working with Puppet's built-in abstractions.

Reusing abstractions. Tortoise can also repair resources that instantiate user-defined types, as the next example shows. Suppose the administrator wants to start using HTTPS to secure websites. Modern web browsers block HTTPS servers from loading JavaScript from unsecured domains. Therefore, sites need to be carefully upgraded to ensure that all third-party code is served over HTTPS too. For this reason, it makes sense to migrate one website at a time.

The process of upgrading www.piedpiper.com to HTTPS involves specifying the certificate, private chain, ciphers, and several other details that are difficult to get right the first time. Moreover, there is a risk that a faulty edit to the `website` type will inadvertently break the other website too. Therefore, it is safer to directly edit the Apache configuration file for www.piedpiper.com instead. Apache has a command-line tool to catch syntax errors (`apachectl configtest`) that the administrator may use for this task. Once the server is working correctly, the administrator can abstract the changes to make it easier to migrate other servers by adding an optional `https` parameter to the `website` type, as sketched in Figure 4.

However, the configuration has again drifted from the manifest. In this case, Tortoise detects that the configuration for piedpiper.com is a concrete instance of the abstraction and automatically adds the `https` attribute:

```
website{"www.piedpiper.com": root => "piedpiper", https => true }
```

This repair is notable because it repairs an instantiation of a type that is not built-in to Puppet.

Summary. We have presented three ways in which Tortoise allows a system administrator to use Puppet and a shell in harmony, benefiting from the unique strengths of each tool. In all cases, Tortoise synthesizes maintainable patches and ensures that no configuration drift occurs.

```
1 file{"/fileA": content => "test"}
2
3 define T($x,$prefix) {
4   if ($prefix) {
5     file{"/dir/$x": content => "test"} }
6   else {
7     file{$x: content => "test"} } }
8
9 T{x => "fileB", prefix => true}
10 T{x => "fileC", prefix => true}
```

Fig. 5: Repair example.

III. The Tortoise Repair Space

Puppet's own linting tools can help administrators fix syntax errors and type errors. However, there are three more ways in which a manifest may need to change. Tortoise helps administrators make the third type of change listed below.

1) **Adding, removing, or modifying dependencies.** The dependencies in a manifest impose a partial ordering on resources. Although Puppet automatically inserts certain dependencies (*"auto-requires"*), others need to be specified explicitly by the administrator. Missing dependencies can cause a manifest to raise an error during deployment. Tortoise does not correct dependency errors, but this is the subject our prior work [47].
2) **Creating new abstractions.** A powerful feature of Puppet is its ability to create new abstractions (defined types and classes) to make manifests modular and reusable. For example, in Section II, we created a `website` abstraction to help manage a website. Tortoise does not help the user create new abstractions. However, given a manifest that has user-defined abstractions, Tortoise can perform repairs within them.
3) **Creating, deleting, and updating resources.** Tortoise supports repairs that involve deleting resources and creating new resources, including instances of user-defined abstractions. In addition, Tortoise supports repairs that involve creating, deleting, and modifying attributes of existing resources, as detailed next.

The rest of this section gives examples of Tortoise-supported repairs. We describe individual repairs in isolation, but a single repair may involve several repairs of the kind illustrated below.

A. Supported Repairs

The most significant class of repairs that Tortoise performs involves adding, removing, and updating attributes on existing resources. Puppet has dozens of resource types and each type has several attributes that can dramatically change how the resource is interpreted. What makes Tortoise powerful is its ability to correct the attributes of both built-in and user-defined resources. To illustrate this, we use the manifest in Figure 5, which has one defined type, `T`, and creates three files. It creates `/fileA` directly, but uses the type `T` to create `/dir/fileB` and `/dir/fileC`. An interesting feature of `T` is that it checks to see if the `$prefix` attribute is set, and if it is, it builds a filename using string interpolation.

Add new attribute. Tortoise can add new constant-valued attributes to a resource. For example, in Figure 5, if the administrator uses the shell to change the owner of `/fileA` to `alice`, Tortoise will add the attribute `owner => alice` to the corresponding resource. If she instead changes the owner of `/dir/fileB` to `alice`, then Tortoise suggests two possible changes, in order:

1) Add an attribute on line 5 that affects `fileC` too:
```
owner => alice
```
2) Change line 5 to create a special case for `fileB`, which does not affect `fileC`:
```
$title == "fileB" ? owner => alice : owner => root
```

Delete existing attribute. Tortoise can also delete attributes. For example, if the administrator changes the owner of `/fileA` back to `root`, it will suggest removing the `owner` attribute.

Update existing constant. In Section II, we saw that Tortoise can update the value of constants in attributes. The same mechanism allows Tortoise to update attribute titles. For example, renaming `/fileA` to `/fileA2` causes Tortoise to update the manifest to refer to the new file (Line 1 of Figure 5). A harder repair involves renaming the files that are created indirectly by the defined type. For example, we could rename `/dir/fileB` in three ways:

1) If renaming the file part, e.g., to `/dir/fileB2`, the repair is in the instantiation of `T` (line 9).
2) If renaming the directory part, e.g., to `/dir2/fileB`, the repair is in the definition of `T` (line 3).
3) If renaming both, e.g., to `/dir2/fileB2`, the repair must affect both locations.

Tortoise supports all three repairs.

Create and delete resource. Tortoise can create and delete resources. For example, if the user deletes `/fileA`, Tortoise suggests removing line 1 from the manifest. On the other hand, if the user creates a new file `/dir/fileD` with the same content specified in the definition of `T`, Tortoise suggests two fixes: (1) create a new **`file`** or (2) create a new `T` resource.

B. Repair Consistency

A key property of Tortoise is that it produces *consistent* repairs: a repair is guaranteed to preserve all changes made using the shell that are within the scope of Tortoise's system model. Section IV presents this model in detail, but at a high-level, we model certain essential properties of regular files and directories, such as their contents and permissions. In contrast, Tortoise does not support repairs that affect running processes or special files, such as the `/proc` file system. For example, many changes to the `/proc` file system are lost after reboot, but can be persisted by editing certain configuration files in the `/etc` directory. These kinds of repairs are straightforward in principle, but would require a lot of engineering.

IV. From Manifests and Shell Commands to ΔP

Section IV-A introduces our modeling language ΔP that provides a uniform way to model the semantics of manifests, the constraints generated from shell commands, and the space of possible repairs. Sections IV-B and IV-C describe primitive and user-defined resources. Section IV-D describes how we model repairs that create and delete resources. Finally, Section IV-E details how manifests and shell commands are translated to ΔP. Section V will describe translating ΔP into formulae for an SMT solver. While it is possible to directly translate manifests and shell commands into constraints, using ΔP has two advantages: (1) it is much easier to model the semantics of manifests in ΔP since it has imperative file-system operations and (2) we can simplify ΔP programs before generating constraints, which makes constraint solving scalable.

Atomic Expressions

a ::= *str*	String
| *bool*	Boolean
| n	Integer
| `undef`	Undefined
| x	Variable reference

Expressions

e ::= a	
| `file?(`a`)`	Test if a refers to a file
| `dir?(`a`)`	Test if a refers to a directory
| `exists?(`a`)`	Test if a refers to a file or directory
| `defined?(`a`)`	Test if not `undef`
| e_1 `+` e_2	String concatenation
| ···	Comparisons and boolean connectives

Statements

c ::= `let` x `=` e	Variable declaration
| **`if`** `(`e`)` c_1 **`else`** c_2	Conditional
| `{` c_1`;` ···`;` c_n `}`	Block statement
| `chmod(`e_1`,`e_2`)`	Set permissions of e_1 to e_2
| `chown(`e_1`,`e_2`)`	Set owner of e_1 to e_2
| `mkdir(`e`)`	Create directory
| `write(`e_1`,`e_2`)`	Create file e_1 with contents e_2
| ···	Other file system operations
| `rlet` x `=` a `from` r	Let x be a, but can be repaired to r
| `assert(`e`)`	Assertion

Repair Spaces

r ::= `[`a_1`;` ···`;` a_n`]`	Finite set of alternatives
| `str`	Any string or `undef`
| `int(`n`)`	Any n-bit number or `undef`

Fig. 6: ΔP Syntax

A. The ΔP Modeling Language

ΔP is an imperative language with primitive operations that manipulate files, so it allows us to model the side-effects that resources have on system state. In addition, it has two features that facilitate repair: (1) it has *repairable variable declarations*, which are ordinary variables that are augmented with a *repair space* of alternate values and (2) it has assertions, which we use to constrain repair spaces. Intuitively, a single ΔP program with repairable variables represents a space of possible programs, ranked by the number and kinds of repairs made. The key to our approach is to translate manifests to ΔP programs with repairable variables and to turn shell commands into ΔP assertions that rule out programs that are not consistent with the user's repair.

File system operations. Figure 6 shows the syntax of ΔP, which consists of statements, expressions, atomic expressions, and repair spaces. Atomic expressions include constants and variable references, which are the simplest kinds of expressions that can appear in manifests. Atomic expressions also include

```
let title = "/fileA";
let content = "Hello world";
if (exists?(title)) {
  rm(title);
}
write(title, content)
```

(a) A trivial encoding.

```
file{"/fileA"^t:
  content => "Hello world"^c
  source => undefined^s
  mode => undefined^m
  ensure => undefined^e
}
```

(b) The annotated manifest.

```
rlet t = "/fileA" from str;
rlet c = "Hello world" from str;
rlet s = undef from str;
rlet e = undef from str;
rlet m = undef from int(9);
if (exists?(t)) { rm(t); }
if (e == "directory") {
  assert(c == undef and s == undef);
  mkdir(t)
}
else if (e == "file" or e == undef) {
  assert(defined?(s) xor defined?(c));
  if (defined?(s)) { cp(s, t); }
  if (defined?(c)) { write(t, c); }
}
else {
  assert(false);
}
if defined?(m) { chmod(t, m); }
```

(c) A repairable encoding.

Fig. 7: A file resource and a portion of its repair space.

the special value `undef`, which we use to explicitly indicate that an optional attribute is not present.

ΔP's expressions include predicates to test if a path refers to a file (`file?`), a directory (`dir?`), or is non-existant (`exists?`). These predicates only read file system state and do not perform writes. For convenience, ΔP also has a predicate to test that an expression is not the special value `undef` (`defined?`). Finally, expressions include all atomic expressions as well as conventional comparisons and boolean operators, which we elide from the figure.

ΔP's statements have imperative operations that model file system updates, including operations to create files (`write`), create directories (`mkdir`), set file permissions (`chmod`), set file ownership (`chown`), and so on. ΔP also has conditionals (**`if`**), immutable variables (`let`), and block statements.

Assertions and repairable variable declarations. An unusual feature of ΔP is that it supports *repairable variable declarations*. The statement `rlet` x `=` a `from` r binds the variable x to the atomic expression a and specifies that r is its *repair space*. ΔP supports three sorts of repair spaces:

1) A finite set of atomic expressions, which may include variables (`[`a_1`; ⋯;` a_n`]`);
2) The space of all strings (`str`); and
3) The space of n-bit integers, for a fixed n (`int(`n`)`).

A repairable variable also expresses the soft constraint that x should be a if possible, thus there is a cost associated with picking an alternate value from r. In contrast, an assertion expresses a hard constraint that cannot be violated (`assert(`e`)`). One way to rank repairs would be by the number of soft constraints violated, but Section V presents a more subtle ranking procedure that works better in practice.

B. Primitive Resources

We now present our model of two key Puppet types.

The file type. The `file` type only manages a single file, but it has 32 optional attributes, some of which dramatically alter its semantics. For brevity, we only discuss five representative attributes, but our implementation supports other attributes too:

1) The `ensure` attribute determines if the resource is a file or directory. If omitted, it is assumed to be a file.
2) The `content` attribute specifies the file source inline and the `source` attribute copies contents from another file. These attributes are mutually exclusive. If the resource is managing a directory then neither may be defined.
3) The `mode` attribute sets the file's permissions.

ΔP has the file system operations needed to model all the behaviors described above. For example, consider the following resource which only specifies a single attribute:

```
file{"/fileA": content => "Hello, world" }
```

We could model this resource as a trivial ΔP program that deletes an existing file or directory, if needed, and replaces it with the specified file (Figure 7a).[1] However, the resource needs to use repairable variables to support repair.

To encode the full repair space, we take the following steps: (1) we produce a program with five repairable variables, one for the title and four for each possible attribute (Figure 7c); (2) we add all unused attributes to the resource and explicitly mark them as undefined (Figure 7b), and (3) we annotate atomic expressions in this manifest with the names of repairable variables. The program in Figure 7c first declares the repairable variables, though note that all variables except `c` and `t` are set to `undef`. After these variables are declared, the program has several cases that describe the space of all behaviors for a `file` resource. With no repairs, the program reduces to the trivial program in Figure 7a. However, repairs can make the other cases relevant.

For example, suppose the user removes the file and creates a directory with the same name. This change produces the assertion `assert(dir?("/fileA"))`, which must hold at the end of the program. The only way to satisfy this assertion, is to make the two following repairs: (1) the variable `e` must be repaired to `"directory"`, since that is the only way that the branch with the `mkdir` statement is reachable, and (2) the variable `c` must be repaired to `undef`, since the branch asserts that `c` must be `undef`. Finally, it is easy to propagate the repair back to the manifest, since we had annotated atomic expressions with their corresponded repairable variables.

The package type. The `package` type is very common in manifests and is a kind of resource type that Tortoise models in a special way. We model a resource that installs a package `p` using provider `s` as a ΔP program that creates an empty file called `s://p` (Figure 8a). Conversely, we model a resource that removes a package `p` using provider `s` as a ΔP program

```
package{"vim"^p:
  ensure => present^e
}
```

(a) The resource.

```
rlet p = "vim" from str;
rlet e = "present" from str;
rlet s = "dpkg" from str;
if (e == "present") {
  create(s + "://" + p, "");
}
else if (e == "absent") {
  rm(s + "://" + p);
}
```

(b) The ΔP model.

Fig. 8: A package resource and its model.

[1] In practice, Puppet would not replace the file if it already had the specified contents. However, our simplified model is adequate for modeling repairs.

```
1 define T($title) {
2   file{$title + "/A"y:
3     content => "textA" }
4   file{$title + "/B"z:
5     content => "textB" }
6 }
7 T{"/dir1"x: }
```

(a) Defined type.

```
1 rlet x = "/dir1" from str;
2 rlet y = "/A" from str;
3 rlet z = "/B" from str;
4 let title0 = x + y;
5 let title1 = x + z;
6 ...
```

(b) ΔP model.

Fig. 9: A naive expansion of a defined type.

that deletes the file `s://p`. Since a repair may either remove an installed package or change the package that is installed, we translate a package resource into a ΔP program with three repairable variables: one for the title, one for the provider, and one for the `ensure` attribute, which determines if package is present or absent (Figure 8b).

To repair a package from the shell, the Tortoise user has to use standard commands, e.g., `apt install` or `apt remove`.[2] When Tortoise monitors system calls from the shell, the system call trace includes commands to launch these programs. We translate invocations of these programs to constraints that create and delete files in the `dpkg://` path. For example, the command `apt remove vim` produces:

```
assert(file?("dpkg://vim") == false)
```

This assertion does not hold after the program in Figure 8b executes, unless we repair the variable `e` to `"absent"`.

Other types. Puppet has several other built in types (48 as of this writing), many of which are operating system-specific. With two exceptions, all types update the state of the file system. Our implementation supports several other common types, such as user accounts, SSH keys, cron jobs, and more. ΔP makes it easy to add support for new types, since it has the primitives needed to model types and their repair spaces. The only two resource types that do not update the file system are (1) *notify*, which prints a log message and has no effect and (2) *service*, which starts and stops running services. The former type is irrelevant for repairs and the latter could be supported with some extensions to ΔP.

C. User-Defined Resources

A manifest can define new resource types, known as *defined types*. A defined type can be thought of as function that produces a manifest. For example, the manifest in Figure 9a defines a type `T` that takes a directory name as a parameter and produces two file resources within that directory. The manifest uses `T` to create two files in the directory `/dir1`. Suppose we use the shell to rename the file `/dir1/A` to `/dir2/C`. The only way to make this edit is to change `dir1` to `dir2` (line 7) and `A` to `C` (line 2). The former edit has the added effect of renaming `/dir1/B` to `/dir2/B`. We express this dependency in the ΔP model by never duplicating atomic expressions in the manifest (Figure 9b).

[2] `apt` is the package manager on Debian-based systems. It should be straightforward to support other package managers too.

D. Creating and Deleting Resources

To support repairs that delete resources, we wrap the statements of each resource in a conditional that is guarded by a repairable boolean variable with the initial value `true`. If the value of the boolean is repaired to `false`, then none of the resource's statements take effect, which corresponds to the resource being deleted. We ascribe resource deletions a much higher cost than attribute edits. We support repairs that create new resources in a similar way, by creating template resources that are guarded with a repairable variable that is instead initialized to `false`.

E. From Shell Commands to Constraints

Tortoise does not parse shell commands but instead intercepts all system calls made during repair. For example, the system call `mkdir("/dirA")` turns into the assertion `assert(dir?("/dirA"))`. In a single repair session, a user may make and revert changes. For example, the command `rmdir /dirA` produces the assertion `assert(!dir?("/dirA"))`. However, if the user first creates and then removes the directory, simply joining both assertions is contradictory. Tortoise handles this kind of case by calculating the strongest postcondition of the system call sequence instead of naively turning each call into an assertion.

V. From ΔP to Logical Formulae

We now discuss how we translate ΔP programs into logical formulae for an SMT solver, specifically Z3-str [63]. The formulae that we produce use the theories of bit-vectors and equalities between concatenated strings and string variables. In our encoding, each model returned by the solver can be interpreted as a combination of a repair, which assigns values to the repairable variables, and a set of variables indicating which repairable variables have changed from their initial value.

At a high-level, we transform a ΔP program into a formula (ϕ) with the following variables:

- $\vec{fs}_{in}$ and $\vec{fs}_{out}$ are sets of variables that model the initial and final state of the file system;
- $\vec{x}$ are the values assigned to the repairable variables (whether or not they are repaired); and
- n counts the number of repairs made.

We generate ϕ such that for all assignments to these variables, ϕ is true, if and only if the modeled program updates the initial file system ($\vec{fs}_{in}$) to the final file system ($\vec{fs}_{out}$), with exactly n repairs to the repairable variables ($\vec{x}$). Therefore, our goal is to find an assignment to the repairable variables such that ϕ holds for all input and output file systems:

$$\exists n, \vec{x} \ . \forall \ \vec{fs}_{in}, \vec{fs}_{out} \ . \phi \left(n, \vec{x}, \vec{fs}_{in}, \vec{fs}_{out} \right)$$

To produce solutions ordered by the number of repairs, we iteratively increase n and search for $\vec{x}$ using counterexample-guided inductive synthesis [51].

Encoding file systems. We model each path (p) using four variables per path:

- The state of the path (s_p): is s_p a file, directory, or none;

`exists?(`p`)`	$s_p = \textbf{dir} \vee s_p = \textbf{file}$
`mode(`p`) = 0700`	$s_p \neq \textbf{none} \wedge m_p = \textbf{0755}$
`contents(`p`) = "hello"`	$s_p = \textbf{file} \wedge c_p =$ `"hello"`

Fig. 10: Examples of expressions and their encodings.

- The contents (c_p), if c_p is a file;
- The owner (o_p), if o_p is a file or directory; and
- The mode (m_p), if m_p is a file or directory.

We model a file system by modeling every possible path. Although the space of paths is potentially unbounded, we only need to consider the (prefixes of) paths that appear in the ΔP program. Recall that we encode repairs as assertions, therefore we model all paths that a repair affects, even if the repair affected paths that did not appear in the original manifest.

Encoding expressions and statements. Since ΔP expressions only read the state of the file system, they turn into predicates. Figure 10 translates some example expressions to predicates. Since ΔP statements update the state of the file system, we model them as relations between two sets of variables that represent the input and an output file system. For example, the statement `mkdir(/x)` constraints the state of `/x` in the output file system ($s'_{/x}$) to be a directory. The mode and owner are also set to `0755` and `"root"` respectively, which are Puppet's defaults. The content variable ($c'_{/x}$) is left unconstrained, which is safe to do, since its value is uninterpreted for directories. Finally, the relation constrains the variables for all other paths such that they are the same in the input and output state:

$$s'_{/x} = \textbf{dir} \wedge o'_{/x} = \texttt{"root"} \wedge m'_{/x} = \textbf{0755} \wedge$$
$$\forall p.p \neq /x \Rightarrow (s_p = s'_p \wedge c_p = c'_p \wedge o_p = o'_p \wedge m_p = n'_p)$$

We translate all other primitive statements in a similar way. Finally, we translate blocks and conditionals by introducing intermediate states and flattening nested conditionals.

Encoding Repairable Variables. A repairable variable, `rlet` x `=` a `from` r, turns into a new existentially quantified variable (x) with the specified domain (r). A repairable variable also has a cost, which is defined as follows: if the value of the variable in a model is equal to the original value (a), then the cost is 0, otherwise the cost is 1. The total cost of repairing a manifest is the sum of all unit costs.

Optimizing Update Synthesis. To speed up repairs for large manifests, we use a minimization procedure that turns repairable variables into constants when it is provably safe to do so, by propagating information from the shell-based repair to the ΔP model of the manifest. We transform the ΔP program, translating repairable variable declarations for paths not affected by the shell commands to ordinary let bindings. In doing this, we have shrunk the overall number of repairable declarations substantially, making the overall Tortoise performance based more around the size of the update rather than the size of the manifest (Section VI-C).

Ranking Repairs. Each model produced by the solver can be interpreted as a repair and a ranking. Tortoise first ranks repairs by the number of repairable variables changed, favoring repairs that makes fewer changes. To break ties between repairs with the same number of changes, Tortoise favors repairs that make fewer changes within defined types. The intuition behind this tie-breaking procedure is that changes within defined types have the potential to affect more resources, whereas changes outside defined types only affect a single resource. Therefore, Tortoise primarily ranks repairs based on the number of syntactic edits, but the secondary ranking favors repairs typically make fewer semantic changes. However, all repairs produced by Tortoise are consistent with the changes made from the shell (Section III-B).

Benchmark	# of resources	# of repair scenarios	Tortoise runtime (ms)	Average repair rank
amavis	6	1	25	1.00
bind	6	3	21	1.60
clamav	6	2	23	3.50
hosting	19	1	26	1.00
irc	18	1	292	1.00
jpa	10	1	21	1.00
logstash	14	6	48	1.00
monit	7	4	25	1.00
nginx	9	4	27	1.00
ntp	4	3	18	1.33
powerdns	5	7	39	1.43
rsyslog	7	4	129	1.25
xinetd	4	5	1,970	1.20
Total	115	42	205	1.31

Fig. 11: Benchmark of real-world Puppet manifests [47]. We identified a total of 42 scenarios in which the manifests would require repair. On average, Tortoise took 205 ms to compute the repairs, and the average rank of the ideal update was 1.31.

Applying Updates. Once a repair is chosen, applying an update is straightforward. Recall that we annotate each atomic expressions in the manifest with the name of the repairable variable that holds its value. We only update those atomic expressions whose repairable variables have been updated.

VI. Evaluation

We evaluate Tortoise in three ways. Section VI-A evaluates the quality of Tortoise-synthesized patches in an experiment by measuring how highly Tortoise ranks correct patches on real-world manifests. Section VI-B presents case studies of some of these manifests. Finally, Section VI-C evaluates Tortoise's scalability on synthetic benchmarks.

A. Evaluating Repair Rankings

We studied the 13 manifests in a Puppet benchmark [47] to identify scenarios in which the manifests may need to be repaired. (Section VI-B highlights some of these repairs.) We identified a total of 42 such repair scenarios. For each scenario, we used Tortoise to perform imperative configuration repair, instead of manually patching the manifest. We ran Tortoise 50 times per scenario to measure average performance. Figure 11 summaries the benchmarks, their size in number of resources, the number of distinct scenarios for that benchmark, and the average time Tortoise took to perform the repair.

For our experiment, we configured Tortoise to produce the five, highest-ranked repairs for each repair scenario. We took

```
1  $params::package_name = "pdns-server"
2  $params::package_provider = "dpkg"
3  define powerdns::install(
4    $package = $params::package_name,
5    $ensure = present,
6    $source = undefined,
7    ...) {
8    package {$package:
9      ensure => $ensure,
10     source => $source,
11     provider = $params::package_provider,
12   }
13 }
14 powerdns::install { ensure => present }
```

(a) PowerDNS

```
1 define ntp ($logfile = 'false', ...) {
2   if ($logfile != 'false') {
3     file { '/etc/logrotate.d/ntpd':
4       ensure  => present,
5       ...
6     }
7   }
8   ...
9 }
10 ntp { logfile => true, ... }
```

(b) NTP

```
1 define xinetd($server_args, $port, ...) {
2   file { "/etc/xinetd.d/rsync":
3     ensure  => present,
4     content => "$server $server_args $port",
5   }
6 }
7
8 $cf = '/etc/rsync.conf'
9 $args = "--daemon --config $cf"
10 xinetd {
11   server_args => $args,
12   port => 873
13 }
```

(c) xinetd

Fig. 12: Portions of three manifests from our benchmarks.

each ranked list, randomized its order, and presented the repairs to one of the authors. The author (without knowing Tortoise's ranked order) selected that repair that captured the intent of the shell commands and labeled it "correct". (Recall that while all repairs are guaranteed to be consistent, some may capture and generalize the intent of the shell command better than others.) The average rank of the correct repair in Tortoise's ranked lists was 1.31. Overall, the highest-ranked repair was the correct repair 76% of the time.

B. Repairs to Real-World Manifests

We describe three different types of repairs from our benchmarks as case studies of Tortoise usage.

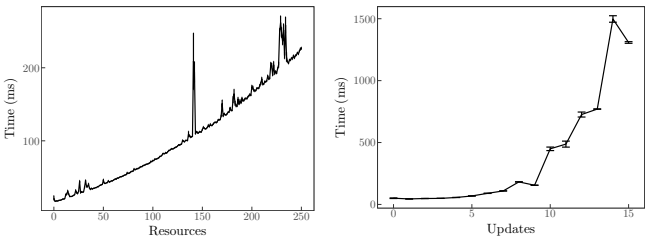
Operating system update. Different Linux distributions offer the same packages under different package names. For example, the *PowerDNS* benchmark (Figure 12a) installs the `pdns-server` package on Debian, but fails on Red Hat where the package is called `pdns`. Running `yum install pdns` from the Red Hat system shell fixes the problem; here, Tortoise's highest-ranked patch is the correct patch. It is notable that the benchmark does not directly create the package. Instead, it has a global variable that's bound to the package name and is used within a defined type.

Updating optional resources. Many reusable manifests provide optional features that the administrator may want to turn on or off and Tortoise can help with these repairs. For example, the NTP benchmark (Figure 12b) has an abstraction that optionally creates a log file. Suppose that the log is initially enabled, but that it subsequently needs to be removed (e.g., because of limited disk space or the log is deemed unnecessary). Tortoise allows the administrator to simply delete the log file from the shell. Its highest-ranked repair changes the `logfile` flag, which is the right way to perform this update.

Configuration file updates. Manifests often use string interpolation to create configuration files from templates. Figure 12c shows a fragment of a benchmark that creates a configuration file in this way. We used a text editor to update the configuration file, changing `rsync.conf` to `piperchat.conf`, and Tortoise updated the variable on line 8.

C. Scalability Experiments

Tortoise's running time is dominated by the SMT solver. The size of each problem depends on (1) the size of the manifest and (2) the number of shell commands. Figure 13a shows how the running time of Tortoise varies with the number of resources. We use a synthetic benchmark that creates n distinct resources and uses the same shell command to update all manifests. The graph shows that Tortoise produces an update in less than 1.5s, even when the manifest has 250 resources. Figure 13b shows how the running time of Tortoise varies with the number of shell commands while the size of the manifest increases proportionally. We generate a sequence of n shell commands where each updates a different resource. Tortoise takes up to 1.5s with 15 shell commands and over a minute for more.



(a) Varying manifest size. (b) Varying update size.

Fig. 13: Scalability: average of 10 trials with standard error.

Although 15 shell commands may appear to be restrictive, note that we can easily batch them. If a repair requires 30 shell commands, we can issue the first 15 to get the manifest to an intermediate state and then issue the next 15 to get the manifest to a final state. Also note that each shell command in the benchmark is an update that leads to a repair. Shell commands that do not update the file system do not generate constraints. Therefore, Tortoise allows the administrator to explore the system as long as she likes.

VII. Scope and Limitations

Threats to Experimental Validity. For our evaluation, one author subjectively measured update correctness. Real system administrators would provide a more accurate correctness measure. We used a suite of benchmarks collected from public GitHub repositories, but did not ask the systems' developers to identify the repairs in these benchmarks. Instead, we injected faults to create repair scenarios. A future user study of industrial users could evaluate Tortoise's usefulness in practice.

Unsupported Puppet Features. Puppet is a sophisticated, evolving language and Tortoise supports a significant subset of Puppet features. Our prototype does not support certain features such as inheritance (which Puppet documentation states should be used "very sparingly") and lambdas (a recent language feature not yet widely used). Nevertheless, it would be possible to add support for these features with more engineering effort. Puppet also has two notable extra-linguistic features: manifests may have embedded shell scripts (the `exec` type) and string templates written in Ruby (ERB). Repairing these features are beyond the scope of this paper.

Limitations of the ΔP Model. Section III describes three classes of repairs, but Tortoise's repair space only includes repairs that add, remove, or update resources. Therefore, Tortoise is *not complete* with respect to the full space of desirable repairs. However, for its supported repairs, Tortoise produces repairs that are *consistent* with changes made from the shell that are within the scope of ΔP (Section III-B). ΔP only models a few key attributes of regular files and directories. If a shell command performs an update beyond the scope of the model, Tortoise will not detect it. For example, if a manifest is configured to start a service and the user terminates the service from the shell, Tortoise will not be able to repair the manifest. Is is possible, in principle, to support this repair by enhancing ΔP to model processes and intercepting more system calls. In practice, Tortoise will require careful engineering to support each kind of primitive resource. This paper supports a subset of common primitive resources.

Interaction Model. During repair, we assume that all changes to the machine are made using the Tortoise shell and we do not detect changes made by background processes. In principle, it is straightforward to support concurrent shells if their system call logs can be totally ordered.

VIII. Related Work

Program Repair and Synthesis. Fundamentally, Puppet manifests are programs and Tortoise is an automated repair tool that uses shell commands as partial specifications of desired behavior. This is not unlike most automated program repair tools [3], [5], [6], [7], [10], [11], [12], [13], [14], [15], [19], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [36], [37], [38], [39], [40], [41], [43], [44], [48], [49], [53], [55], [56], [57], [60] that use partial specifications, often tests, to produce program patches that satisfy those specifications.

Our repair approach is a form of syntax-guided synthesis [1]. Tortoise models a space of possible repairs, similar to the repair models of Singh et al. [50]. Whereas they repair student-written programs to conform to complete, teacher-provided specifications, Tortoise uses partial specifications provided from shell commands and ranks candidate repairs based on size. Tortoise allows users to freely manipulate a manifest and its output while propagating changes from one to the other, which is similar to prodirect manipulation [9].

Synthesis-based program repair tools, e.g., Angelix [37], DirectFix [36], and SemFix [38], synthesize patches for more complex C programs than Puppet manifests. Because manifests are relatively simpler, Tortoise (1) is much faster, (2) generates and ranks multiple patches for the user to select the best one, and (3) does not require the user to write tests, instead turning shell commands into assertions to guide repair, which is a more natural interface for system administrators.

Configuration Languages. Automated testing and verification of system configuration languages has focused on universal properties such as convergence [21], idempotence [24], and determinism [47]. These universal properties are necessary, but insufficient for a manifest to be correct. Tortoise is an interactive repair tool that can repair logic errors too. ConfValley [23], PCheck [61], and ConfigC [46] are complementary tools that validate program-specific configuration files.

Tools like ConfSuggester [62], AutoBash [52], and ConfAid [4] find errors in configuration files, using dynamic analysis to track how configuration values affect program execution. When a Puppet manifest creates a buggy configuration file, it is the manifest the needs to be repaired and not the generated configuration file itself. Given a fixed configuration, Tortoise can repair a manifest and thus compliments these tools.

μPuppet [17] formalizes a subset of Puppet, including many language features that Tortoise does not support. In contrast, Tortoise models the effects that resources have on system state (i.e., *resource realization*), which is out of scope for μPuppet.

Shell Script Analysis. Tortoise complements shell script bug-finding tools, such as ABash [35] and synthesis tools, such as StriSynth [20], as it works on Puppet manifests.

Other configuration languages. Tortoise leverages Puppet's DSL to model resources, which should be possible for languages like Salt [22], Ansible [45], and LCFG [2], but harder for Chef [8], a Ruby-embedded domain-specific language.

IX. Contributions

System configuration languages, such as Puppet, can make system administration easier. However, manifests often have bugs and the shell is often the best tool for diagnosing bugs. Using Tortoise, administrators can fix bugs using the shell, because Tortoise automatically synthesizes repairs to the underlying manifest. We have demonstrated that Tortoise is fast on reasonably sized manifests, and that 76% of the time, it produces repairs equivalent to those written by humans.

Acknowledgments

We thank the anonymous reviewers, Shriram Krishnamurthi, and Christian Kästner for their thoughtful feedback. We thank Rachit Nigam for his work on an early version of Tortoise. This work is supported by the National Science Foundation under grants CNS-1413985, CCF-1453474, CCF-1564162, CCF-1717636, and CNS-1744471. Aaron Weiss was affiliated with UMass Amherst during his primary work on Tortoise.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[2] Paul Anderson. Towards a high-level machine configuration system. In *USENIX Large Installation System Administration Conference (LISA)*, 1994.

[3] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation*, pages 162–168, 2008.

[4] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[5] Jeremy S. Bradbury and Kevin Jalbert. Automatic repair of concurrency bugs. In Massimiliano Di Penta, Simon Poulding, Lionel Briand, and John Clark, editors, *International Symposium on Search Based Software Engineering (SSBSE) fast abstract*, Benevento, Italy, September 2010.

[6] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming (ECOOP)*, Lancaster, England, UK, July 2011.

[7] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 782–791, San Francisco, CA, USA, 2013.

[8] Chef. Chef, inc. https://www.chef.io, 2009. Accessed Sep 8 2017.

[9] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[10] Zack Coker and Munawar Hafiz. Program transformations to fix C integers. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 792–801, San Francisco, CA, USA, 2013.

[11] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *IEEE/ACM International Conference on Automated Software Engineering (ASE) short paper track*, pages 550–554, Auckland, New Zealand, November 2009.

[12] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, Paris, France, 2010.

[13] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA)*, pages 30–39, Hyderabad, India, 2014.

[14] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 233–243, Portland, ME, USA, July 2006.

[15] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *ACM/IEEE International Conference on Software Engineering (ICSE) Formal Demonstration track*, pages 855–858, Leipzig, Germany, 2008.

[16] Facebook Engineering Team. More details on today's outage. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/, 2010. Accessed Sep 8 2017.

[17] Weili Fu, Roly Perera, James Cheney, and Paul Anderson. μPuppet: A declarative subset of the Puppet configuration language. In *European Conference on Object-Oriented Programming (ECOOP)*, 2017.

[18] Google Inc. Google App Engine incident 16008. https://status.cloud.google.com/incident/appengine/16008, 2016. Accessed Sep 8 2017.

[19] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 173–188, Saarbrücken, Germany, March 2011.

[20] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. StriSynth: synthesis for live programming. In *International Conference on Software Engineering (ICSE)*, 2015.

[21] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. Asserting reliable convergence for configuration management scripts. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016.

[22] Thomas S. Hatch. Salt. https://saltstack.com, 2011. Accessed Sep 8 2017.

[23] Peng Huang, William J. Bolosky, and Yuanyuan Zhou Abhishek Singh. ConfValley: A systematic configuation validation framework for cloud services. In *European Conference on Computer Systems (EuroSys)*, 2015.

[24] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence and convergence for infrastructure as code. In *ACM/IFIP/USENIX International Middleware Conference*, 2013.

[25] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *International Conference on Program Comprehension (ICPC)*, pages 70–79, Vancouver, BC, Canada, May 2009.

[26] Mingyue Jiang, Tsong Yueh Chena, Fei-Ching Kuoa, Dave Toweyb, and Zuohua Dingc. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software*, 2016.

[27] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 389–400, San Jose, CA, USA, 2011.

[28] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306, 2015.

[29] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 802–811, San Francisco, CA, USA, 2013.

[30] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *AMC/IEEE International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich, Switzerland, 2012.

[31] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38:54–72, 2012.

[32] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 299–309, Zurich, Switzerland, 2012.

[33] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 166–178, Bergamo, Italy, 2015.

[34] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 298–312, St. Petersburg, FL, USA, 2016.

[35] Karl Mazurak and Steve Zdancewic. Abash: Finding bugs in Bash scripts. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007.

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*, 2015.

[37] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, 2016.

[38] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Satish Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*, 2013.

[39] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[40] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering (TSE)*, 40(5):427–449, 2014.

[41] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[42] Puppet, Inc. Puppet. https://www.puppet.com, 2005. Accessed Sep 8 2017.

[43] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance (ICSM)*, pages 180–189, Eindhoven, The Netherlands, September 2013.

[44] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36, Baltimore, MD, USA, 2015.

[45] Red Hat, Inc. Ansible. https://www.ansible.com, 2012. Accessed Sep 8 2017.

[46] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. Probabilistic automated language learning for configuration files. In *International Conference on Computer-Aided Verification (CAV)*, 2016.

[47] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: A configuration verification tool for Puppet. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[48] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, November 2005.

[49] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 43–54, Portland, OR, USA, 2015.

[50] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[51] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit A. Seshia. Combinatorial sketching for finite programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[52] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[53] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[54] The Wall Street Journal. NYSE says Wednesday outage caused by software update. http://www.wsj.com/articles/stocks-trade-on-nyse-at-open-1436450975, 2015. Accessed Sep 8 2017.

[55] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, Trento, Italy, 2010.

[56] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Palo Alto, CA, USA, 2013.

[57] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 364–374, Vancouver, BC, Canada, 2009.

[58] Kim Weins. New DevOps trends: 2017 state of the cloud survey. http://www.rightscale.com/blog/cloud-industry-insights/new-devops-trends-2017-state-cloud-survey, 2017.

[59] Lucas Welch. Chef appoints companys first chief marketing officer to drive continued growth. https://blog.chef.io/2016/11/10/chef-appoints-companys-first-chief-marketing-officer, 2016.

[60] Josh L. Wilkerson, Daniel R. Tauritz, and James M. Bridges. Multi-objective coevolutionary automated software correction. In *Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1229–1236, Philadelphia, PA, USA, 2012.

[61] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[62] Sai Zhang and Michael D. Ernst. Which configuration option should I change? In *International Conference on Software Engineering (ICSE)*, 2014.

[63] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2013.