2018 IEEE International Symposium on Software Reliability Engineering Workshops

An Approach to Identifying Error Patterns for Infrastructure as Code

Wei Chen*†, Guoquan Wu*†‡, Jun Wei*†‡
*Institute of Software, Chinese Academy of Sciences, Beijing, China
†University of Chinese Academy of Sciences, Beijing, China
‡State Key Laboratory of Computer Sciences, Beijing, China
Email: {wchen, gqwu, wj}@otcaix.iscas.ac.cn

***Abstract*—Infrastructure as Code (IaC), which specifies system configurations in an imperative or declarative way, automates environment set up, system deployment and configuration. Despite wide adoption, developing and maintaining high-quality IaC artifacts is still challenging. This paper proposes an approach to handling the fine-grained and frequently occurring IaC code errors. The approach extracts code changes from historical commits and clusters them into groups, by constructing a feature model of code changes and employing an unsupervised machine learning algorithm. It identifies error patterns from the clusters and proposes a set of inspection rules to check the potential IaC code errors. In practice, we take Puppet code artifacts as subject objects and perform a comprehensive study on 14 popular Puppet artifacts. In our experiment, we get 41 cross-artifact error patterns, covering 42% crawled code changes. Based on these patterns, 30 rules are proposed, covering 60% identified error patterns, to proactively check IaC artifacts. The approach would be helpful in improving code quality of IaC artifacts.**

***Index Terms*—Infrastructure as Code; Error pattern; Puppet artifact**

I. Introduction

Infrastructure as Code (IaC) [1] automates environment set up, system deployment, configuration and run-time management. The annual report[1] states that in 2016 there are over 43% enterprises surveyed are using the *state-of-the-art* IaC tools, e.g. Puppet, Chef and Ansible. The popular IaC communities publicly offer a large number of *off-the-shelf* artifacts. In particular, the communities of the aforementioned three tools offer 14,000+ artifacts developed by 4,000+ contributors, serving hundreds and thousands of millions of downloads.

However, developing and maintaining high-quality IaC artifacts is challenging. Although IaC practices perform project lifecycle management in the way similar to the conventional software engineering, there are still many issues (particularly code errors) in IaC code repositories. Consequently, using the artifacts containing potential code errors would result in failures and even worse system crashes. Some efforts have been devoted to improving IaC code quality based on static verification [2] and automatic testing [1][3]. But even so, there are still many IaC code errors not addressed.

Inspired by the finding of the prior studies that, *similar (or identical) bugs would be fixed with similar (or identical) code changes* [4], we propose an approach to handling the fine-grained and frequently occurring IaC code errors. It first extracts *error-fix-induced* code changes from historical commits and clusters them into groups according to their features. Then error patterns are manually identified from the clusters, and according to which a set of rules are proposed to inspect the IaC code errors. In practice, we take Puppet code artifacts as subject objects and implement a prototype named *Puppet Analyzer*. *Puppet Analyzer* constructs a model to feature IaC code changes and uses an unsupervised machine learning algorithm, Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [5], to cluster the code changes. By empirically investigating these clustered code changes, we identify error patterns and propose a set of inspection rules for them. Consequently, *Puppet Analyzer* would automatically inspect the other IaC artifacts against the rules, to find potential code errors. Note that although this paper focuses on Puppet, the approach is general applicable for other configuration languages and IaC artifacts.

We evaluate our work with 14 popular Puppet artifacts. In our experiment, we get 41 cross-artifact error patterns, covering 42% crawled code changes. Based on these patterns, 30 constraint rules are proposed, covering 60% identified error patterns. They help to proactively inspect IaC artifacts.

In summary, the main contributions of this paper are: (1) an approach which identifies IaC code error patterns by mining code commits and checks fine-grained errors with proposed inspection rules; (2) a comprehensive study on the cross-project error patterns in some popular Puppet artifacts.

The rest of this paper is structured as follows. Section II briefly introduces IaC (in particular Puppet) and the motivation. Section III elaborates our approach. Section IV presents our experiments, comprehensive studies and some discussions. Section V surveys related work. Finally, Section VI summarizes this paper and discusses future work.

II. Background and Motivation

A. Infrastructure as Code

IaC specifies system configurations in an imperative or declarative way, and the specifications, in form of configuration code artifacts, are executed by the tools to achieve automation. Puppet, one of the most popular tools, determines how to reach the state by executing the IaC artifacts called *Puppet modules*.

[1] http://www.rightscale.com/lp/2016-state-of-the-cloud-report

978-1-5386-9443-5/18/$31.00 ©2018 IEEE
DOI 10.1109/ISSREW.2018.00-19

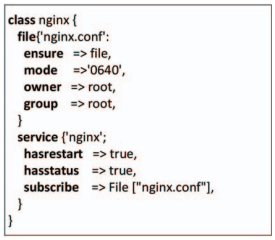

Fig. 1. A Puppet code snippet

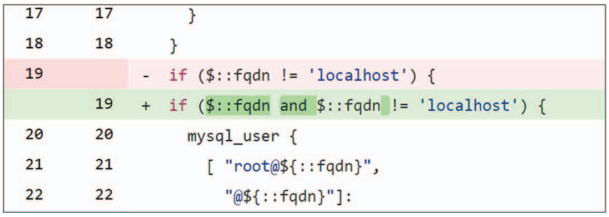

Fig. 2. A code change that fixes a null-value error

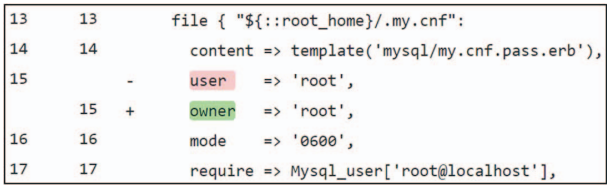

Fig. 3. An IaC code change example

Puppet modules, being self-contained bundles of code and data with the specific directory structures, are the basic building blocks for Puppet. In a Puppet module, `manifests` is the basic directory containing executable code (in `.pp` files). Besides, there are other directories named `files`, `templates`, `libs`, `spec`, etc. In this paper, we focus on those code files in directory `manifests` since they contain most of the IaC code of a Puppet module.

In Puppet language, `resources` are the fundamental units for modeling system configurations. Each resource describes some aspects of the managed system with their attributes, and the *resource declaration* is an expression that describes the desired state. The type of each resource determines what kind of configuration it manages. Puppet provides tens of built-in types of resources, including `user`, `group`, `package`, `file`, `service`, etc., and it also supports custom type definition. Puppet language has features in common with the conventional programming languages, such as multiple types of `variables`, `expressions`, `assignments`, and `conditionals`, and it has "object-oriented" feature which enables class definition and inheritance [6]. Figure 1 gives a code snippet that defines a class, which contains two resources `file` and `service`, automating the procedure of restarting the nginx service when its configuration (*nginx.conf*) updates.

B. Motivation

Similar to conventional Open Source software projects, the development lifecycle of Puppet modules - bug fixes, refactoring, and feature work - occurs within public code repositories, such as Github.

The prior work [7] has proved that the identical (or similar) bugs are often fixed in the identical (or similar) ways, resulting in similar code changes in or across projects. We are inspired to identify IaC error patterns from the code changes, and which would be helpful in improving IaC quality.

Figure 2 presents a code snippet which contains a change in a commit, where an error is fixed by adding a judgment statement in the original conditional (line 19), which checks if the variable `$::fqdn` is `undefined`. It can be inferred from this code change that, there might be a *null-value* error occuring in the original IaC code before this commit.

We find that there are many commits containing code changes that fix the errors similar to that in Fig. 2. As such, we would extract a *null-value* error pattern from the code commit set and inspect the other IaC repositories with this pattern. Thus, we are motivated to propose an approach to identifying cross-project error patterns for IaC, by mining the code commits and clustering the code changes.

III. Approach

Our approach comprises three steps, i.e., extracting code changes, identifying error patterns and proposing constraint rules. The details of each step are presented as follows

A. Extracting IaC Code Changes

1) IaC Code Changes: Besides error fixes, there are various root causes of code commits, such as adding new functions and refactoring artifacts. Compared with error fix, the other kinds of commits would involve adding and removing the whole code blocks, or updating many lines of code scattering in the various places. The prior work has proven that the frequent bug fixes always involve a few lines of code, and the proportion of bug fixes is not more than 5% when the changed code exceeds 10 lines [8]. As such, we focus on the changes involving only several contiguous lines of code (not exceeding a threshold), which most likely imply error fixes [7]. Figure 3 gives an example contains a single line code change, where the resource `file` is updated by removing an attribute `user` and adding another one `owner`. The change is caused by a misuse of the two different attributes.

We use AST (Abstract Syntax Tree) differencing [7] to locate code changes. Given an IaC code file, the approach constructs its ASTs of before and after a commit. It then compares the nodes in the same position by traversing the trees, to identify node insertions, removes and updates. Figure 4 gives an example of AST differencing based code change identification, which corresponds to the code change in Fig. 3.

2) Modeling IaC Code Changes: Change classification [9] is effective in discovering complex changes based on *basic change types (BCT)* [7]. We propose a set of BCTs to model the lowest level code changes. According to the syntax of Puppet language, a BCT is specified in four dimensions,

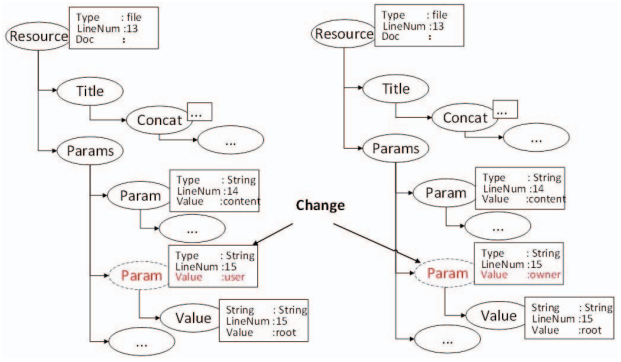

Fig. 4. An example of AST differencing based code change identification

TABLE I
SOME INSTANCES OF EACH DIMENSION OF BCT

dimension	representative
type	Function, Resource, IfStatement, CaseStatement, Relationship, Vardef, Selector, ...
context	File, Service, Node, Package, Cron, Exec, Include, Versioncmp, Test, Order, Case, Option, Var, FunctionName, ...
content	Argument, Title, Path, Name, Mode, Ensure, Onlyif, Unless, Enable Undefined, Variable, File, Notice, Value, ...
action	insert, remove, update

i.e., *type*, *context*, *content* and *action*. **Type** specifies the syntax types of BCT, e.g., *resource*, *function*, *conditional*, *variables*, *relationship*, etc. **Context** specifies the location of the change, such as *resource declaration*, *function definition*, *conditional*, etc. **Content** exactly specifies the code change, such as *attributes* in a resource and *parameters* in a function. **Action** describes how the code is changed, including *insert*, *remove* and *update*.

Table I lists some instances of the each dimension. An IaC code change can be modeled as a vector in which each element is a BCT. For example, the code change shown in Fig. 3 would be represented as `<resource-file-owner-insert, resource-file-user-remove>`

B. Clustering Code Changes

An IaC artifact is modeled as a set of code files $\{f_1, f_2, \ldots, f_n\}$, and for each file $f_i (1 \leq i \leq n)$ whose commits are represented as $\{c_{i,1}, c_{i,2}, \ldots, c_{i,m}\}$. A commit of a file $c_{i,j} (1 \leq j \leq m)$ would contain multiple BCT instances, $\{BCT_1, BCT_2, \ldots, BCT_p\}$. We then use a feature matrix $\mathbb{M}$ to model the code changes of an artifact. In the matrix, each column denotes a BCT and each row denotes a code change, and each element represents the occurrence times of the corresponding BCT.

We use HDBSCAN [5] algorithm, an extension to DBSCAN, to cluster code changes, for the reasons that (1) the number of clusters is unpredictable and (2) the shapes of clusters are agnostic. During clustering, we use Manhattan distance rather than Euclidean distance and cosine distance, to measure the similarity between code changes.

TABLE II
THE SUBJECT PUPPET CODE PROJECTS

	project	# of commits	# of downloads
1	puppetlabs/puppetlabs-apache	3,038	5,426,606
2	elastic/puppet-elasticsearch	1,956	576,227
3	voxpupuli/puppet-nginx	1,704	37,147,106
4	puppetlabs/puppetlabs-mysql	1,604	4,407,493
5	puppetlabs/puppetlabs-postgresql	1,523	4,597,108
6	puppetlabs/puppetlabs-firewall	1,273	4,422,729
7	puppetlabs/puppetlabs-apt	1,264	15,209,389
8	garethr/garethr-docker	1,057	2,632,417
9	voxpupuli/puppetlabs-rabbitmq	1,070	52,274
10	stackstorm/puppet-st2	988	6,043,249
11	puppetlabs/puppetlabs-concat	840	63,121,675
12	voxpupuli/puppet-python	740	11,348,767
13	elastic/puppet-logstash	701	351,630
14	voxpupuli/puppet-mongodb	682	565,783

C. Identifying Error Patterns and Generating Constraints

We investigate the clusters and identify error patterns from the code changes. According to the patterns, we propose rules to constrain value type, format and other aspects of IaC artifacts. More details are presented in Section IV-C and IV-D, with Puppet code artifacts as the subject projects.

IV. EXPERIMENT AND EVALUATION

A. Experiment Setup

We crawled 14 popular and active Puppet code projects from GitHub, and their details are listed in Table II. Statistically, there are 15,000+ commits in total, and each project has 500+ commits on average (49 commits on average between Sep. 2017 and Feb. 2018), serving 1,400,000+ downloads.

We filtered out commits of irrelevant files and code changes. We first remove those commits in files are not `.pp` ones, and then we discard those changes involving more than 10 lines of code in the remaining files. Finally, we got 1,980 valid code changes from the 14 projects.

We first make the statistics of extracted BCTs. Then we extract patterns from the 14 Puppet modules and make a comprehensive study on them. Besides, we propose constraints correspondent to those error patterns.

B. Statistics of Extracted BCTs

We extracted 3,099 BCT instances from the valid code changes, and among which there are 642 distinct ones. Statistically, the number of BCT instances contained in each code change ranges from 1 to 10, accounting for 1.56 instance per change on average. Figure 5 presents the top 10 BCTs that occur most frequently, in particular, among which `ifstatement_if_test_update` occurs over 300 times, which implies that incorrect `if-condition` is common.

We investigated the BCT instances on their syntax types. As Fig. 6 shows, a large portion of code changes relate to `resource`, which is coincident with the feature of Puppet language of taking `resources` as the fundamental units. We then drilled down into the distributions of resource types and found that the most popular built-in resources include `file` (746), `exec` (345), `package` (61) and `service` (55),

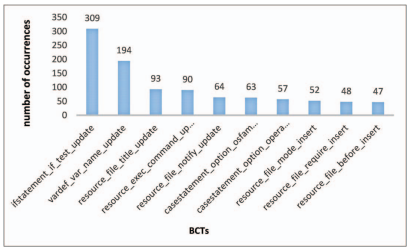

Fig. 5. A statistical result of the BCTs occurring most frequently

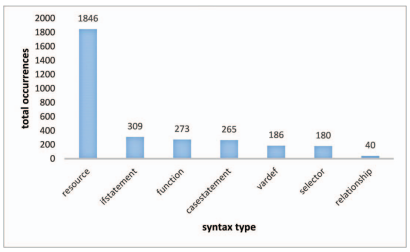

Fig. 6. A statistics of the distributions of BCT types

which because Puppet is a configuration management tool responsible for downloading and installing software, managing configuration files and launching/stopping services.

We also performed statistics of the actions. `update` accounts for nearly 2/3 code changes, in contrast, `insert` and `remove` are much fewer, accounting for 26% (983) and 9% (303) respectively. Since we focus on *error-fix-induced* code changes by filtering out those ones might relate to code refactoring and new feature addition, the majority of remaining code changes are introduced in by `update` actions.

C. Clustering Code Changes

Based on HDBSCAN algorithm [5], we initially get 85 clusters, where we empirically set the minimal number of instances in a cluster with 5. We then manually investigate these clusters and extracted 41 valid ones that contain explainable code changes.

As Table III lists, we classified the 41 cross-project error patterns into 9 categories according to their semantics. We made a comprehensive study on these patterns, and some representatives are introduced as follows.

Variable related error. The error patterns of this category manifest as *undefined variable*, *null-value reference*, *incorrect variable type or value*, etc. In our experiment, the errors of this category account for 12%. Puppet language supports two kinds of variables: (1) *facts and built-in variables* and (2) *user-defined ones*. Dissimilar to the first kind of variables which are *pre-set* with system information and are directly used, the latter kind of ones must be defined before use. Similar to the conventional programming languages, if the variables are undefined or their value types are not expected ones, there would be exceptions in IaC execution. The fixes to such errors include: (1) adding variable definitions before references, or (2) adding judgment statements before uses. The BCTs involved in these error patterns are `vardef_var_name_insert` and `ifstatement_if_test_insert`. Figure 7(a) shows an example of fixing the error of undefined variable `$provider`.

File related error. `File`, being a built-in type of resource, is the fundamental unit for managing *files*, *directories* and *symlinks*. In our experiment, this kind of errors account for 20%. Some error patterns manifest as follows. (1) **Misuses of attributes**, for example, `path` and `name` are two attributes respectively denoting file path and name, and there are many errors suffer from the misuses of them, which involve BCTs `resource_file_path_insert` and `resource_file_name_remove`. (2) **Incorrect attribute value**, which can be subdivided into **incorrect type** and **incorrect content**. For example, attribute `mode`, specifying the authority of a file, whose value is a four-digit string rather than a number. The involved BCT is `resource_file_mode_update`. (3) **Invalid state**, attribute `ensure` declares the expected file state, whose value must be one of *absent*, *present*, *file*, *directory* and *link*. The involved BCT is `resource_file_ensure_update`. Figure 7(b) shows a fix to the error of attribute `mode`.

Operating system (OS) related error. OS is an important factor affecting configuration management. In a Puppet artifact there might be IaC code correspondents to various types of OSs. A judgment to the OS type is necessary to decide which code path should be chosen. For example, the statements relevant to yum would be executed when the OS is CentOS, while apt related code would be chosen if the OS is Debian. The patterns manifest as *missing type judgement statement*, *misuse of facts* (e.g., `operatingsystem` and `osfamily`) and so on. This kind of errors account for 6% of the total, and the fixes to such errors are adding or updating the judgment statements to OS type. Some BCTs involved in are `casestatement_option_operatingsystem_add`, `casestatement_option_osfamily_add`, `selector_value_operatingsystem_add` and `selector_value_osfamily_add`. Figure 7(c) shows an example, where the `case statements` are added to make a judgement of the fact `operatingsystemrelease`, in order to execute the appropriate code blocks.

Function call related error. Puppet includes multiple built-in functions, particularly in `puppetlabs-stdlib`. The evolutions of the language and the built-in functions result in incorrect function calls and invalid parameters. Such errors account for 5% in our experiment. Some involved BCT is `function_{functionName}_param_modify`, where `{functionName}` represents a specific function. Figure 7(d) gives an example of fixing the error of missing parameter of function `validate_bool`.

D. Rule-based IaC Inspection

We specify rules based on a parameterized template. Figure 8(a) presents the JSON format template, where there are several elements. (1) `Syntax`, with `$type` as the value, specifies which type of code should be inspected; (2) `position`, with `$context` as the value, specifies where the target code

TABLE III
CATEGORIZATION OF ERROR PATTERNS

	category	semantics of pattern
1	file related error	incorrect attribute value, invalid path, missing name, misuse of attributes, missing group, missing notification, missing dependencies, incorrect content, invalid state
2	variable related error	incorrect reference, duplicate definition, null value, incorrect assignment, variable undefined, context error
3	OS related error	missing judgment, statement mismatch, incorrect environment variable, missing selector
4	exec related error	missing dependency, tittle conflict, path not found, incorrect conditional
5	function call error	incorrect function call, invalid parameters, regular expression error
6	relationship error	missing declaration, incorrect ordering
7	package related error	missing installation package, incorrect package name, misuse of title and name
8	class related error	missing parameter, duplicate definition, class inheritance error
9	selector related error	missing options, incorrect condition setting

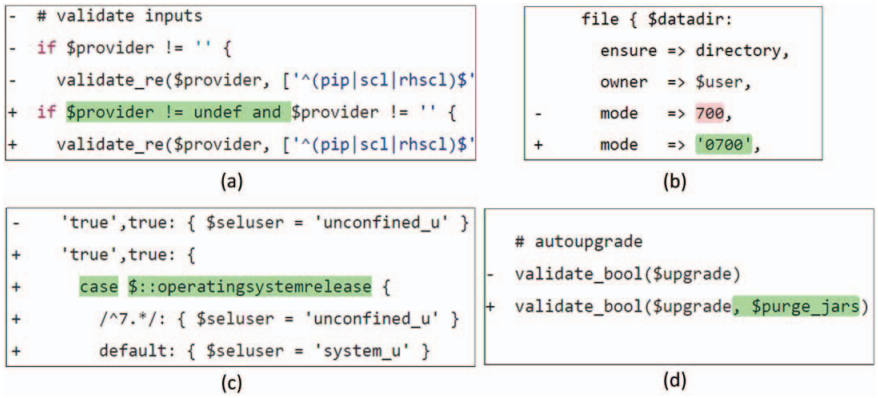

Fig. 7. Some examples of the representative error patterns

block is; (3) `attribute`, with `$content` as the value, specifies what should be inspected; (4) `msg`, with `$message` as the value, provides the message if an error occurs; (5) `rule`, parameterized with `$constraint`, specifies how to inspect the target code.

With the identified error patterns, we propose a set of rules based on the template. Figure 8(b) gives a rule instance, which checks attribute `mode` of resource `file`. The rule specifies the constraint on the value format with a regular expression. We currently proposed 30 rules, which cover 60% identified error patterns that relate to various types of resources. Note that the template is extensible to propose more inspection rules. Given a Puppet code file, the approach loads the checking rules and transforms the code into an AST. It then iterates each node in the AST and checks it against the corresponding rules. If the constraint is not satisfied, the error message would be logged. Finally, an error report would be generated from of the logs.

We implemented a prototype named *Puppet Analyzer* by integrating puppet-lint[2], Puppet Code Validator[3] and our approach. Figure 9 presents a case study, where the main area lists the checking result of file `init.pp` in the Puppet module `single-hadoop`. The report contains three parts: (1) the first part is syntax validation, the message shows that there is a syntax error; (2) the second part is code style checking

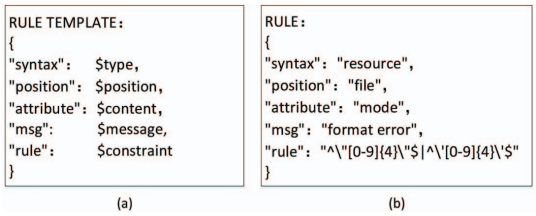

Fig. 8. Rule template and an instance

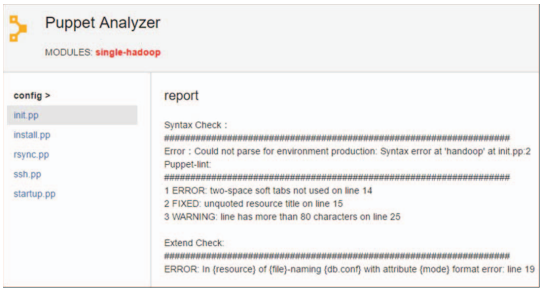

Fig. 9. The checking result of a case study

result, which denotes there are some code style violations; (3) the third part is constraints validation, the message hints there is a format error of `file` attribute `mode` at line 19.

[2]https://github.com/rodjek/puppet-lint
[3]https://https://validate.puppet.com

E. Discussion

First of all, the rationale of our approach is "*the identical (or similar) code errors would be fixed with the identical (or similar) code changes*", nevertheless, it does not hold for all of the code errors. For example, some code errors, especially those relevant to specifying resource dependencies, would be fixed in various ways, including using attributes `notify` or `require`, or additionally specifying a relation between the resources. Although the code errors are semantically similar, the different code changes hinder the pattern identification.

Secondly, our approach only considers the code changes not more than 10 lines as potential error fixes. It is too rigid to miss some error patterns, affecting the efficiency. Although the rule originates from the previous empirical study [8], we would refine the grain of error code in future work.

Thirdly, we currently only take the single code file as a unit and do not consider the correlations between multiple code files. In practices, an error fix might cross several code files, and our approach cannot identify such patterns.

The last but not the least, the language evolves over time. Our work is based on Puppet 4.x, while the latest language version is 5.x. As such, some specific properties featuring the code changes would not be applicable any more. However, our approach is general applicable, and in practices the feature model is adaptable for the specific version of language.

V. Related Work

Some efforts have been devoted to improving IaC code quality. Rehearsal [2] verifies Puppet manifests to determine whether the code is deterministic and idempotent. A black-box approach [1] is proposed to test Chef cookbooks for their idempotence, by systematically executing the actions specified by the code. A model-based testing framework determines whether a system configuration converges to a stable state [3]. Puppeteer [10] analyzes 4,621 Puppet repositories and proposes a catalog of 13 implementation and 11 design configuration smells violating the recommended best practices. Our work is different to them, it focuses on the fine-grained IaC code errors rather than the violations of high-level properties (e.g., idempotence, convergence and determinacy) and code styles.

Mining bug patterns is based on either bug reports [11][12] or code changes [7]. Li et. al. group bugs by classifying bug reports with natural language processing technique [11]. Ocariza et al. [12] manually inspect bug reports for JavaScript code and find that the majority of reported bugs on the client-side are DOM-related. BugAID [7] discovers server-side JavaScript bug patterns by using language construct-based changes. Our work is similar to BugAID in identifying the patterns by clustering code changes according to their features. However, our work focuses on IaC code errors and identifies 41 patterns from 14 Puppet code artifacts.

There are many empirical studies on code changes and bug fixes [13][14][15], and our work is based on their findings and conclusions. In our comprehensive study on the identified IaC error patterns, we find some patterns are similar to the previous work, e.g., null-value reference and conditional related ones. We also find the error patterns specific to Puppet language syntax and construct.

VI. Conclusion and Future Work

This paper proposes an approach to handling fine-grained and frequently occurring IaC errors, by identifying error patterns. It extracts *error-fix-induced* code changes from historical commits and clusters them to identify error patterns. With this approach, we make a comprehensive study on 14 popular Puppet artifacts and identify 41 cross-artifact error patterns. We also propose 30 constraints to inspect the potential IaC code errors. In future work, we would evaluate the approach further by applying it to much more IaC artifacts. Moreover, we would exploit more patterns and inspection rules.

Acknowledgment

The authors would like to thank the comments of the reviewers. This work was partially supported by the National Key R&D Program of China under Grant No.2016YFB1000803, the National Natural Science Foundation of China under Grant No.61572480.

References

[1] W. Hummer, F. Rosenberg, and et al., "Testing idempotence for infrastructure as code," in *Middleware*, 2013, pp. 368–388.

[2] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *PLDI*, 2016, pp. 416–430.

[3] O. Hanappi, W. Hummer, and S. Dustdar, "Asserting reliable convergence for configuration management scripts," in *OOPSLA*, 2016.

[4] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298–312, 2016.

[5] R. J. Campello, D. Moulavi, and et al., "Hierarchical density estimates for data clustering, visualization, and outlier detection," *ACM Transactions on Knowledge Discovery from Data*, vol. 10, no. 1, p. 5, 2015.

[6] W. Fu, R. Perera, and et al., "μpuppet: A declarative subset of the puppet configuration language," in *31st ECOOP*, 2017.

[7] Q. Hanam, F. S. d. M. Brito, and et al., "Discovering bug patterns in javascript," in *Proceedings of the 24th ACM SIGSOFT International Symposium on FSE*. ACM, 2016, pp. 144–156.

[8] H. A. Nguyen, A. T. Nguyen, and et al., "A study of repetitiveness of code changes in software evolution," in *Automated Software Engineering*. IEEE, 2013, pp. 180–190.

[9] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *ICPC. 14th IEEE International Conference on*. IEEE, 2006, pp. 35–45.

[10] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 189–200.

[11] Z. Li, L. Tan, and et al., "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 25–33.

[12] F. S. Ocariza, K. Bajaj, and et al., "A study of causes and consequences of client-side javascript bugs," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 128–144, 2017.

[13] J. Park, M. Kim, B. Ray, and et al., "An empirical study of supplementary bug fixes," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 2012, pp. 40–49.

[14] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on FSE*. ACM, 2012, p. 53.

[15] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 422–432.