How Good Is Your Puppet? An Empirically Defined and Validated Quality Model for Puppet

Eduard van der Bent
and Jurriaan Hage
Utrecht University, the Netherlands
J.Hage@uu.nl, eduardvdbent@gmail.com

Joost Visser
Software Improvement Group
Amsterdam, the Netherlands
j.visser@sig.eu

Georgios Gousios
TU Delft, Delft, the Netherlands
G.Gousios@tudelft.nl

***Abstract*—Puppet is a declarative language for configuration management that has rapidly gained popularity in recent years. Numerous organizations now rely on Puppet code for deploying their software systems onto cloud infrastructures. In this paper we provide a definition of code quality for Puppet code and an automated technique for measuring and rating Puppet code quality. To this end, we first explore the notion of code quality as it applies to Puppet code by performing a survey among Puppet developers. Second, we develop a measurement model for the maintainability aspect of Puppet code quality. To arrive at this measurement model, we derive appropriate quality metrics from our survey results and from existing software quality models. We implemented the Puppet code quality model in a software analysis tool. We validate our definition of Puppet code quality and the measurement model by a structured interview with Puppet experts and by comparing the tool results with quality judgments of those experts. The validation shows that the measurement model and tool provide quality judgments of Puppet code that closely match the judgments of experts. Also, the experts deem the model appropriate and usable in practice. The Software Improvement Group (SIG) has started using the model in its consultancy practice.**

I. Introduction

In recent years, cloud computing has become increasingly popular [26]. In many cases, deployment of software systems in the cloud entails the use of a configuration management language, such as Chef [7], Ansible [3], or Puppet [18]. These languages allow software developers and/or system administrators to specify the infrastructure needed to run an application and how it should be deployed. Although the specifications can be considered executable code, they are typically very different from ordinary programming languages. As a result, practitioners cannot simply apply existing notions of code quality to these new languages.

The languages Chef and Ansible are imperatively styled, which means that the developer specifies the steps that should be taken to get a host to the required configuration. Puppet on the other hand is declarative: one specifies the required state of the server, and then the Puppet tool figures out how to reach it. Various comparisons between these three languages have been made by others [17], [24], [9].

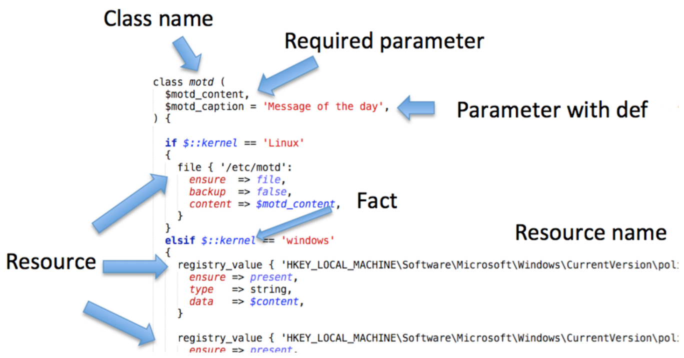
Server configuration specifications written in Puppet are developed, executed, and maintained like code in traditional programming languages. As there are risks in having to maintain ordinary code, one can imagine the issue of Puppet code quality to be relevant for organisations that have a lot of Puppet code in their portfolio (see, e.g., [29] and [31]). A lot of work has been done on measuring and managing code quality for (general-purpose) programming languages, but not for configuration management languages like Puppet.

The major contribution of the paper is an expert-validated quality model for Puppet that provides a rating of the quality of a repository of Puppet code. This rating is based on a number of metrics, as determined from a survey among Puppet developers (Section IV) and that can be computed automatically for Puppet source code (Section V). The scores for the various metrics are combined into a rating and calibrated against a benchmark set of repositories in Section VI. We validate our ratings with experts (Section VII).

II. A Short Introduction to Puppet

In this section we describe the basics of Puppet; more comprehensive information can be found in [20]. In Puppet, the basic building block is called a *resource*. Each resource has a resource type. Common examples are files, packages and service resource types; additional custom resource types can be defined. In Puppet, a *class* is simply a collection of resources. Figure 1 displays a (slightly modified) Puppetlabs `motd` class [16] that configures the message of the day for a system. It has two parameters: `$motd_content` is a required parameter, and `$motd_caption` is a parameter with a default value. Three resources are defined in this example: the file resource is set for Linux-based systems; for Windows, we use two registry values.

Facts in Puppet are system-specific variables and constants collected by the Facter tool [10]. Facts can be used in Puppet code like any other variable. Built-in facts in Puppet offer a powerful way to get system information, since an implementation is already provided for the user. *Generic facts* are OS-independent. Examples of OS-specific facts are `macosx_buildversion` and `solaris_zones`. In Figure 1, the built-in fact `$::kernel` is used to determine the kind of operating system.



```
class motd (
  $motd_content,
  $motd_caption = 'Message of the day',
) {

  if $::kernel == 'Linux'
  {
    file { '/etc/motd':
      ensure  => file,
      backup  => false,
      content => $motd_content,
    }
  }
  elsif $::kernel == 'windows'
  {
    registry_value { 'HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\pol
      ensure => present,
      type   => string,
      data   => $content,
    }

    registry_value { 'HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\pol
      ensure => present,
```

Fig. 1: An example of a class in Puppet.

An *exec* is a particular, important kind of predefined resource type, and is essentially a shell command run from Puppet. The following exec updates the list of packages for the Cabal package manager:

```
exec { 'cabal update' :
  user          => $user,
  environment   => "HOME=$home",
  path          => ['/bin', ...],
}
```

Puppet *modules* typically configure a single application or service. A module consists of files to configure a service, including tests, a README, templates in embedded Ruby, custom types and providers, and custom facts.

Templates are either Embedded Ruby or Embedded Puppet files, that can be run by providing parameters to them. This is useful when a configuration file is dependent on run-time constants, such as the number of cores or the amount of available memory. Custom types and providers are custom resources, written in Ruby. The type provides the interface, the provider contains the implementation.

At top level, Puppet runs on a *node*, a device managed by Puppet. The following is an example node definition that simply instantiates the motd class of before:

```
node 'test' {
  class { 'motd' :
    motd_content => 'Welcome to Section II'
  }
}
```

III. Motivation for Our Work

The form of our quality model for Puppet is inspired by existing quality models for rating the maintainability of general purpose programming languages [14]. First, a set of metrics is defined to measure various aspects of maintainability, like file length and amount of code duplication. The metrics are applied to each unit (what a unit is, depends on the language). Then for every metric, risk profiles are defined, telling us what metric scores are considered medium risk, which imply a high risk, etc. Risk profiles are turned into a 5 star rating by specifying for a given rating what percentage of a given system may fall into each of the risk profiles. For example, to achieve a 5-star rating for file length, there may be no code with very high risk, at most 3 percent may be high risk, and at most 30 percent may be of medium risk. Typically, these categories are defined based on a large set of real world source code repositories. Finally the rating for the various metrics are combined into a single rating, e.g., by taking the average.

Which metrics to employ and how exactly we should combine measurements for those metrics into ratings is likely to be substantially different for Puppet as compared to a general purpose language. For example, existing models typically include unit complexity where the definition of a unit corresponds to the smallest executable piece of a program, e.g., a method. But how does that translate to Puppet? It can be argued that the units in Puppet are resources, since these are the smallest executable pieces of code [22]. However in Puppet, it is considered bad practice to put any kind of complexity inside resources [23]. Furthermore, unit testing is done at the class level in Puppet. But classes in Puppet are often much longer than methods in a general purpose language, so we would need to consider new thresholds for computing our ratings for Puppet as compared to, say, Java.

Puppet also has other properties that are not captured by existing models, like how well is data separated from code? Most data in Puppet should go into Hiera — the key-value lookup tool that comes with Puppet —, but , e.g., parameter defaults are put directly in the code.

IV. A Survey on Code Quality among Puppet Developers

So, what makes Puppet code of low, or high quality? We answer that question by asking Puppet developers by means of a questionnaire. Below we describe our target population, and the survey protocol we follow.

Population: As have others, we tap into the Puppet community through GitHub [30], the currently most popular web-based code repository service. To determine the survey population, we used GHTorrent [11] to find all users that had more than 10 commits in at least 2 repositories each that were labeled by the GitHub API as containing Puppet, and that were pushed in the last six months. In addition to this, we also required the text "Puppet" to appear in the repository clone URL, to increase the likelihood that Puppet programming was part of the focus of the repository, and that non-trival Puppet development has taken place.

To establish this, we joined the tables of contributors and users to add email addresses to the result (Users without emails cannot be reached, so they are excluded). We joined this table with the table of repositories, and then filtered out all contributors with 10 or fewer commits. Then, every contributor that appeared more than once was selected for the survey population.

The Construction of the Survey: Given the absence of established theory or literature on Puppet code quality, we decided to aim for a survey that is explorative in nature. Therefore, we opted for a short list of open questions, in order to reach a sizable audience and thereby increase our coverage. We created an initial survey with which we carried out a pilot. The results led us to make considerable adjustments to the survey. This new survey was piloted twice before proceeding to carry out the survey at scale.

To determine the questions of the survey, a large list of candidate questions that might be relevant was created. Then we discussed the questions with various experts in the fields of software engineering, empirical studies, and Puppet and removed questions that would invite vague answers, questions that were difficult to understand and questions whose answers would be irrelevant to our research.

The questions in this initial survey were rather generic. We found that people familiar with maintainability gave useful answers, but in a pilot among five internal Puppet programmers and in our first external try-out, we found that generic questions, like "What are typical issues you encounter when reading Puppet code?", were too unfocused. In the end we kept only 4 of our 10 questions, and added a number of questions about Puppet software quality. This is the final list of questions that we sent out to 257 Puppet programmers:
- How many years of experience do you have with programming Puppet?
- If you had to judge a piece of Puppet code (file or module) on its quality what would you look for?
- Could you give some examples of bad practices in Puppet code?
- What aspects do you take into account when selecting a Puppet module from GitHub or Puppet Forge?
- How do you decide what goes into Hiera and what doesn't?
- What is your biggest problem when programming Puppet?

Analysis: Out of 257 we got 42 responses, a rate of about 16%. The results of the survey have been published online [5].

The responses to the first survey question show that our population ranges from new to very experienced Puppet developers: 11 participants had 5 years of experience or more, 13 had 3-4 years, 15 had 1-2 years, and 3 had less than one year of experience.

In the remainder of this section, we briefly discuss the answers to the other questions of our survey. We analyzed the answers to questions 2 to 6 by coding the answers, following the process specified in Gousios et al. [12]. We required at least two responses for a code where possible, and tried to group unique responses together in a more general category. Since some answers consisted of long enumerations of quality aspects, there was no upper bound on the number of codes a response could get. The graphs showing the codes and their frequencies are included in Appendix A of [4]. The dataset and code table are available online [6].

If you had to judge a piece of Puppet code (file or module) on its quality, what would you look for? Looking at the top 5, code quality in Puppet is not very different from normal programming languages. Testing, documentation, code style, design and linting are common concepts in general purpose programming languages as well. More interesting is the *program structure* — examples here are proper usage of the Package, Config, Service design pattern, and making use of a params class to set defaults. The *exec resource* is also frequently mentioned and specific to Puppet: usage of execs should be kept to a minimum.

Less frequently mentioned categories include *readability*, *complexity* and *dependencies* (common also for general purpose programming languages) with Puppet-specific categories including the correct use of parameters, custom types, being cross-platform, and behavior. Note that although these were less frequently mentioned, this does not mean we disregarded them when constructing our model.

Could you give some examples of bad practices in Puppet code? The *improper usage of execs* was prominent here. Other bad practices are a *lack of program structure* (no PCS / params.pp for instance, or having very large files), *hardcoded variables* (that should have been parameters), *having too many dependencies* (or too few), *non-idempotence* (a catalog applied multiple times should not result in changes being made every run) and having *too many or too few parameters*. Also failing to use Hiera when this is needed is considered bad practice. Non-Puppet specific bad practices are lack of documentation, complex code, and lack of tests.

What aspects do you take into account when selecting a Puppet module from GitHub or Puppet Forge? The dominant anwers to this question are not Puppet-specific: programmers look for signs of *recent activity*, *documentation*, *popularity*, the *reputation of the author*, and *functionality* followed by the amount of testing, support of platform, and the Puppet version that is used. Only four respondents said they actually look at the Puppet code itself.

How do you decide what goes into Hiera and what does not? Most responses indicate that data is stored in Hiera and code is in Puppet, but it appears there is a difference in opinion on what is data and code. Some people put defaults in Puppet, but others put them in Hiera (module-data). A recurring pattern seems to be that varying data, such as data that is company, environment or site specific should go into Hiera.

In other cases, less disagreement is evident. Operating system-specific information should not go in Hiera. An example of such a variable is the package name for the Apache application. In Debian, it is simply called *apache*, but in RedHat it is called *httpd*. The best practice is to have these differences in the code, and not in Hiera.

What is your biggest problem when programming Puppet? The responses here indicate that different respondents have different problems with Puppet. Problems were therefore grouped by us based on their commonalities.

Most reported problems are related to the way that Puppet itself is designed. Examples include *snowflake servers*, the *limitations of having a single node*, and *nondeterminism*.

Problems related to third-party modules are also a common theme, leading to developers to apply fixes to third party code or switching to a different supplier. The poor quality of third-party modules was also mentioned.

Other examples of problems include the lack of high quality tests, documentation for the types and providers API, functionality that respondents would like Puppet to have, and the quality of error messages. Finally, a few respondents mentioned a general lack of best practices.

V. The Puppet Quality Model

A. The Metrics We Started With

Taking the answers from the survey as a starting point, the following metrics were considered for inclusion in the quality model:

- File Length: having all code in a single class was mentioned in the survey, as well as the number of lines. Having too much code inside a single file (which according to best practices, should only contain one class [21]) is an indication that too much is going on in the file to understand it.
- Complexity: complexity, simplicity and if-statements were mentioned in the survey results, so we will measure the complexity. This is measured at the file level.
- Number of exec statements: an exec is a shell command in Puppet, and best practice suggests to minimize these.
- Warnings and errors from Puppet-lint [19]: Puppet-lint checks for style violations and code that may easily lead to errors (like quoted booleans). We decided to omit warnings and errors that we believed to be *purely* style issues. That left only two kinds of errors to consider and since these turned out to be quite rare in our dataset, we only looked at the warnings (see Figure 2 for the list).
- Number of resources: resources are the basic building blocks of Puppet, so measuring resources could be an indication that a file has too many responsibilities. Since best practice with execs is to use the proper (custom) resource, we also measure the number of custom resources.
- Dependencies: Puppet offers many different types of dependencies. Some are across different classes, such as a class including or referring to another class. Dependencies may also exist between resources, such as a resource with a metaparameter requiring it to be applied after a certain other resource. Dependencies can be between files or within files. We measure for each file both fan-in, fan-out, and internally between components in the same file.
- Relations between resources: these can occur in the form of arrows or metaparameters. In Puppet, relations specify the ordering of operations.
- Parameters and variables: since it is difficult to measure whether a variable or parameter is actually useful, we have decided to measure how many variables / parameters per file are present. Since parameters can have defaults, we also measure how many required parameters there are: parameters for which a value must be supplied when calling the class.
- Hardcodes: we do not distinguish between harmless hardcodes like ”root” and harmful ones like hardcoded ip-addresses, email addresses, passwords, certificates and private keys.

single quoted string containing a variable found
”foo::bar not in autoload module layout”

quoted boolean value found
string containing only a variable
variable not enclosed in
ensure found on line but it’s not the first attribute
mode should be represented as a 4 digit octal value or symbolic mode
unquoted resource title
case statement without a default case
selector inside resource block
not in autoload module layout
class defined inside a class
class inherits across module namespaces
right-to-left (<-) relationship
top-scope variable being used without explicit namespace
class not documented
defined type not documented

Fig. 2: Relevant errors (above the double line) and warnings. In the end only the warnings were counted.

We added to this list the following well-known metrics: volume (size of the code base), code duplication and file-level fan-in (aka module coupling). Since the characteristics of Puppet seem quite different from general purpose languages to which these metrics have been applied, we will be looking for new thresholds. For example, a high-volume Puppet codebase might be very small when compared to the average Java system.

B. From Metrics to Quality Model

To obtain our quality model, we implemented the proposed metrics and applied them to a large number of Puppet repositories obtained from GitHub. The process to obtain these repositories was as follows: using the GitHub search API we collected the clone URLs of all Puppet repositories on GitHub (16.139 at the time of collecting). On February 1, 2016, we cloned these Puppet repositories, except repositories that were removed or inaccessible. We used shallow cloning and cloned recursively where possible — some submodules were in private repositories and could not be reached.

The GitHub dataset contained repositories that were marked as Puppet code (because of the .pp extension), but that were not actually Puppet code, e.g., Pascal and PowerPoint files. Whether we dealt with real Puppet code or not was discovered by running Puppet-lint: all files that gave parse errors were removed from our dataset. After this clean-up our dataset consisted of 15.540 repositories with Puppet code.

We computed the pair-wise Spearman correlation between the various implemented metrics for our dataset as shown in Figure 3 and Figure 4. Now, if we have two highly-correlated metrics, we can drop one in favour of the other. For example, both the number of resources and the number of hardcoded values are strongly correlated with file length, so we dropped the former two in favour of the latter.

Moreover, we decided to omit metrics that apply to only a small percentage of our dataset. For example, the number of lint errors was low overall, so we dropped the metric from our list. The same applied to having global defaults, although that was not the only reason for dropping it: in some situation having global defaults is fine (inside the manifest, as remarked upon by one of our interviewees), and in other cases it is not.

Only very few Puppet repositories had more than one class per file. Therefore, it makes more sense to flag having more than one as a violation, instead of using this information as part of our quality model. The number of variables metric turned out to behave erratically: the majority of the files did not contain any variables. Also, we have seen one Puppet repository with 200 arrows (an *arrow* in Puppet creates relationships between resources), but in most cases the number of arrows was small, so we decided to omit this metric too.

We opted to use file fan-in instead of other metrics such as fan-out, the number of metaparameters, and the number of internal calls. Having a high fan-out, a large number of metaparameters, and/or internal calls typically means that the file itself is already too large, so we consider measuring the filelength an adequate measure. The fan-out of modules is also measured as part of the module degree metric.

The final model consisted of the metrics shown in Table 5, together with, for reasons of space, only a short description of how exactly they are measured. Although we have removed some of the Puppet specific metrics, in favor of keeping more generic ones, note that this is only because we believe that this choice does not affect the *star ratings*. Our tool still supports the other metrics to show in more detail what might be the cause for obtaining a low rating.

VI. The Tool and Its Calibration

A. The Tool Implementation

We implemented the metrics by extending Puppet-lint to generate our measurements, and wrote a custom program to aggregate the measurements and to draw dependency/call graphs.

We adapted Puppet-lint to automatically traverse an entire folder structure and check every `.pp` file, but we excluded all filepaths that contained folders called `spec`, `tests`, `files` and `examples`, since this is either test code or code that skews the metrics. For example, resources are often declared outside of classes in files in the `examples` directory, but since this is not production code we left it out of our measurements. We also included a duplicated block detector in our tool.

Now that we have a reasonable set of implemented metrics, we want to turn the measurements we make for a given application into a star-rating that is more easily communicable. This process is called calibration.

B. Establishing the Benchmark

The configurations we selected for our benchmark were taken from the original dataset of Section V. We took repositories that had at least 5000 lines of Puppet code (not counting commentary, whitespace and lines containing only curly braces), and that had a ”modules” directory, which is for ordinary Puppet configuration the default location of the modules. Putting the limit lower than 5000 would strongly increase the number of Puppet system we would have to manually investigate to perform the fork detection, the calibration and the validation of Section VII, and we would run more risk of having a single dominant module that would skew the metrics. It also seems to us that it is more likely for maintainers of a large Puppet system to want to diagnose the maintainability of their Puppet code, than those maintaining a small system.

This left us with 299 repositories (out of 15.540, of which more than 14.000 were under 1000 lines). Since we cloned every GitHub repository, we may have cloned a lot of forks (or identical repositories not marked as such). We wanted to be reasonably sure that we were left with a number of *unique* repositories, and therefore computed the total lines of code, the total number of lines and the percentage of duplicatation of each repository. Repositories with numbers close together were considered to be the same, and we kept only one of each. At this point 100 repositories remained, of which we then deleted repositories that had large chunks of third-party code. Although third-party code is a risk of its own, we were not out to measure the quality of code that the owners should *not* maintain.

We were then left with 26 repositories. We manually reviewed each repository, and removed a few more duplicates and repositories that were not configurations, but simply large collections of scripts. Finally, this resulted in 17 repositories.

C. The Difference between Third-party Code and Own Code

One of the observations we made in our study is that code from third-party code is vastly different from the code that has been written for ”own use”.

Unsurprisingly, third-party modules try to handle every use case, support more Operating Systems and have more parameters / options. For all assumed non-third party modules we have drawn quantiles of filesize, complexity and parameters, and we have done the same for all Puppet approved and supported modules that contain Puppet code. These quantiles are shown in Figure 6. We can conclude from these numbers that generally most non-third party modules are smaller, have fewer parameters and are less complex than third party modules. On the other hand, there are non-third party modules that are much larger, have more parameters and are more complex than third party modules.

D. Rating Repositories

To turn the metrics into ratings, we made use of risk profiles [2]. With risk profiles, it is possible to rate how much of a given code base falls into either the low risk, medium risk, high risk or very high risk category, and using these percentages for all systems, it is possible to make a comparison and give a quality rating. The advantage of risk profiles is that instead of raising an error saying ”this file is too long” after it passes a

	filelength	#resources	#exec	complexity	#warnings	#errors	#parameters	#arrows
filelength		0.80	0.29	0.52	0.25	0.22	0.52	0.25
#resources	0.80		0.29	0.33	0.21	0.25	0.32	0.21
#exec	0.29	0.29		0.21	0.14	0.09	0.18	0.13
complexity	0.52	0.33	0.21		0.24	0.14	0.40	0.22
#warnings	0.25	0.21	0.14	0.24		0.31	0.05	0.16
#errors	0.22	0.25	0.09	0.14	0.31		NA	NA
#parameters	0.52	0.32	0.18	0.40	0.05	NA		0.18
#arrows	0.25	0.21	0.13	0.22	0.16	NA	0.18	
#globaldefaults	0.06	0.04	NA	0.06	NA	0.05	NA	0.09
#classesperfile	0.30	0.32	0.13	0.20	0.30	0.62	0.17	NA
#hardcodes	0.90	0.81	0.25	0.40	0.18	0.22	0.34	0.20
#requiredparams	0.26	0.16	0.11	0.17	0.10	0.05	0.58	0.10
fanout	0.43	0.44	NA	0.14	0.07	0.17	0.18	0.11
fanin	0.23	0.15	0.14	0.21	0.09	-0.06	0.16	0.09
internal	0.52	0.56	0.37	0.23	0.13	0.25	0.25	NA

Fig. 3: First half of Spearman correlations. Correlations with P-values higher than 0.01 are listed as NA.

	#globaldefaults	#classesperfile	#hardcodes	#requiredparams	fanout	fanin	internal
filelength	0.06	0.30	0.90	0.26	0.43	0.23	0.5
#resources	0.04	0.32	0.81	0.16	0.44	0.15	0.6
#exec	NA	0.13	0.25	0.11	NA	0.14	0.4
complexity	0.06	0.20	0.40	0.17	0.14	0.21	0.2
#warnings	NA	0.30	0.18	0.10	0.07	0.09	0.1
#errors	0.05	0.62	0.22	0.05	0.17	-0.06	0.2
#parameters	NA	0.17	0.34	0.58	0.18	0.16	0.3
#arrows	0.09	NA	0.20	0.10	0.11	0.09	NA
#globaldefaults		-0.05	0.06	0.00	0.00	NA	NA
#classesperfile	-0.05		0.27	0.14	0.12	0.17	0.3
#hardcodes	0.06	0.27		0.11	0.40	0.16	0.5
#requiredparams	0.00	0.14	0.11		0.10	0.12	0.1
fanout	0.00	0.12	0.40	0.10		NA	0.2
fanin	NA	0.17	0.16	0.12	NA		0.2
internal	NA	0.31	0.51	0.12	0.17	0.18	

Fig. 4: Second half of Spearman correlations of listed metrics.

certain threshold, which might appear hundreds of times when first analyzing a codebase, you can accurately see how severe the problem is — are the files just a bit too long, or are the files extremely large?

We derived Puppet-specific thresholds for the metrics in our model, similar to Alves et al. [1], and applied these thresholds. In our case, we opted for thresholds that were very different (widely spread) from each other. Where possible, we ensured that the threshold for medium/high risk was twice as large as low/medium risk, and high/very high risk was as least thrice as large as low/medium risk. Otherwise, a small measurement difference between two systems could, e.g., lead to one being assigned low risk, and the other high risk.

We then created a risk profile per metric by deriving the 70/80/90% values of a configuration: the 70 percent of all lowest scores mapped to low risk, the next 10 percent to medium risk, etc. Using this risk profile, we derive the thresholds to make the data fit a 5/30/30/30/5% distribution. So the 5% best systems get 5 stars, the next 30% of systems 4 stars, and so on. After fitting the systems to the 5/30/30/30/5% distribution, we derive the thresholds to get a certain star rating. Thus, if the code has exactly the same or less than the 5-star threshold, it will get 5 stars. If it has less than the 4-star threshold, but more than the 5-star threshold, the rating will be between 4 and 5.

An indicative example is the metric for measuring complexity. The risk profile we derived is shown in Figure 7a, and the associated thresholds for the star-ratings are shown in Figure 7b. These numbers imply that if at most 2.4% of the lines of code are in files of a ”medium risk” complexity, and no LOC within files of ”high risk” and ”very high risk”, then the given system belongs to the best 5% of observed systems. There is no limit on the amount of ”low risk” code that can be present in a configuration.

Measurement	Implementation
Filelength	The number of lines per file.
Complexity	The number of control statements per file, plus number of alternatives in case statements.
# of parameters	The number of parameters per file.
# of exec resources	The number of exec resources per file.
# of filtered lint warnings	The number of filtered lint warnings per file.
File fan-in	The number of incoming dependencies per file.
Module Degree	The number of incoming and outgoing number of dependencies per module.
Duplication	The number of redundant lines of a code base divided by the number of lines of a code base.
Volume	The number of lines of a code base.

Fig. 5: Our quality model measurements and their implementation.

Our second example is the Puppet-specific metric for execs. The chosen thresholds are given in Figure 7c and the thresholds for the various star-ratings are given in Figure 7d. Whilst the number of execs allowed per file may seem very low, the thresholds for the star rating shows that you are still allowed to use them sparingly without getting punished too harshly. The risk profiles and star-ratings for filelength are shown in Figure 7e and Figure 7f.

We computed the ratings for the number of parameters, the number of filtered lint warnings, fan-in and module degree in the same manner. For file length we used 25/50/75% instead of 70/80/90% to ensure that, in absolute numbers, the difference in size between a low risk and a high risk system was not too small. We rated volume by putting the systems in 5/30/30/30/5% bins, ordered by volume, and gave a rating based on that. Duplication was rated in a similar manner. The final rating for a system was computed as the average of the separate metric ratings. Due to lack of space, we do not include the risk profiles and star ratings for these metrics.

VII. Expert Validation

A. *Comparing the Outcomes of our Tool with Expert Rankings*

The quality model was validated with semi-structured interviews, with different groups of people. We interviewed several employees of the SIG that had Puppet knowledge, as well as a Puppet programmer from a different company, an external researcher with knowledge of Puppet, and two employees of the company behind Puppet.

The goal of the validation is to both see if the proposed quality metrics are good enough predictors of code quality, but also to see if there are potential flaws in the model. For example, to discover whether trying to improve on certain metrics will not (negatively) affect the maintainability of a system, or if certain metrics are simply not suited to the language or the domain of configuration management. It is also be possible that some metrics should be weighted more heavily (i.e. volume or a complex architecture) than is currently implemented.

The interview consisted of answering questions and making a comparison between systems. We split the 17 systems into 5 bins: one with the 20% worst quality systems, the second with the next 20%, etc. We then selected from every bin the most up-to-date system. We decided to use only 5 systems, because there is a limited amount of time available for every interviewee to rank the systems. The time scheduled for the interview was one hour. With 5 systems, we used on average 45 minutes for the comparison and 15 minutes for the questions.

Although our tool computes a rating for each system, we asked our experts to come up with a ranking (similar to Cox et al. [8]). The reason is that the interviewees were not all familiar with automated tooling to rate software systems. We then compared their ranking with the rating based ranking computed by our tool.

In advance, we sent the participants the GitHub links of the snapshots of the repositories that we used [32], and asked them to take a look at them, and if possible rank them. Everyone looked at them before the interview, for a period ranging from five minutes to two hours. At the start of the interview, some had already made up their mind on the ranking whilst others had not yet decided on a ranking.

During the interview, the following questions were asked:

1. What maintainability problems have you encountered in Puppet?
2. A lot respondents to the survey have said that overuse of execs is a bad practice. Do you agree? If so, what issues arise when execs are overused?
3. What do you think is missing from the model? What should be changed?
4. What do you think should be added to the model?
5. Do you think the model is useful in practice?

At the start of the interview we first explained what maintainability is according to the ISO 25010 standard. Different people have different interpretations of quality, so we made clear the focus of the interview was on maintainability. After this, we asked questions 1 and 2.

Then, we asked the interviewees to make a ranking of the five systems, with the best system being first and the worst system last, based on their maintainability and quality. We provided descriptive statistics of the related systems and a call graph for every system.

After this ranking, we showed them our model and explained our use of risk profiles, and asked questions 3, 4 and 5. This concluded the interview.

As is visible in Figure 8, there are differences between the raters in rankings of the systems. These differences can be attributed to a few different factors. Some interviewees put a very large emphasis on the volume of the codebase, and mostly used that to rank the codebases. Some interviewees

#parameters	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
non-third party	0.0	0.0	0.0	2.0	4.0	7.0	12.0	21.0	48.0	2416.0
third party	3.3	10.6	22.0	30.6	54.0	87.6	112.2	170.4	331.5	633.0

#LoC	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
non-third party	10	18	29	44	63	92	141	212	406	19262
third party	37	119	209	276	479	596	826	1261	1935	3869

#branches	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
non-third party	1	1	2	4	5	7	12	20	39	1301
third party	6	14	22	30	47	73	111	140	174	480

Fig. 6: Quantiles of #parameters, #LoC and #branches per module

Risk Category	Complexity Value
Low Risk	1-7
Medium Risk	8-20
High Risk	21 - 34
Very High Risk	34+

(a) Risk profile for complexity

Stars	Medium Risk	High Risk	Very High Risk
*****	2.4	0.0	0.0
****	13.0	1.5	0.0
***	69.0	53.2	7.4
**	86.5	76.2	34.0

(b) % of LOC allowed per risk category for Complexity

Risk Category	Execs value
Low Risk	0
Medium Risk	1
High Risk	2
Very High Risk	3+

(c) Risk profile for execs

Stars	Medium Risk	High Risk	Very High Risk
*****	0.50	0.00	0.0
****	5.90	2.50	0.7
***	54.95	51.25	1.8
**	77.45	75.25	7.6

(d) % of files allowed per risk category for the number of execs

Risk Category	File Length
Low Risk	0 - 45
Medium Risk	46 - 110
High Risk	111 - 345
Very High Risk	345+

(e) Risk profile for filelength

Stars	Medium Risk	High Risk	Very High Risk
*****	31	3.3	0.0
****	56	26.0	7.3
***	90	77.5	14.0
**	95	89.5	45.0

(f) % of LOC allowed per risk category for filelength

Fig. 7: Risk profiles and star ratings

System	GovUK	RING	Fuel-infra	Wiki-media	Kilo-Puppet
Rating	3,57	3,47	3,05	2,91	2,34
Rank	1	2	3	4	5
Interview 1	3	1	2	4	4
Interview 2	1	2	3	4	5
Interview 3	2	1	3	5	4
Interview 4	1	4	3	2	5
Interview 5	1	2	4	3	5
Interview 6	1	2	3	4	5
Interview 7	1	2	4	2	5
Interview 8	2	1	3	5	4
Interview 9	4	1	2	5	3
Median Rank	1	2	3	4	5

Fig. 8: For each of 5 systems, the rating and rank according to our model, the rankings the interviewees gave the systems and the median of the rank of the interviewees.

considered the presence of tests and documentation to be very important, while others did not. In addition to this, the ratings of some systems were very close to each other, e.g., RING and GovUK. Some interviewees, like number 7, ranked some repositories as equally maintainable. Finally, there might have been differences in the definition of maintainability used. Some interviewees considered how easy it would be to *start* maintaining while others considered how easy it would to *keep* maintaining the system.

Notice, first, that the median of the interview ranks is equal to the rank given by the tool. We have used the median here since rank is an ordinal variable, and if the mean of this data is taken, this might be heavily influenced by outliers (for example, an interviewee might rank a system as 5, but can still consider it to be of high quality). It would have been meaningful to take the mean of this data if we had asked for a rating, since in that case the size of the observed quality difference between systems can be expressed.

The inter-rater agreement using Kendall's W — we did not include our tool as a rater — was 0,589, with a p-value of $2 * 10^{-4}$. This means that between the interviewees there was strong consistency [13].

B. Answers to Questions

We summarize the answers given to the questions asked during the interview. The interview transcripts can be found in Appendix B of [4].

What maintainability problems have you encountered in Puppet? Different types of problems were reported. Upgrading third party libraries was reported, as well as a general lack of tests or the necessity to maintain existing tests.

A lot respondents to the survey have said that overuse of execs is a bad practice. Do you agree? If so, what issues arise when execs are overused? The overuse of execs is considered bad across the board. The main reasons for this are non-idempotence, being outside the declarative model, and lack of support from the IDE and the language. That said, most people did indicate that execs aren't that bad if used with care. This is in agreement with the survey answers.

What do you think is missing from the model? What should be changed? A few items were mentioned here.

Execs: instead of a rating, a violations category for execs should be considered. Additionally, look for execs that always run, either by analyzing the resource itself or the output of a Puppet run.

Lint Warnings: some warnings might be disabled, which can differ from project to project. In addition, some warnings are unavoidable or false positives. For example, a warning results if you inherit from a params class, because doing so makes code incompatible with Puppet 2.7. However, this is not problematic if you use Puppet of a later version. Another example is having a quoted boolean in the code. Sometimes doing so is quite acceptable, leading to a false positive.

Parameters: having a high number of parameters is considered of lesser importance by the experts. Some of them indicated that the number of parameters should have different thresholds, or that only the number of required parameters should be taken into account.

Duplication: duplication is a useful metric, but either the block size should be changed (currently, things may be considered duplication that aren't), or a better clone detector should be implemented so that cloning and reordering parameters can be robustly detected. Our current implementation does not detect these cases.

What do you think should be added to the model? Again, a few aspects were mentioned.

Metric to rate a module: in some cases a "god module" was present in the configurations. Having a module with too many responsibilities is not following the best practice of modules only having one task, and can become a maintenance risk. Whilst module degree measures whether a module has too many dependencies, if the module has a few connections but is very large, this is also an indication of too many responsibilities.

Testing: currently tests are not taken into account, but our experts do report they would like to see a metric that does, e.g., by measuring test code volume, assert density, or "node coverage".

Library Management: are third party modules that are included in code (if any) marked as such and never edited? The best practice is to never edit third-party libraries, because it may become difficult if not impossible to combine with upstream changes, while also adding to the total maintenance burden. In addition to this, new employees will also have to get used to your edited third party libraries instead of the ones that are commonly used.

Documentation: experts found documentation to be a very important aspect. Particularly if a codebase is very large, having documentation to explain where to start and how code is developed is very important. An additional challenge is making sure that what is written in the code and in the documentation remain consistent. This is of course very hard to detect automatically.

Other quality related aspects should be considered as well when judging a configuration, such as how passwords and other secrets are handled, end-of-life third-party modules, and the presence of certain hardcodes. The last aspect is likely to be particularly difficult to implement.

We conclude this section with our final question: *do you think the model is useful in practice?*

Every respondent considered the metrics to be useful both to find out what kind of problems a codebase poses, and as a general indication of quality. One interviewee indicated that he would want to use a tool like this in his own organization.

C. Threats to Validity

As usual in this kind of study there are various threats to validity. We consider internal (credibility) and external validity (generalizability). As to the former, the survey described in Section IV ensures that our metrics reflect what Puppet developers believe to contribute to the maintainability of Puppet code. The outcome of the first survey question implies that our respondents have a broad range of levels of experience, and the set-up of the survey ensures that we contacted only developers with more than a glancing exposure to Puppet by ensuring that they have made more than 10 commits in at least 2 Puppet repositories.

As we explained before, rating the quality is particularly important for large repositories. Therefore, we included only large Puppet repositories (5000 lines and up) in our calibration and made an effort to delete potential clones and duplicates so that our calibration could not be easily biased by the inclusion of many (almost) copies of the same repository.

As to generalizability, the thresholds for the various star-ratings derived here were only for 17 systems. It is not unlikely that when more or other systems are considered, the computed thresholds will change. Also, we restricted ourselves to publicly available systems, and our work may not generalize to private ones. Moreover, due to time constraints on the part of our experts, we could not consider more than 5 systems in the validation study. On the other hand, considering more systems would probably also make it harder for experts to come up with a strict ranking.

VIII. Related Work

The work of Sharma et al. [25]on Puppeteer shares many similarities with ours: we both extended puppet-lint, constructed a metric catalog, and mined GitHub repositories. There are also many differences: (1) we did an extensive expert validation, while in their case only one Puppet developer was consulted; (2) we selected only complete Puppet repositories, while they selected all repositories that contained Puppet code and had 40 or more commits; (3) we did not include the original puppet-lint output as a metric; (4) we took great pains to ensure that the final 17 systems we worked with were of substantial size, and not clones of each other, while for Puppeteer they used many more repositories for validation and were not so selective. For example, they employed the puppet-lint parser to decide that certain files were Puppet source files, but as it happens that parser is very forgiving. Also, we manually inspected the 105 or so systems we were left with, and among these found only 17 that we considered to be unique enough.

However, the main difference is the focus of the research: the goal of Puppeteer is to detect code smells in Puppet, and this makes sense both for large and small repositories, while in our case we are trying to find a general maintainability rating for (large) Puppet repositories. Notwithstanding, our models do show a lot of similarities: both models include measures on complexity, duplicate blocks, exec statements, class (file in our model) size and modularity, although the precise implementation might differ. Puppeteers metric catalog, as well as puppet-lint itself, is a valuable tool for development teams to check their own code for any known smells and ensure that their codebase adheres to as many known best practices as possible.

Xu & Zhou [27] have researched errors in configuration management, so-called misconfigurations, and have analyzed their causes. Just as unmaintainable software can be a financial disaster, misconfigurations can also cost a lot of money. They analyze why things are misconfigured, and also suggest solutions to some of these problems. Their subject differs from ours, but we consider it a useful source to help troubleshoot difficult errors.

Alves et al. [1] have shown how to derive thresholds from metrics for the SIG quality model. They demonstrate how to automatically derive thresholds, and provide a justification for why the thresholds in the SIG model are what they are. We followed a similar approach.

Cox [8] has used expert opinion to rate a dependency freshness metric. The dependency freshness metric tries to capture how well developers keep third-party dependencies of their software system up-to-date. Their validation approach was similar to ours, using both semi-structured interviews and having participants construct a ranking of the subject systems.

Lampasona et al. [15] have proposed a way to perform early validation of quality models, based on expert opinion. We propose validating quality models with respect to their completeness (it addresses all relevant aspects) and to their minimality (a model only contains aspects relevant for its application context). We used minimality and completeness in our validation interview, asking what could be removed/changed, or what should be added.

Building Maintainable Software [28] was a good resource for explaining the measurements in the SIG model. The book describes ten guidelines on how to build maintainable software, and whilst other papers on the SIG model explain the implementation, the book also provides justifications for why certain metrics are used and also demonstrates what happens when things go wrong.

IX. Conclusion

In this paper we have described the complete process of developing a code quality model for the Puppet configuration language. We have used an existing quality model as a starting point, considered various Puppet specific metrics for our model based on a survey among Puppet developers, implemented the metrics in a tool, ran the tool over a large bechmark extracted from Github, derived thresholds for each of the chosen metrics to arrive at a single 5-star rating for real-world Puppet configurations, and validated our ratings with experts. The validation showed that the measurement model and tool provide quality judgments of Puppet code that closely match the judgments of experts. The experts deemed the model appropriate and usable in practice.

As to future work, the model we devised cannot be considered minimal and complete. A case can be made for parameters to be left out as a metric, because there is a high Spearman correlation between both parameters and complexity, and the former was mentioned by one expert as not being very important. Note though that the parameter count can be used to distinguish third party code from other code. Section VII lists a number of additional metrics that could be included in our tool.

The 17 systems we used to calibrate our model gave us a first good initial calibration, but is too low to achieve a reliable calibration. More work is therefore needed here. Finally, we note that at this time metrics are weighed uniformly and it may well be that better results can be obtained by assigning different weights to the various metrics.

More generally, we expect that the approach as laid down in this paper can be applied to Ansible and Chef. We surmise that the more general metrics will be useful again, but that our model will not work for them unchanged: some of the Puppet-specific metrics will need to be replaced by Ansible or Chef specific ones. More study is certainly needed here.

Acknowledgments

We thank the survey participants as well as the experts for their contribution to our work. We thank the anonymous referees for their comments and help us improve our paper.

References

[1] T. Alves, C. Ypma, and J. Visser. ”Deriving Metric Thresholds from Benchmark Data”, In proceedings of the 26th IEEE International Conference on Software Maintenance, 2010.

[2] T. Alves, J. Correia, and J. Visser. ”Benchmark-based Aggregation of Metrics to Ratings”, In Proceedings of the Joint Conference of the 21th International Workshop on Software Measurement (IWSM) and the 6th International Conference on Software Process and Product Measurement (Mensura), pp20-29, IEEE Computer Society, 2011.

[3] https://www.ansible.com

[4] E. van der Bent. ”Defining and measuring Puppet code quality.” MSc Thesis (ICA-3583600). June 21, 2016. Dept. of Information and Computing Sciences, Utrecht University. `https://dspace.library.uu.nl/bitstream/handle/1874/335044/thesis.pdf?sequence=2`

[5] http://evanderbent.github.io/puppet/2016/01/28/puppet-code-quality-survey.html

[6] https://github.com/evanderbent/PuppetSurveyData

[7] https://www.chef.io

[8] J. Cox. ”Measuring Dependency Freshness in Software Systems”, Master’s thesis, Radboud University Nijmegen, 2014.

[9] T. Delaet, W. Joosen, and B. Vanbrabant. ”A Survey of System Configuration Tools .” LISA. 2010.

[10] https://docs.puppet.com/facter/

[11] G. Gousios, ”The GHTorrent Dataset and Tool Suite”, in Proceedings of the 10th Working Conference on Mining Software Repositories (MSR ’13), 2013, pp. 233–236, IEEE Press.

[12] G. Gousios, M. Storey, and A. Bacchelli. ”Work Practices and Challenges in Pull-Based Development: The Contributor’s Perspective”, in Proceedings of the 38th International Conference on Software Engineering, 2016.

[13] C. Griessenauer, J. Miller, B. Agee, W. Fisher, J. Curé, P. Chapman, P. Foreman, W. Fisher, A. Witcher, and B. Walters. ”Observer Reliability of Arteriovenous Malformations Grading Scales using Current Imaging Modalities”. Journal of Neurosurgery, 120.5, 2014.

[14] I. Heitlager, T. Kuipers & J. Visser. ”A Practical Model for Measuring Maintainability”, QUATIC 2007, pp. 30–39.

[15] C. Lampasona, M. Kls, A. Mayr, A. Gb, and M. Saft. ”Early Validation of Software Quality Models with respect to Minimality and Completeness: An Empirical Analysis.” Software Metrik Kongress. 2013.

[16] https://github.com/puppetlabs/puppetlabs-motd

[17] S. Pandey. ”Investigating Community, Reliability and Usability of CFEngine, Chef and Puppet .” Department of Informatics, University of Oslo, 2012.

[18] https://www.Puppet.com

[19] http://puppet-lint.com

[20] https://docs.puppet.com/puppet/

[21] https://docs.Puppet.com/Puppet/latest/reference/lang_classes.html

[22] https://docs.Puppet.com/Puppet/latest/reference/lang_resources.html

[23] https://docs.Puppet.com/guides/style_guide.html

[24] K. Torberntsson & Y. Rydin. ”A Study of Configuration Management Systems .” Uppsala University, 2014.

[25] T. Sharma, M. Fragkoulis, and D. Spinellis. ”Does your configuration code smell?”, in 13th international conference on Mining Software Repositories (MSR ’16), 2016.

[26] K. Weins, ”Cloud Management Blog”, http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey

[27] T. Xu and Y. Zhou. ”Systems Approaches to Tackling Configuration Errors: A Survey .” ACM Computing Surveys 47, no. 4 (July 2015).

[28] J. Visser, P. van Eck, R. van der Leek, S. Rigal, and G. Wijnholds. ”Building Maintainable Software. Ten Guidelines for Future-Proof Code”, O’Reilly Media, 2016.

[29] Y. Jiang, and B. Adams. ”Co-evolution of Infrastructure and Source Code: An Empirical Study”, Proceedings of the 12th Working Conference on Mining Software Repositories (MSR ’15), IEEE Press, pp 45-55, 2015.

[30] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. ”An Empirical Analysis of the Docker Container Ecosystem on GitHub”, Proceedings of the 14th Working Conference on Mining Software Repositories (MSR ’17), IEEE Press, pp 323-333, 2017.

[31] B. Adams and S. McIntosh, ”Modern Release Engineering in a Nutshell – Why Researchers Should Care,” 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 78-90.

[32] https://github.com/fuel-infra/Puppet-manifests
https://github.com/alphagov/govuk-Puppet
https://github.com/wikimedia/operations-Puppet
https://github.com/CCI-MOC/kilo-Puppet
https://github.com/NLNOG/ring-Puppet