Configuration File Manipulation with Configuration Management Tools

Student Paper for *185.307 Seminar aus Programmiersprachen*, SS2016

Bernhard Denner, 0626746
Email: bernhard.denner@gmail.com

***Abstract*—Manipulating and defining content of configuration files is one of the central tasks of configuration management tools. This is important to ensure a desired and correct behavior of computer systems. However, often considered very simple, this task can turn out to be very challenging. Configuration management tools provide different strategies for content manipulation of files, hence an administrator should think twice to choose the best strategy for a specific task. This paper will explore different approaches of automatic configuration file manipulation with the help of configuration management tools. We will take a close look at the configuration management tool Puppet, as is offers various different approaches to tackle content manipulation. We will also explore similarities and differences of approaches by comparing it with other configuration management tools. This should aid system administrators who have to deal with configuration files and will assist in choosing the "best" strategy for a specific configuration task.**

I. Introduction & Overview

In these days, system administration can be a complex task. With the advent of cloud computing, the number of computer systems an administrator has to handle increased dramatically. As one can imagine, installing software and managing the configuration of hundreds or even thousands of systems is a big challenge [1]. Administrators have to react on changes to the environment, fix security issues and keep software up-to-date. Doing this manually, machine by machine is almost impossible [2].

Therefore configuration management (CM) tools have been developed to assist administrators in managing their systems. The most popular tools today are *CFEngine* [3], *Chef* [4], *Ansible* [5] and *Puppet* [6]. Despite their differences, they all have one common approach. An administrator defines a desired system state in form of text files. Such states can be seen as software packages to be installed, configuration files with defined content to exist or system services to be running. This desired system state is applied on as many computer systems as required, in a reproducible way.

One of the central tasks of configuration management is the manipulation of configuration files. Because of different requirements, many CM-Tools offer different methods for doing this, often with different approaches such as partially modifying configuration files or defining the complete content at once. Each method has something to commend it, being simplicity and ease of use or having complete control.

These different approaches of configuration file manipulation by CM-Tools will be explored in this paper. Therefore, we will put a focus on *Puppet*, which is one of the most popular CM-Tool in these days and offers a huge variety of configuration manipulation methods. But we will also take a look at other CM-Tools to explore their view of configuration file handling.

In Section II an introduction will be given about CM-Tools, examining which ones are established in the community and how they work. Before we dive into file manipulation strategies, Section III will give some introduction in configuration stores. Section IV will describe different configuration manipulation strategies of *Puppet*, whereas Section V will look at approaches of other CM-Tools. Section VI will show some scientific papers about CM-Tools. Finally we will draw a conclusion on the topic in Section VII.

II. Configuration Management Tools

Configuration management tools are being used by system administrators for a long time now, however it is sometimes stated to be one of the most controversial aspects of system administration [7].

One of the first theoretical work on configuration management tools was done by Burgess. In his work [8], he introduces *CFEngine* [3]. The aim of this tool is to provide the user a central management instrument for different types of computer systems within a network. *CFEngine* enables its users to define many aspects of UNIX based systems in one central file in a high-level manner by a newly introduced configuration language. This configuration language is used as a form of abstraction mechanism, which hides variations between different operating systems and allows the user to think in a more abstract way. Desired properties of managed systems are defined in a declarative manner, as well as relationships and dependencies between these properties. *CFEngine* uses the concept of bundles and classes to group system aspects and provide a form of modularization. A *CFEngine* configuration program is then used on the target systems to automatically enforce the defined aspects and properties. In Burgess later works [9] and [7], he further extended his theoretical concepts for *CFEngine*, which later got integrated in newer versions.

Puppet [6], grew out of dissatisfaction with *CFEngine* [10], therefore the primary concepts of these two tools are very similar. *Puppet* also allows the user to define desired system properties in a central repository. Therefore its own declarative domain specific language (DSL) is used to model system properties (called *resources*), order-relations and dependencies between these resources. Similar to *CFEngine promises*, *resources* are the main abstraction mechanism in *Puppet*, hiding

differences between operating systems. Multiple *resource* definitions can be grouped by classes and modules, which helps organizing *Puppet* source code, written in *Puppets* own DSL. An *agent* running on the target system ensures all defined resources are in the desired state, otherwise certain actions will be initiated to enforce this state, e.g. installing a package, creating user accounts or updating the content of files [11], [12].

Beside *CFEngine* and *Puppet* a bunch of other CM-tools have emerged in the industry, such as *Chef* [4], *Ansible* [5] or *Salt* [13], to name a few.

III. Configuration Files

Software configuration values can be stored in different forms. Plain text files are very common for UNIX based applications, while on other platforms, such as Microsoft Windows OS Family, often a central configuration database is used. Both forms have their advantages and disadvantages. Central configuration databases allow information querying and manipulation through a standardized API, while plain text files requires parsing its content based on a special syntax of the configuration format. Over time, different formats for such configuration files have evolved, and often it seems each tool uses its own syntax for its files. However, there are programming libraries available for a lot of the commonly used file formats, such as INI. Very often, high level programming languages ship such libraries already as part of their runtime core.

When manipulating plain text configuration files, especially with CM-tools in an automatic way, we have to consider certain aspects. We have to respect and observe the format and syntax rules of the configuration file, otherwise the configured software might refuse reading the file. When syncing configuration files between different computer systems, we might accidentally overwrite some system-individual settings like hostnames or IP-addresses. Often configuration file formats allow an administrator to add comments for documentation purposes, which can help during problem analysis. Automatically generated configuration files often miss such comments.

For the comparison of the individual configuration manipulation methods, we will focus on the following aspects:

- syntax awareness: is the correct syntax of the configuration file automatically ensured, or is it in the responsibility of the user?
- defaults preservation: is the method able to change only certain parts of the configuration file, in order to keep default settings untouched?
- expressibility: is the method able to utilize all aspects of the corresponding configuration file format, especially handling of comments?

IV. Manipulating Configuration Files with Puppet

Before describing different methods for configuration file manipulation with *Puppet*, we will give a short introduction of the primary *Puppet* concepts and of the *Puppet* DSL to make things easier to understand.

As already stated above, the central element in *Puppet* for defining certain aspects of a system, is the *resource*. Each individual characteristic of a computer system is treated as a resource, e.g. each single file, each package, user, directory but also parts of files or settings of configuration files. Resources that are not subject of the configuration are usually ignored by *Puppet*, those resources are called *unmanaged resources*. In contrast, *managed resources* are all resources which are defined in the underlying *Puppet* code. These resources are managed by *Puppet* and the desired states (file permissions or content, package state...) are enforced [11], [12].

Each resource is clearly defined by its *resource type*, such as `file`, `package`, `user`, a type-unique resource identifier, such as file or package name, and variable amount of properties, which define the desired resource state. An example can be seen in Listing 1. Resource types are implemented in Ruby and form the primary abstraction mechanism in *Puppet*, to hide differences between different operating systems. Thus a *Puppet* user will always install packages with the resource type `package`, being on Ubuntu, Fedora, Solaris, MacOS or even Windows. *Puppet* comes with many built-in resource types to manage the most important aspects of computer systems. However, each user can implement other, more specialized resource types, as part of a new *Puppet* module. Many of such specialized modules are publicly available on [14], a collaboration and sharing platform for *Puppet* modules.

```
# manage a single file
file { "/etc/resolv.conf":
  ensure  => "file",
  owner   => "root",
  group   => "root",
  mode    => "0644",
  content => "nameserver 8.8.8.8"
}

# install a package
package { "firefox":
  ensure  => "installed"
}
```

Listing 1: example of resource definitions

Another important language construct in *Puppets* DSL are *classes*. Classes are used to group resource definitions and help to modularize *Puppet* code. Classes can also have parameters, which allow to separate code and data (the concrete configuration values). Listing 2 shows the class `dns` with one parameter `nameserver` with a default value.

```
class dns($nameserver = "8.8.8.8") {
  file { "/etc/resolv.conf":
    content => "nameserver ${nameserver}"
  }
}
```

Listing 2: example class

The following sections will describe different methods for manipulating and defining the content of configuration files.

A. Content Defined by Strings

The built-in resource type `file` offers the *Puppet* developer two parameters to define the content of a file: `content`

and `source`. The first one takes a String argument and, as the name suggests, directly defines the content of file. An example usage can be seen in Listings 1 and 2. Listing 1 defines the file content by a hard-coded string, whereas Listing 2 uses the more generic way, as the hard-coded string is expanded through a class parameter or variable.

An alternative to the classic string definition are *here documents* or *Heredocs*, as also known from the *Bash* or *Perl* scripting languages. A Heredoc defines a string, which is allowed to span over multiple lines without special treatment of quoting character such as '"'. Heredocs are enclosed by a special Heredoc start tag '`@("CONTENT")`' and the corresponding end tag '`| CONTENT`', whereas the keyword (here `CONTENT`) is freely selectable. As for normal string definitions, Heredocs also allow to use variables in its content. This makes the definition of a complex string, such as multi-line configuration files, directly in the *Puppet* source code possible. However, it might not be a good programming practice, as it mixes code and data in the same source file.

The advantage of this file-content approach is its simplicity. However, extensive uses, especially Heredocs can rapidly grow source files and a developer quickly looses the overview of his/her *Puppet* source code. Syntax awareness, simply does not exist. Its possible do define any content, therefore it has a very high expressibility. However, managing only portions of configuration files is not possible either, as this is simply an all-or-nothing approach.

B. Pull from Other Sources

As described above, the resource type `file` offers a second parameter for defining the content of files: `source`. If this parameter is set to a suitable value, *Puppet* will pull the content for this file from the defined location, either by downloading from a Web server, *Puppets* own built-in file transfer mechanism or from a locally available file. However, this file-source method doesn't allow any dynamic content definition. The content is treated as-is. Therefore, it might be not very suitable for configuration file manipulation.

C. Content Generated by Templates

Beside resource types and classes, *Puppets* DSL also includes the concept of *functions*, as known from typical procedural languages. Functions have a unique name, can take an arbitrary number of arguments and one return value as result. In *Puppet* they are often used for type checking, type conversions and string manipulations.

Puppet has two built-in functions, which are quite useful for defining content of files: `template()` and `inline_template()`. Both functions allow rendering of strings based on ERB templates. ERB (Embedded RuBy) is a special feature of Ruby, which is the language *Puppet* is implemented in, to embed Ruby code in normal text files. This embedded code is evaluated during template parsing, which defines the result in a dynamic way. The function `template()` takes a file name as string argument, defining the path to the used template file, whereas the function `inline_template()` directly takes the template definition itself as string argument. Both function return the rendered template result, which can be directly used by the `file` resource type for example. Listing 3 shows an example ERB template to define the content of the UNIX resolver configuration file.

```
<% if @dns_search != '' -%>
search <%= @dns_search %>
<% end -%>
<% @dns_servers.each do |server| %>
nameserver <%= server %>
<% end %>
```

Listing 3: example ERB template

The corresponding usage of such a template within a `file` resource can be seen in Listing 4.

```
class dns($dns_search, $dns_servers) {
  file { "/etc/resolv.conf":
    ensure  => "file",
    content => template("path/2/template")
  }
}
```

Listing 4: example ERB template usage

As one can see in Listing 3, Ruby code fragments are enclosed by the special tags '<%' and '%>'. Everything else is treated as normal text. Similar concepts can be found for example in PHP. *Puppet* allows to use its local class variables (here `$dns_search` and `$dns_servers`) within the template as Ruby instance variables. Listing 3 demonstrates simple uses of conditions ('`if ...`') and loops ('`.each do ...`').

Beside the Ruby-ERB template engine, starting with version 3.8, *Puppet* includes its own template engine, called *EPP*. It brings a better integration with the *Puppet* DSL. The corresponding function `epp()` and `epp_inline()` are equivalent to its ERB counterparts, whereas the EPP (embedded Puppet) template engine is used.

Defining content of configuration files by ERB/EPP templates is very powerful. It allows variable substitution, conditions, loop and much more, as it allows Ruby code execution for generating the resulting file content. Since it is possible to define any form of text content, the resulting output is not tied to any special format or syntax. Therefore the expressibility is very high. However, syntax validation of special configuration formats is not in the scope of this method. Therefore, its quite easy for *Puppet* developers to generate configuration files with an invalid syntax.

This form of content definition is primary driven by the template functions, which have to be used together with a *Puppet* resource type to 'transfer' the result to the target configuration file. As already said, the resource type `file` is often used for this purpose. However, as we have seen in section IV-A, the `file` resource type is not able to manage portions of a file. Therefore its not possible to preserve default values using this method.

D. Line based manipulation

Until now, only methods for defining the complete content of files were discussed. When it comes to partial modifications

of configuration files, the resource type `file_line` from the *Puppet* module *puppetlabs-stdlib* is very interesting. This resource type allows line based modifications of files based on regular expressions, similar to the UNIX tool *sed*.

```
file_line { "/etc/resolv.conf_searchlist":
  ensure => 'present',
  path   => "/etc/resolv.conf",
  line   => "search $dns_search",
  match  => "^search .*",
}
```

Listing 5: file_line example

Listing 5 shows an example resource definitions of the `file_line` resource type. The desired value defined by parameter `line` will replace the line matched by the regular expression of parameter `match`. If a matching line is found, this line will be replaced by the defined value, otherwise a line with the desired value will be added. This makes this resource type very suitable for modifying configuration settings in configuration files, as it can be used to replace single configuration values, while ensuring settings are not added twice. Therefore this method allows partial modifications of configuration files and enables its users to preserve default settings supplied by the OS distributions.

In terms of expressibility, the `file_line` resource type is very flexible, since it allows to define any content. However, when it comes to modifying multiple lines at once, for example to add comments to certain settings, this method is not suitable, since the `match` property will only match a single line, as the resource type name already suggests. Therefore we will consider expressibility of this method to be limited. The `file_line` resource type does not care about the managed content itself. Therefore syntax validation of the given content is not performed at all.

E. Format Specific Modules

A more advanced way for modifying configuration files are methods, which respects the special format and syntax of configuration files. A prominent example of such a resource type is the `ini_setting` type, defined by the module *puppetlabs-inifile*. This module allows to modify single settings in configuration files respecting the INI-File format.

```
ini_setting { "puppet-server":
  ensure  => 'present',
  path    => "/etc/puppet/puppet.conf",
  section => "main",
  setting => "server",
  value   => "example.com"
}
```

Listing 6: ini_setting example

Listing 6 shows an example application of the `ini_setting` resource type. This example will modify the value of the setting `server` within the section `main`. If this configuration setting does not exist in the defined file, the setting will be added. One important thing to mention here is, that this resource type treats each setting within an INI-based file as a single resource instance. Therefore it allows partial modifications of configuration files in simple way, while ensuring the correct syntax of the file.

Beside the more general `ini_setting` resource type, *Puppet* has built-in support for modifying important configuration files of UNIX based systems. The resource types `host`, `user`, `group` and `mailalias` are examples here. `host` manages entries in `/etc/hosts` database, `user` will ensure certain user accounts exists and therefore manages `/etc/passwd`, `group` does the same for user groups and `mailalias` manages mail alias files. Each of these resource types, respects the format of its corresponding configuration files and treats each setting as a single resource instance. However, some resource types are able to manage more than just the configuration. For example, the `user` type can be used to manage the home directory of the desired user too.

Additionally, *Puppet* has built-in support for generation of configuration files for the monitoring tool *Nagios*, which is a well established monitoring solution. The `nagios_*` (`nagios_host`, `nagios_service`, `nagios_command`...) resources types can be used to manage single entries for *Nagios* configuration files. Again, these resource types are treating each setting (Nagios object definition) as single resource instance and therefore are able to manage the corresponding configuration files partially.

Many application specific modules found in [14] allow partial modifications of the corresponding application configuration files, however they are limited to a specific application and will not be considered here.

As we have seen, each of the described methods here, automatically ensures the compliance of the underlying syntax of the configuration file. Additionally, each method manages files partially, through modifying dedicated entries or regions. Therefore its possible to preserve default values. In terms of expressibility we will consider this method also as limited, as none of these modules is able to add comments to the corresponding settings. However, this is just limited by the underlying resource type implementation. It would be possible to add special treatment of comments to each resource type, for example implementing a `comment` parameter to the `ini_setting` resource type.

F. Frontends to Central Configuration Backends

The configuration definition strategies defined here are not modifying the underlying configuration files directly, instead they use tools to manage a central configuration database. One prominent resource type for doing this, are the `registry_key` and `registry_value` resource types found in the module *puppetlabs-registry*. These two resource types, allow manipulations of the Microsoft Windows registry configuration database.

Another example for such a strategy is the `gconf` resource type, defined by the *rohlfs-gconf* module, which will manage configuration settings for the *Gconf* configuration database, used by the *GNOME* platform.

Both methods allow partial modifications of configuration settings, which leaves other values untouched. Syntax validation is automatically ensured by the underlying configuration

frontend tools used by these resource types. However, expressibility is again limited as none of the two tools allows to add comments to configuration settings, although this is mainly a limitation of the underlying configuration database and the used frontend tooling.

G. AUGEAS

The fact that many configuration file formats have emerged, which makes automatic modification of configuration files difficult, was the main motivation for the development of *AUGEAS* [15]. *AUGEAS* is a library allowing retrieval and modification of configuration values of different configuration files, which adhere to supported formats, by a standardized API.

Puppet has a built-in support of the *AUGEAS* API, and can be used with the `augeas` resource type. Listing 7 shows an example application of this resource type, modifying the file `/etc/puppet/puppet.conf`. This example will set the configuration option `server` of section `main` to the value of the variable `${server}` and adds a comment line one line before the `server` configuration option.

```
augeas{ "puppet-server" :
  context => "/files/etc/puppet/puppet.conf/main",
  changes => [
    "set server ${server}",
    "set #comment[following-sibling::server]\
      [last()] 'central puppet server'"
  ]
}
```

Listing 7: AUGEAS example

This method is one of the most advanced strategies for automatic modification of configuration files. *AUGEAS* automatically ensures the correct syntax of configuration formats. Additionally, modifications are done on a per-configuration setting basis. Therefore, it allows the user to leave default values untouched. And further, through the *XQuery* like syntax for referencing sections around a specific configuration setting, it also allows modifications of surrounding regions, especially comment lines within a configuration file.

V. Manipulating Configuration Files with other CM-Tools

Other CM-Tools have quite similar approaches compared to *Puppet*. *CFEngine* also allows defining content of (configuration) files with the following three major approaches: copying from source, template-based and line-based editing. These different methods are all integrated in the *CFEngine* construct `files` *Promise*. The line based editing approach works like search and replace and is based on regular expressions, similar to the `file_line` resource type of *Puppet*. *CFEngine* has no *AUGEAS* integration. However, *CFEngine* provides format specific configuration file editing operations for common file formats, such as XML- or INI-formats [16].

Chef, another popular CM-Tool implemented in Ruby, allows defining file content by its resources `file`, `remote_file` and `cookbook_file`, whereas the latter two copy content from remote locations. *Chef* also supports the ERB template engine and is implemented by the `template` resource. Line based editing is only possible through executing specific scripts, for example using the methods provided by the Ruby class `Chef::Util::FileEdit`. Format specific configuration file manipulations are not possible with standard *Chef* concepts. However, *AUGEAS* integration can be added with a community provided extension [17].

Salt enables file management through its `file.managed` state. The main use case is to pull the desired content from a *Salt* master node, either as-is, without modifications or by applying one of the supported template engines. Line based editing is possible with the `file.replace` state, which also supports regular expressions. JSON and YAML files can be defined by the `file.serialize` state, which could be quite useful for simple configuration files, whereas this is a "all-or-nothing" approach. Additionally, format specific manipulations can be done through the *AUGEAS* integration of *Salt* [13].

Ansible supports similar file operations, such as content pulling from remote sources, rendering file templates or line based editing. Relevant *Ansible* modules are `copy`, `fetch`, `template`, `lineinfile` and `replace`. However, *Ansible* has a unique feature especially for configuration file editing, provided by the `blockinfile` module. This allows managing a complete block of text at once. *Ansible* adds special marker tags before and after the inserted text block, which enables *Ansible* to identify the concrete region within a file. These marker are added as comment lines, in order not to break format requirements of the configuration file. Additionally, *Ansible* has built-in support for manipulation of INI-format based files. *AUGEAS* support can be added through a community provided module [5].

VI. Related Work

In scientific areas, there is a lot of research going on in the field of configuration management. Spinellis, for example, in [2] gives a good introduction to the use of CM-Tools and why it is worth the effort.

One of the pioneers on the theoretical work on CM-Tools was Burgess with his works [7]–[9], which formed the basis for *CFEngine*. [1] presents *Meta-Config*, a CM-Tool chain specially designed for provisioning private cloud setups. It comes with its own configuration management engine, however the authors stated to replace it in flavor of *Puppet*, which did not happen until now. Vanbrabant introduces in [18] a configuration management framework designed for distributed systems. This framework also comes with its own configuration management engine.

Since there are many different CM-Tools, all with individual features, usage scenarios and different pros and cons, it is not an easy task for system administrators to decide which tool fits best to their own needs. A lot of CM-Tool comparisons can be found on the Internet ([10], [19], [20]) and in scientific literature [21]–[23]. Delaet, Joosen, and Van Brabant present in [21] a comparison framework for CM-Tools, which should help to make decisions on a solid basis. For illustration purposes, this framework is used for evaluating 11 different CM-Tools. [22] extends this framework by a *community* parameter, which should quantify the public community behind each CM-Tool. [23] describes a automatic quality assurance system for open-source CM-Tools. It will automatically measure certain quality

attributes on community provided configuration scripts and inform developers of the corresponding scripts, once problems with their extensions arise. This should help to increase the quality of community provided tool extensions.

VII. Conclusion

This paper summarizes different strategies for automatic manipulations of configuration files with the help of configuration management tools, especially possibilities proposed by *Puppet*. We can see that there are three major approaches for file manipulations:

- the "all-or-nothing" strategies, either with direct content definition, utilizing template engines or pulling from other sources
- line-based manipulations, a search and replace approach, mainly based on regular expressions
- format specific manipulations

However, there is no single "best" strategy. It always depends on the context and which method fits best to the system administrators needs. The predefined-content method is simple to use and deterministic, however very inflexible. The template methods are more flexible and also deterministic, whereas it is still an "all-or-nothing" approach. Methods which modify only parts of a configuration file are very flexible and allow an administrator to keep the focus on desired parts of the configuration, although these methods are not deterministic. Wrong settings that are not touched by the CM-Tool will stay wrong. Therefore it is in the responsibility of the administrator to choose the "best" method for the concrete problem.

References

[1] T. D. Nielsen, C. Iversen, and P. Bonnet, "Private cloud configuration with metaconfig," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, Jul. 2011, pp. 508–515. DOI: 10.1109/CLOUD.2011.63.

[2] D. Spinellis, "Don't install software by hand," *Software, IEEE*, vol. 29, no. 4, pp. 86–87, Jul. 2012, ISSN: 0740-7459. DOI: 10.1109/MS.2012.85.

[3] *CFEngine - Configuration management tool*. [Online]. Available: https://cfengine.com/ (visited on 03/06/2016).

[4] *Chef - Configuration management tool*. [Online]. Available: https://www.chef.io/chef/ (visited on 03/06/2016).

[5] *Ansible is Simple IT Automation*. [Online]. Available: https://www.ansible.com/ (visited on 03/06/2016).

[6] *Puppet Labs: IT Automation Software for System Administrators*. [Online]. Available: https://puppetlabs.com/ (visited on 03/06/2016).

[7] M. Burgess and A. Couch, "Modeling next generation configuration management tools," *Proceedings of LISA '06: 20th Large Installation System Administration Conference*, pp. 131–147, Dec. 2006. [Online]. Available: http://usenix.org/event/lisa06/tech/full_papers/burgess/burgess.pdf.

[8] M. Burgess, "CFEngine: a site configuration engine," vol. 8, 1995.

[9] ——, "On the theory of system administration," *Science of Computer Programming*, vol. 49, no. 1–3, pp. 1 –46, 2003, ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2003.08.001.

[10] A. Tsalolikhin, *State of the Art of Automating System Administration with Open Source Configuration Management Tools*, Jul. 9, 2010. [Online]. Available: http://www.verticalsysadmin.com/config2010/ (visited on 04/12/2016).

[11] S. Krum, W. Van Hevelingen, B. Kero, J. Turnbull, and J. McCune, *Pro Puppet*, Second Edition. Apress, Dec. 2013, ISBN: 978-1-4302-6040-0.

[12] J. Turnbull, *Pulling Strings with Puppet, Configuration Management Made Easy*. Apress, 2007, ISBN: 978-1-59059-978-5. DOI: 10.1007/978-1-4302-0622-4.

[13] *SaltStack automation for CloudOps, ITOps and DevOps at scale*. [Online]. Available: http://saltstack.com/community/ (visited on 03/06/2016).

[14] *Puppet Forge*. [Online]. Available: https://forge.puppet.com/ (visited on 03/06/2016).

[15] D. Lutterkort, "Augeas - a configuration api," in *Proceedings of the Linux Symposium*, vol. 2, Jul. 2008, pp. 47–56. [Online]. Available: http://www.landley.net/kdocs/ols/2008/ols2008v2-pages-47-56.pdf (visited on 03/04/2016).

[16] *CFEngine Reference - Standard Library - Files Bundles and Bodies*. [Online]. Available: {https://docs.cfengine.com/docs/master/reference-standard-library-files.html} (visited on 05/05/2016).

[17] *Augeas cookbook*. [Online]. Available: https://github.com/nhuff/chef-augeas (visited on 05/05/2016).

[18] B. Vanbrabant, "A Framework for Integrated Configuration Management of Distributed Systems," Faculty of Engineering Science, KU Leuven, Jun. 2014. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/453199 (visited on 03/08/2016).

[19] E Dunham, *Configuration Management Comparison*, Jun. 5, 2015. [Online]. Available: http://edunham.net/2015/06/05/configuration_management_comparison.html (visited on 03/08/2016).

[20] P. Venezia, *Review: Puppet vs. Chef vs. Ansible vs. Salt*, Nov. 21, 2013. [Online]. Available: http://www.infoworld.com/article/2609482/data-center/data-center-review-puppet-vs-chef-vs-ansible-vs-salt.html (visited on 03/08/2016).

[21] T. Delaet, W. Joosen, and B. Van Brabant, "A survey of system configuration tools.," in *Proceedings of the Large Installations Systems Administration (LISA) conference*, 2010. [Online]. Available: https://www.usenix.org/event/lisa10/tech/full_papers/Delaet.pdf.

[22] S. Pandey, "Investigating community, reliability and usability of cfengine, chef and puppet," Network, System Administration Oslo, and Akershus University College, 2012. [Online]. Available: http://urn.nb.no/URN:NBN:no-31901 (visited on 12/18/2015).

[23] S. Meyer, P. Healy, T. Lynn, and J. Morrison, "Quality assurance for open source software configuration management," in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, Sep. 2013, pp. 454–461. DOI: 10.1109/SYNASC.2013.66.