Unified Configuration Setting Access in Configuration Management Systems

Unified Configuration Setting Access in Configuration Management Systems

Markus Raab
TU Wien
Vienna, Austria

Bernhard Denner
Thales
Vienna, Austria

Stefan Hahnenberg
University of Duisburg-Essen
Duisburg-Essen, Germany

Jürgen Cito
TU Wien, Austria
MIT CSAIL
Cambridge, MA, USA

ABSTRACT

The behavior of software is often governed by a large set of configuration settings, distributed over several stacks in the software system. These settings are often manifested as plain text files that exhibit different formats and syntax. Configuration management systems are introduced to manage the complexity of provisioning and distributing configuration in large scale software. Globally patching configuration settings in these systems requires, however, introducing text manipulation or external templating mechanisms, that paradoxically lead to increased complexity and, eventually, to misconfigurations. These issues manifest through crashes or bugs that are often only discovered at runtime. We introduce a framework called Elektra, which integrates a centralized configuration space into configuration management systems to avoid syntax errors and avert the overriding of default values, to increase developer productivity. Elektra enables mounting different configuration files into a common, globally shared data structure to abstract away from the intricate details of file formats and configuration syntax and introduce a unified way to specify and patch configuration settings as key/value pairs. In this work, we integrate Elektra in the configuration management tool Puppet. Additionally, we present a user study with 14 developers showing that Elektra enables significant productivity improvements over existing configuration management concepts. Our study participants performed significantly faster using Elektra in solving three representative scenarios that involve configuration manipulation, compared to other general-purpose configuration manipulation methods.

ACM Reference Format:
Markus Raab, Bernhard Denner, Stefan Hahnenberg, and Jürgen Cito. 2020. Unified Configuration Setting Access in Configuration Management Systems. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389257

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389257

1 INTRODUCTION

Configuration settings can be seen as decisions in programs deferred to later stages of the software development process. They drive the run-time behavior of our programs. Values for these configuration settings are, in most cases, manifested as plain text files with vastly different formats and syntax. While several common formats have emerged (e.g., YAML, JSON, XML), most applications impose further restrictions on certain aspects of the configuration file's syntax (e.g., INI) [34]. Configuration management systems, such as Puppet [17] or Chef [23], are introduced to manage the complexity of provisioning and distributing configuration in large scale software systems. They enable automation of software infrastructure, through definition of desired system properties in source code. Manipulating configuration settings, against all intuition, is a source of complexity in configuration management systems [40].

Challenges in Managing Configuration Settings

Globally patching configuration settings that reside in plain text files requires introducing text manipulation or templating mechanisms that lead to increased complexity. Additionally, the configuration management system has no inherent knowledge of syntax rules for different configuration file formats. Problems that are introduced through illegal values for configuration settings cause syntax rule violations that are only detected at a very late stage [39]. Manipulating only a subset of settings, i.e., partial configuration editing, through text manipulation introduces further possibilities for syntax errors [29]. These issues manifest through crashes or bugs that are often only discovered at runtime [39].

Unified Configuration Setting Manipulation

To mitigate these challenges, we propose Elektra providing unified configuration setting access. Elektra introduces a key/value abstraction that reads and writes configuration settings without requiring the developer to perform lower-level file manipulations [7, 27]. We extend the core programming language of configuration management system Puppet with language constructs that enable partial manipulation of configuration files through well-known key/value assignments. The extension, called KDB, combines the flexibility of general purpose configuration manipulation methods through a unified interface for different formats, while still preserving validation for specific syntax rules. Dealing with the intricacies of different configuration file formats is relayed to a configuration

library with a rich set of existing plugins. Our contribution is providing unified access of partial manipulation of configuration files within the configuration management system.

User Study

To investigate the effectiveness of KDB, we conducted a user study with 14 software developers working on three large configuration management scenarios. We measure productivity (i.e., time it took to finish the tasks) as a proxy for usability of the abstractions and dealing with misconfigurations (similar to existing studies in software engineering research [8, 13, 19, 30, 36]). Our baseline were two general-purpose configuration manipulation abstractions (Augeas and ERB), and two specific abstractions for particular configuration file types (INI and HOST). The results show that our proposed system enables significant productivity improvements compared to the general-purpose methods and that the productivity differences to the specific methods were inconclusive (i.e., they did not show a significant difference either way).

2 BACKGROUND

2.1 Configuration Management Languages – Puppet

Configuration management tools allow developers to define desired system properties in order to configure computing resources automatically. While there has been substantial theoretical work on configuration management [3, 4, 12], many systems in the past few years have been developed in industry, such as Puppet [17] or Chef [23]. They provide abstractions for managing several kinds of underlying computing resources, such as package dependencies, permissions, and also *configuration files* – the focus of our study. These systems are often modeled as domain specific languages (DSL), that also encourage developers to build further abstractions.

Puppet is implemented as an embedded DSL in Ruby. Beside Puppet specific language constructs, the Puppet DSL contains very common elements, such as variables and conditional statements [16]. In the following, we briefly describe the main language constructs that are important to understand our proposed approach.

Resources are the main abstraction mechanism in Puppet. 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. 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 a variable amount of properties, which define the desired resource state. An example is shown in Listing 1, which includes two resource declarations. The first declaration uses the resource type "`file`" with the unique resource ID /etc/resolv.conf and defines five parameter. Therefore, this resource declaration describes a desired state for the file /etc/resolv.conf with the specified properties. The second resource declaration is of type "`package`" with the unique resource ID "`firefox`". This declaration instructs the Puppet agent to ensure the package "`firefox`" is installed.

```
# 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: Resource declaration examples

Multiple *resource* declarations can be grouped by classes and modules, which help to organize Puppet source code. An *agent* running on the target system, called *managed node*, ensures all defined resources are in the desired state, otherwise certain actions will be initiated to enforce this state, e.g. installing dependencies, creating user accounts or updating the content of files [15, 35].

Beside resource types and classes, the Puppets DSL also includes the concept of *functions*. 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.

2.2 Configuration File Manipulation

We briefly describe different existing, state-of-the-art methods that facilitate manipulation of configuration files. The examples we show are implemented as abstractions within the Puppet DSL, but have similar implementations in other common configuration management systems, such as Chef.

2.2.1 Templating – ERB. Puppet has a built-in function, which is quite useful for defining contents of files: "`template()`". The function enables rendering of strings based on ERB templates. ERB (Embedded Ruby) is a special feature of Ruby to embed Ruby code in 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 a string argument, defining the path to the used template file and returns the rendered template result, which can be directly used by the "`file`" resource type, for example. Listing 2 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 2: Example ERB template

The corresponding use of such a template within a "`file`" resource declaration is shown in Listing 3.

```
class dns($dns_search, $dns_servers) {
  file { "/etc/resolv.conf":
    ensure  => "file",
    content => template("listing_2.tmpl")
  }
}
```

Listing 3: Example ERB template use

As one can see in Listing 2, Ruby code fragments are enclosed by the special tags '<%' and '%>'. Everything else is treated as normal text. Puppet allows us to use its local class variables (here "`$dns_search`" and "`$dns_servers`") within the template as Ruby instance variables. Listing 2 demonstrates simple uses of conditions ("`if. . .`") and loops ("`each do. . .`").

Defining content of configuration files by ERB templates is powerful. It enables variable substitution, conditions, loops and much more, as it is based on 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, it is quite easy for Puppet developers to generate configuration files with an invalid syntax.

This form of content definition is primarily 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, it is not possible to manage partial manipulations of a file with this construct. Therefore, default values are not preserved using this method, leading to potential security vulnerabilities.

2.2.2 Partial Configuration Setting Manipulation – Augeas. The emergence of many different configuration file formats made automated modification of configuration files difficult. Augeas [18] is a library that uses lenses [2] to manipulate configuration files. It allows developers to retrieve and modify configuration values of different configuration files, which adhere to supported formats, by a standardized API.

Puppet has built-in support for the Augeas API, and can be used with the "`augeas`" resource type. Listing 4 shows an example application of this resource type, modifying the file /etc/puppet/puppet.conf. This example sets the configuration option "`server`" of section "`main`" to the value of the variable "`${server}`" and adds a comment line directly before the "`server`" configuration option.

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

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

Listing 4: Augeas example

2.2.3 Format Specific Abstractions – INI and HOST. A more specialized way for modifying configuration files are methods, that take care of the concrete 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 modifying 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 5: "`ini_setting`" example

Listing 5 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 facilitates partial modifications of configuration files in simple way, while ensuring the correct syntax of the file.

Besides the "`ini_setting`" resource type, Puppet has built-in support for modifying important configuration files of UNIX-based systems. The resource types "`host`", "`user`" and "`mailalias`" are examples here.

For instance, "`host`" manages entries in /etc/hosts database (used in our study). Each of these resource types, respects the format of its corresponding configuration files and treats each setting as a single resource instance.

As we have seen, each of the described methods 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, it it possible to preserve default values. However, in terms of expressibility this method is quite limited, as, for instance, none of these modules is able to add comments to the corresponding settings.

3 UNIFIED ACCESS FOR CONFIGURATION SETTINGS

We briefly introduced common ways to deal with configuration files in configuration management systems. They range from flexible, general purpose methods, such as templating (ERB) or partial manipulation of configuration files (Augeas) to very rigid, format specific abstractions (INI and HOST).

We propose an approach that provides unified configuration setting access by introducing a key/value abstraction into configuration management systems that can deal with specifics of different configuration file formats through modularization. It combines the flexibility of general purpose methods through a unified interface for different formats, while still preserving validation for specific syntax rules.

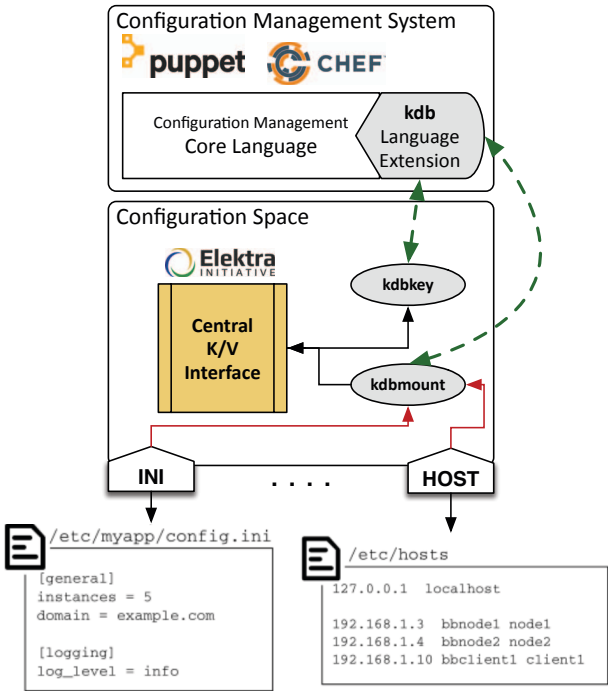

Figure 1: ***Overview of a Unified Configuration Setting Access approach facilitated through configuration language extension Elektra***

3.1 Overview and Workflow

Figure 1 gives an overview of our approach and shows **KDB**, a language extension for configuration management system DSLs, that interfaces with a general purpose configuration library, called **Elektra**.[1] We briefly describe the workflow of our approach:

- The *configuration space* Elektra can be viewed as a central interface using key/value abstractions that provides access to all configuration setting values regardless of their specific formats and allows for their manipulation. Applications can use Elektra to manage configuration settings without ever requiring the use of a configuration file. Also Administrators can directly use Elektra, e.g., to debug the results of a configuration management system run.
- The process of *mounting* is required to lift a particular configuration file into Elektra. As soon as a file is mounted, users can manipulate settings in Elektra and they are transitively manifested through partial manipulation in the file, preserving default settings. The bidirectional transformations from and to configuration files are provided through models in Elektra (i.e., plugins in the general purpose configuration library). Mounting within the configuration management system is only necessary if the application did not already mount the configuration file during installation.
- We extend the existing core language of Puppet by two keywords **kdbkey** and **kdbmount** that correspond to interactions with Elektra.

Dealing with the intricacies of different configuration file formats, including their validation, is now handled by a configuration library with a rich set of existing plugins and modular system. Our contribution is providing unified access of partial manipulation of configuration files within the configuration management system by extending its core language with a well-known key/value abstraction.

To facilitate the definition of all required aspects of a key in Elektra, our **kdbkey** construct, requires the following attributes:

- `name`: defines Elektra's key and defaults to the resource's title.
- `value`: represents the value of a key.
- `ensure`: defines the existence of a key (i.e. present or absent).
- `check`: appends metadata to the key, which are enforced by plugins in Elektra. For example, data types can be used to restrict possible values.

The **kdbmount** construct requires the following attributes to express a valid mount point in Elektra:

- `name`: defines Elektra's key used for mounting and defaults to the resource's title.
- `file`: defines which configuration file is used for that mount point.
- `plugins`: list of Elektra's plugins (together with the plugin's configuration settings).
- `ensure`: defines if the mount point should exist or not.

Listing 6 shows a full example of how a value can be changed within a space-separated INI file.

```
kdbmount { 'system/sw/myapp':
   ensure  => 'present',
   file    => '/etc/myapp/config.ini',
   plugins => {
      ini => { separator => ' '},
      enum => {}
   }
}
kdbkey { 'system/sw/myapp/key':
   ensure  => 'present',
   value   => 'changed'
}
```

Listing 6: Example `kdbmount` and `kdbkey` resource declaration with plugin configuration

[1] https://www.libelektra.org/

3.2 Implementation

Our approach is designed to generalize beyond specificities of concrete systems. The overview figure shows two different implementation options: Puppet and Chef for configuration management tools, as well as INI and HOSTS as different configuration file formats. We chose to implement our initial prototype with Puppet (due to its popularity). The result of this implementation is a Puppet extension that we have published as open source software on GitHub, including more detailed documentation on the aforementioned language constructs (see Section 4.5).

Puppet directly uses the configuration space Elektra. Elektra provides key/value manipulation methods, but it does not implement a key database in the traditional sense. Instead it directly writes into different configuration files of the system. The key database can be seen as virtual file system but with key/values instead of files.

The keyword *kdbkey* enables us to directly manipulate keys and values within configuration files. Plugins in Elektra take care that the structure of the configuration files are preserved.

The keyword *kdbmount* mounts a configuration file to the key given as argument. After mounting, the whole content of the configuration file resides in subkeys of the given key.

The implementation challenge of Elektra is to find a specification language that systematically describes configuration access. The simple-to-use kdbmount is the result of these efforts to find such a specification language, implemented in Elektra [26]. Elektra provides a way to specify a mapping from every configuration file format to key/value pairs by assembling and configuring plugins [28]. Configuration management tools, but also other applications, then use Elektra's plugins for configuration access.

Elektra describes syntactical and semantical content of the files. For some plugins, the specification is predefined. For example, the hosts plugin automatically checks IPv4 and IPv6 addresses for validity. This is the case because the hosts plugin states which data type it expects for which key/value pair. Separate operating-system-dependent network validation plugins then actually enforce valid configuration values.

4 EVALUATION

To evaluate whether the proposed technique has measurable benefits for user's productivity, we designed and executed a randomized controlled trial that is described in detail in this section.

4.1 Research Question and Initial Considerations

We want to answer the following research question:

RQ: Which approach of configuration file manipulation provides the best results if compared in terms of usability and maintainability? Does the proposed solution have an advantage over existing methods?

To properly design the user study, we refine this high level research question by making it more precise and transform it into a testable hypothesis. One indisputable difference between different techniques is the needed time to fulfill a task. The use on development time as a measurement for the benefit of a technique has been applied in multiple experiments in the past (such as for the benefit of a visualizations [6, 13], the benefit of a syntactical construct [32] or the benefit of a programming language construct (see for example [19, 36]).

Therefore, we formulate the following null hypothesis:

H_0**:** There is no difference in completion time for a given maintenance scenario using the approach KDB in comparison to other general-purpose partial manipulation.

Currently, only two general-purpose partial manipulation methods are available: Augeas and KDB. In practice, most manipulation methods are specialized to a single configuration file format. It would be unrealistic to claim that a general-purpose method can be better than these specialized method. Nevertheless, we also added these manipulation methods in the evaluation to see how these tools compare.

Finally, we expect that different subjects differ widely with respect to the time they require to solve a given scenario. As a consequence, it is from our perspective desirable to test each subject under all conditions. In such a so-called within-subject design the absolute differences between solution times of different subjects becomes negligible and only the differences within each subject become relevant for the analysis.

4.2 Scenario Description

The whole experiment consists of three scenarios in two or three variants. This section describes the scenarios in more detail.

4.2.1 **Scenario 1: JSON Configuration for App `calculator`.** The goal of this scenario was to finish a Puppet module that configures the fictional application "`calculator`". This application expects a configuration file in JSON format, with four settings in total. The Puppet module for this application was already predefined, whereas, only the configuration file management was missing, which had to be added by the participants. The main module class "`init.pp`" already defined four parameters, one for each configuration setting. This task had to be solved in two different variants:

- Variant ERB: Resource Type "`file`" and ERB Template: The participants were asked to solve this task by using the resource type "`file`" only. The content of the configuration file should be defined by an ERB template. The task description contained an example of JSON-style configuration file.
- Variant KDB: Resource Types "`kdbmount`" and "`kdbkey`": Now the same task had to be solved using the resource types "`kdbmount`" and "`kdbkey`" only. The "`config.pp`" already suggested a Elektra mount point for the configuration file.

Scenario 2: Hosts File Manipulation. The goal of this scenario was, to write Puppet code to manipulate a Hosts file. Participants were asked to add two new entries and update two existing entries. Each entry consists of an IP-address, a hostname and one alias name. This scenario had to be solved in three different variants:

- Variant HOST: Resource Type "`host`": This variant had to be solved with the Puppet built-in resource type "`host`", which is specially designed to manipulate Hosts files.

- Treatment AUG: Resource Type "augeas": The same task had to be solved with the resource type "augeas" only. The task description contained an example of how the Augeas Hosts lens transforms a Hosts entry to its internal representation.
- Variant KDB: Resource Types "kdbmount" and "kdbkey": This variant for Scenario 2 had to be solved with the resource types "kdbmount" and "kdbkey" only. As for method AUG, the task description contained an example of how the Elektra hosts plugin transforms a Hosts entry to Elektra's internal representation of the configuration space.

Scenario 3: Samba Configuration File Manipulation. The aim of this scenario was to manipulate the existing Samba configuration file /etc/samba/smb.conf, an INI-style configuration file with multiple sections. The participants were asked to modify three settings in the main section "global" and to add two new share sections, each with three settings. The Puppet module code was already predefined and the participants had to extend the empty file "config.pp" with Puppet code to manipulate the configuration file in question. Again, this task had to be solved in three variants.

- Variant INI: Resource Type "ini_setting":
  We asked the participants to solve this task only with the resource type "ini_setting".
- Variant AUG: Resource Type "augeas":
  The same task had to be solved with the resource type "augeas" and its built-in lens for the Samba configuration file. A sample representation of Samba lens was added to the task description.
- Variant KDB: Resource Types "kdbmount" and "kdbkey": This method required to solve this task using the resource types "kdbmount" and "kdbkey". The task description did not say anything about the Elektra storage plugin that can be used or how the internal representation may look like. It was up to the participant to find this information.

4.3 Study Participants

Our experiment was conducted with 14 subjects who are master students recruited from a computer science department. We asked subjects about prior knowledge of the technologies and techniques used in the study to get a better understanding of their skill level in order to avoid introducing bias in the study. We aim to recruit subjects with a variety of backgrounds in software development and system administration. Three subjects had no experience, ten with basic experience and one subject had good system administration experience. None of them was a professional system administrator. Four subjects stated, that they already knew Puppet and have done minor experiments with it. The majority did not know Puppet before this user study. One subject had also experience with another CM-tool[2]). The majority of our participants (10 subjects) stated, that they have good software development experience. 3 subjects were professional software developers and one subject just had basic programming experience. However, 8 subjects have never used Ruby, 4 of them had minimal experience, and 2 use Ruby on a regular basis. The ERB template system was never used before by 13 subjects. However, 4 subjects have used other template systems before. Elektra was not known by 7 subjects, 3 subjects already experimented with it and one was using it on a regular bases. The experiment included 3 participants who are part of the Elektra's developer community.

Augeas was unknown by 12 subjects and only two subjects had already experimented with it. However, XPath, a central concept of Augeas, has been already used by 9 subjects, whereas 5 subjects did not know XPath. In contrast to this, all subjects state that they already used regular expressions once or on a regular basis. All subjects already modified a configuration file by hand. From the subjects, 11 have worked with JSON, 9 with INI-style files, 7 with Hosts, and 4 with YAML.

4.4 Environment and Protocol

When designing a software experiment, it is important to consider and try to control all influencing variables of this experiment. This allows us to measure the effect of one explicitly changed variable (the experiment factor) [37]. Therefore, the experiment environment was designed to control as many influencing variables as possible:

- Same source editor for all participants: We used the Atom[3] text editor in version 1.15.0 with Puppet syntax highlighting and automatic syntax checking.
- A new system environment for each task method: Puppet agent executions of earlier programming tasks do not have any influence on later Puppet agent executions.
- Puppet language and resource type guide: A Puppet introduction guide was written, including important language concepts as well as a reference and examples of all resource types used during the experiment. The content of this guide was presented to all participants before the experiment was started. In addition, it was allowed to use this guide as a reference during the experiment. Therefore, each subject had a printed version of this guide available.
- Same hardware for all experiments: The whole experiment was conducted on equally equipped machines with the same screen size and resolution. We used machines with an Intel Core i5-6600 CPU at 3,30GHz with 16GB RAM connected to monitor with a resolution of 1920 x 1080 and a diagonal viewing size of 60,45 centimeters.
- Automatic time measurements and information recording: The experiment environment was designed to record as much information as possible. For each programming task, the environment recorded the task duration time, as well as additional information such as amount of script executions, source code changes or test results.
- Upper time limit for each programming task: The experiment environment tracked the time for each experiment. Once the upper time limit of 75 minutes (=4500 seconds, determined from pilot data) was reached, the experiment supervisor was notified about this.

Before the experiment was started, all participants were introduced into the Puppet language based on the introduction guide. As already stated earlier, participants were allowed to use this guide as a reference during the whole experiment. In addition to this

[2]Slack CM: https://github.com/jeviolle/slack

[3]https://atom.io/

guide, a task description paper was handed out to all subjects. After the subjects were introduced to Puppet, the experimenter assigned each subject to a group in round-robin fashion to ensure balanced groups and started the experiment environment.

4.5 Study Reproducibility

To ensure reproducibility of our study, we created a replication package containing the environment and setup in the form of a Docker container [5]. Additionally, an anonymized form of the measured data (Section 4.6) in combination with all analysis scripts for the statistics tool R [10] can be found in an online appendix[4]. We also published our implementation as an open source project[5].

4.6 Measurements and Analysis

This section presents the measured results and subsequent analysis. Section 4.6.1 shows all measured results and basic descriptive statistics. Section 4.6.2 presents our analysis steps to determine differences in significance between the used methods.

4.6.1 Descriptive Statistics. Table 1 shows the collected duration times in seconds. Each row contains all measured times for one subject. For example, subject 1 required 604 seconds for Scenario 2 with method HOST and 505 seconds with method KDB.

Based on this data, we computed descriptive statistics, shown in Table 2. The first row contains the sum of all subjects for each task and method. So all subjects together required 23317 seconds for Scenario 1 with method ERB. The maximum, minimum, arithmetic mean, median and standard deviation for each task and method is shown in the next five rows. Based on this, we also calculated the difference between the total task duration time between two methods. For Scenario 1, this is done by calculating the difference between method ERB and method KDB. For Scenarios 2 and 3, we did the same for each method combination. The same calculation was done for the arithmetic mean, which is shown in row number eight. The last row contains the number of occurrences, where the difference between two methods of one subject is less than zero. For example, for Scenario 1 we can see, that 3 subjects required less time to solve the task with method ERB than with method KDB.

A first glance of these calculations already gives us a good overview of our experiment results. The majority of subjects (79%) solved Scenario 1 faster, when they were using our proposed approach. A similar situation can be seen for Scenarios 2 and 3 between method AUG - KDB and HOST/INI - AUG. Here, the situation is even more clear: all subjects (100%) solved Scenarios 2 and 3, when they were using method HOST or INI, than with method AUG. All subjects required less time for Scenario 2, when using method KDB compared to AUG. Only one subject was faster with method AUG compared to KDB for solving Scenario 3. However, if we look in Table 1, we can see, that subject 14 required only 2 seconds more to solve Scenario 3 with method KDB compared to method AUG.

To visualize these first tendencies of duration times between each method, we have created box plots for each scenario. Figure 2 shows the box plot for Scenario 1.

The duration in seconds is shown on the X-axis, whereas on the Y-axis we see the different treatment levels (variants). The box

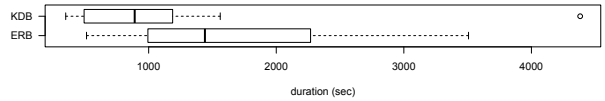

Figure 2: Box plot for Scenario 1

represents the 25%, the 50% (also know as median) and 75%-quantile. This means, half of our duration times is within the box and the other half is outside the box. The marks next to the box, called whiskers, represent the 10% and 90%-quantile [37]. Small circles outside the whiskers show outliers.

The box plot for Scenario 2 (Figure 3) strengthens the assumption, that subjects were usually faster when they used Elektra.

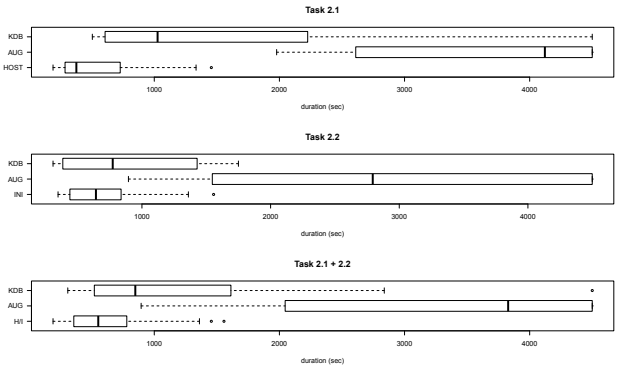

Figure 3: Box plot for Scenario 2

The box plot for Scenarios 2 and 3 (Figure 3) shows even greater differences between the used methods. The box-plots for method AUG are particularly interesting. In both Scenarios 2 and 3, there is no overlap with the other two methods. It even seems that the maximum time limit of 4500 seconds has cut-off at the AUG boxes, as the 75% and the 90%-quantile are both 4500 seconds. This is a result of the fact, that 6 subjects (43%) hit the time limit for Scenario 2 with method AUG and 4 subjects (29%) for Scenario 3. 3 subjects hit the time limit for both scenarios.

This great difference is not seen between method KDB and HOST/INI, whereas the for Scenario 2 we can assume that our subjects were usually faster with method HOST. The situation for Scenario 3 seems to be slightly different. The majority was usually faster when using method INI, however some subjects required less time with method KDB, since the 10% and 25%-quantile of method KDB are both smaller than those of method INI.

4.6.2 Testing for Significant Differences. A classic test method for intra-subject experiments is the paired T-test, which tests if the means of two treatment levels of one subject are not significantly different [37]. However, this statistical test has the precondition, that the differences between the treatment levels for each subject should be normally distributed [20]. Testing a distribution for normality can be performed by the Shapiro-Wilk test [11, 31]. An

[4] Omitted for double blind review - hosted on Zenodo
[5] http://puppet-userstudy-results.libelektra.org

subject	Scenario 1		Scenario 2			Scenario 3		
	ERB	KDB	HOST	AUG	KDB	INI	AUG	KDB
1	1283	557	604	3962	505	710	3276	406
2	3397	948	1456	4500	4500	1557	4500	1750
3	2269	4382	728	4500	2838	953	4215	1703
4	3507	1561	828	3173	1881	1361	3786	1522
5	513	609	191	2114	662	476	1547	384
6	1202	494	298	2458	607	733	1555	410
7	994	844	275	3870	601	600	1483	385
8	1783	350	366	4500	537	394	4500	331
9	658	937	389	2610	800	505	4500	649
10	1789	1130	347	4500	1428	348	2312	1397
11	2371	1188	1334	1978	691	685	1915	1151
12	1599	1247	643	4500	2750	838	4500	1429
13	717	385	287	4281	1253	440	1157	309
14	1235	426	245	4500	2226	385	895	897

Duration values in seconds

Table 1: Experiment results

function	Scenario 1		Scenario 2			Scenario 3		
	ERB	KDB	HOST	AUG	KDB	INI	AUG	KDB
Sum	23317	15058	7991	51446	21279	9985	40141	12723
max	3507	4382	1456	4500	4500	1557	4500	1750
min	513	350	191	1978	505	348	895	309
arith. mean	1665.5	1075.57	570.79	3674.71	1519.93	713.21	2867.21	908.79
median	1441	890.5	377.5	4121.5	1026.5	642.5	2794	773
std. dev.	941.56	1021.02	400.11	990.54	1191.49	365.42	1438.54	561.22
Δ Sum	8259		HOST - AUG: -43455 HOST - KDB: -13288 AUG - KDB: 30167			INI - AUG: -30156 INI - KDB: -2738 AUG - KDB: 27418		
mean diff.	590		HOST - AUG: -3104 HOST - KDB: -949 AUG - KDB: 2155			INI - AUG: -2154 INI - KDB: -196 AUG - KDB: 1958		
#(diff. < 0)	3		HOST - AUG: 14 HOST - KDB: 12 AUG - KDB: 0			INI - AUG: 14 INI - KDB: 8 AUG - KDB: 1		

Table 2: Descriptive Statistics

alternative to the paired T-test is the Wilcoxon signed rank test, which also can be used, if the differences of our samples are not normally distributed [20, 21, 37]. However, Prechtelt [25] mentions that testing for a normal distribution is often a dangerous thing. A statistical test is most meaningful, if the null hypothesis is rejected, i.e., the tested sample is not normally distributed. Furthermore, such tests are rather sensitive, if the sample is small [25].

As a result of this, we have decided to perform the Shapiro-Wilk test to check for a normally distributed sample. Regardless of its result, we performed both hypothesis tests, the paired T-test and the Wilcoxon test, in order to see if the results of both tests largely differ. However, since both results were comparable, we only report the outcome of the Wilcoxon test.

Hypothesis Theorem 1: Null hypothesis H_0 of the Wilcoxon signed rank test: the median difference of the paired samples is zero. Alternative hypothesis: the difference is not zero [21].

Hypothesis Theorem 2: Null hypothesis H_0 of the Shapiro-Wilk normal distribution test: the tested sample is normal distributed. Alternative hypothesis: it is not [11].

The null hypothesis for the Wilcoxon signed rank test can be expressed as: the required effort to solve the scenario with method X is the same as for method Y.

Based on the null hypothesis/alternative hypothesis from our statistical tests, we can make decisions on a certain significant level α. We use a significance level of $\alpha = 0.05$ for the analysis.

Table 3 shows the calculated p-values for each test and each method grouping. The null hypothesis is rejected if the calculated

Scenario	Methods	Wilcoxon test	H_0 rejected	Faster
S1	ERB - KDB	0.02	**yes**[†]	KDB
S2	HOST - AUG	6×10^{-6}	**yes**	HOST
S2	HOST - KDB	0.005	**yes**	HOST
S2	AUG - KDB	4×10^{-4}	**yes**	KDB
S3	INI - AUG	3×10^{-5}	**yes**	INI
S3	INI - KDB	0.7	no	Inconclusive
S3	AUG - KDB	3×10^{-4}	**yes**	KDB

[†] Result of Wilcoxon test used, since test for normality was rejected. However, the result of the paired T-test is very close to our chosen significance level

Table 3: Hypothesis test results

p-value is less than the significance level $\alpha = 0.05$. If this is the case, the difference of the compared methods is statistically significant, i.e., the probability of the alpha error is less than 5% (based on our chosen value for α). If we cannot reject the null hypothesis, the probability of the alpha error is greater than 5%. The decisions to reject or accept H_0 is based on the Wilcoxon signed-rank test, as we had to reject the null hypothesis for the normality test in all cases.

As a result from these tests, we are able to conclude, that there are, with exception of two cases, always significant differences between the Puppet file manipulation methods used during this user study. We highlight these findings with an additional column ("Faster") that indicates which of the approaches is significantly faster.

From the box plot for Scenario 1 (Figure 2) we can assume, that our subjects usually were faster when they used method KDB to solve this task. The hypothesis test underpins this assumption, as we can also reject the null hypothesis based on the Wilcoxon signed-rank test.

The box plots for Scenarios 2 and 3 show that there are large differences between the times spent to solve these scenarios when subjects used method AUG or the specialized methods HOST or INI. The statistical hypothesis tests clearly underpinned this initial observation. A similar situation, however not that dramatically, is seen between the methods AUG and KDB. Also for this case we can conclude that the difference between these two methods is significant. Therefore, our subjects had to spend much more time to solve Scenarios 2 and 3 when they were using method AUG in contrast to the other two methods.

The remaining method combination for Scenarios 2 and 3 . Based on the Wilcoxon test we can conclude, that for Scenario 2 the difference between method HOST and KDB is significant. Thus, our subjects required more time to solve Scenario 2 when they were using method KDB. The same is not true for Scenario 3. Here we can see, that our statistical hypothesis test does not reject the null hypothesis. This means, that there is no significant difference between these two methods. However, directly concluding that the opposite is true and both methods are equal would be a wrong and dangerous conclusion. A statistical hypothesis test gives us a clear statement, if the null hypothesis is rejected [25]. From the box plot in Figure 3 we already can see this behaviour. Usually our subjects required more time to solve Scenario 3 with method KDB in contrast to method INI. However, there are cases where subjects were faster with method KDB. Also, the fastest time to solve Scenario 3 was achieved with method KDB. This makes the result of this combination inconclusive. It seems that there are more factors involved in this situation, which require more investigation.

4.7 Threats to Validity

While we carefully considered our experiment design to reduce threats to validity as far as possible, there are still a number of limitations that need to be discussed.

4.7.1 External Validity. To have general conclusions, it is essential that the approach can be generalized to other configuration management system. To address this issue we implemented further prototypes of our proposed approach for Chef and Ansible. The prototypes showed that setting key/values within these configuration management systems would work similarity. Therefore, we assume that user studies conducted with the other configuration management systems would lead to similar results.

An important matter is whether the scenarios are representative. In the study design, we particularly focused on realistic tasks. We used different types of applications and configuration file formats for exemplary scenarios.

Although rare, some configuration files are not only data but may be mixed with scripts. While such configuration files also can be transformed to key/values by executing the script, manipulating configuration settings within such configuration files may interfere with the present code. We exclude such configuration files with scripts from our claims.

Other configuration management systems work fundamentally different in many aspects. When it comes to configuration file manipulation, however, they use similar methods, like Augeas. So we think that our results are generalizable to other configuration management systems which also include Augeas.

4.7.2 Internal Validity. Documentation handed to the participants influences the subjects' performance. To mitigate this issue, we only gave essential documentation of similar length for each method.

Within-subject design easily produces learning outcomes that influence succeeding tasks. We addressed this problem by randomly assigning the methods within the scenarios.

Helping participants might influence time in an uncontrolled way. Thus, we did not provide any help while subjects worked on the scenarios. Participants, however, might get frustrated if they cannot finish tasks. Thus we provided them with the solution after they reached the timeout of 4500 seconds.

Different environments change outcomes in different ways. Next to a controlled lab environment, we also fixed the used editor and the window layout.

The completion of a task was automated by tying it to success of a unit test. While the unit tests were written with great care, it is possible that suboptimal solutions were still accepted. Furthermore, there is a danger that participants used prohibited methods. We addressed these issues by manual inspection of samples.

4.7.3 Construct Validity. Users of configuration management systems might use a convenience layer and might not directly use a generic method as proposed by Augeas (method AUG) or our approach (method KDB). From our experience, we rarely see this

happen. For example, Augeasproviders[6] lists only 17 such modules compared to over 200 Augeas lenses. Thus we claim that our time measurement, where we directly used generic methods is realistic.

5 DISCUSSION

The results of this user study showed, that there are significant productivity differences between the compared Puppet configuration file manipulation methods.

Key/Value abstractions vs Templating and Partial Manipulation. The experiment results for Scenario 1 indicated, that the proposed solution has a significant impact on development productivity and therefore, helps to reduce the required development time. At least this result is valid for more complex configuration file formats, such as the used JSON format. Subjects often reported, that they have struggled with syntactical issues while they were implementing the ERB template for Task 1 using method ERB.

The user study also showed, that method AUG is difficult to use. On average subjects required more than 6 times longer to complete Scenario 2 with method AUG compared to method HOST and more the 2 times longer compared to method KDB. This is a result of the complexity of the resource type "`augeas`". Users have to be aware of its XPath like notion of specifying key names, the specific transformations to its internal representation of the configuration values and specific behaviour of the used lens. Especially new Puppet users require a lot of time to use this method effectively.

Usability in Specific Interfaces vs Flexibility in Generalized Interfaces. Scenarios 2 and 3 were designed to compare four different partial file manipulation methods. Two very specific methods, HOST and INI and two general methods, KDB and AUG. Our expectation was that the specific methods would require a shorter time to complete the tasks, as both resource type APIs are adopted to their underlying configuration file format. The resource type "`host`" of method HOST has three main elements that reflect the elements of a Hosts entry: "`name`", "`ip`" and "`host_aliases`". Therefore, it is very clear to a user, how to write the resource declaration to manage one particular Hosts entry with given values. The more general methods are at a disadvantage here, as users have to figure out which keys have to be manipulated.

The situation for method INI is similar.
The resource type "`ini_setting`" has four main attributes: "`path`" to specify the target configuration file, "`section`", "`setting`" and "`value`". This makes it also very clear to a Puppet user how to write a resource declaration to manipulate a specific setting within a defined INI configuration file.

Our expectations were confirmed by the results of the user study. Subjects usually required less time to implement Scenarios 2 and 3 when they used a format specific method. The measured time differences between method HOST and the methods KDB and AUG are significant. Therefore, we can conclude that Puppet users will be most productive when they use the resource type "`host`" to manipulate Hosts entries.

The measured difference between method INI and KDB shows a similar trend. However, this difference is not significant. In fact, some subjects (43%) required less time using method KDB compared to method INI.

[6] http://augeasproviders.com/providers/

6 RELATED WORK

Other configuration libraries, such as Apache Commons Configuration [9, 24], differ from our chosen approach by not being introspectable via configuration management tools. Thus they do not provide a key/value interface for configuration and specifications.

We are not aware of any existing system (both in academia and industry) that has attempted to introduce unified access to a variety of configuration files by a modular specification language. We thus give an overview of related approaches to reduce misconfigurations.

Xu et al. [41] surveyed approaches to reduce misconfiguration. Nagaraja et al. [22] tries to find misconfiguration but avoids a completely duplicated production environment.

ConfErr [14] tries to inject misconfiguration into applications. However, because it is not guided by any specifications, it has little chances in actually finding misconfigurations. Spex [40] improved this idea and guides the process of finding misconfiguration by source code analysis. Spex complements our approach as it is used for construction of validation rules.

AutoBash [33] and ConfAid [1] require availability of predicates testing the application on the productive system.

Xu et al. [38] argue that configuration settings must be reduced in the first place. We agree with this notion and see it as necessary step for sustained reduction of misconfiguration.

Zhang et al.[42] try to detect problematic error messages.

7 CONCLUSION

Existing methods for configuration file manipulation in configuration management systems suffer from a variety of problems. Format-specific methods of managing configurations use different abstractions for every format and have limited expressibility, while general purpose methods of editing configuration lead to hampered productivity due to the higher likelihood of introducing errors.

We proposed an approach that unifies the best of both worlds by extending the core language of configuration management systems with language constructs that enable partial manipulation of configuration files through well-known key/value abstractions. The newly introduced language constructs provide unified configuration setting access as a general-purpose method, while still providing validation and preservation of default settings.

We conducted a user study showing that developers using our approach are significantly faster in common configuration management scenarios compared to other general-purpose configuration manipulation methods (ERB Templates and Augeas). Specialized manipulation methods, such as "`HOST`" or "`INI`" not surprisingly have an advantage in absolute task times to provide a solution to the scenario. However, statistical testing shows, for instance, that the results of comparing our approach with "`INI`" is inconclusive, i.e., it does not show significant differences in productivity.

ACKNOWLEDGEMENT

The last author was funded by the Swiss National Science Foundation (SNSF) under project Automated Synthesis and Repair of Infrastructure Code (no. 178036).

REFERENCES

[1] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–11.

[2] Aaron Bohannon, J Nathan Foster, Benjamin C Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: resourceful lenses for string data. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 407–419.

[3] Mark Burgess. 1995. CFEngine: a site configuration engine. In *The USENIX Association, Computing Systems*, Vol. 8.

[4] Mark Burgess. 2003. On the theory of system administration. *Science of Computer Programming* 49, 1âĂŞ3 (2003), 1 – 46. https://doi.org/10.1016/j.scico.2003.08.001

[5] Jürgen Cito, Vincenzo Ferme, and Harald C Gall. 2016. Using Docker containers to improve reproducibility in software and web engineering research. In *International Conference on Web Engineering*. Springer, 609–612.

[6] Jürgen Cito, Philipp Leitner, Martin Rinard, and Harald C. Gall. 2019. Interactive production performance feedback in the IDE. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 971–981. https://doi.org/10.1109/ICSE.2019.00102

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.

[8] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How Do API Documentation and Static Typing Affect API Usability?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 632–642.

[9] Apache Software Foundation. [n.d.]. https://commons.apache.org/configuration/. Accessed February 2017.

[10] The R Foundation. 2017. *R: The R Project for Statistical Computation.* https://www.r-project.org/

[11] Asghar Ghasemi and Saleh Zahediasl. 2012. Normality Tests for Statistical Analysis: A Guide for Non-Statisticians. *International Journal of Endocrinology and Metabolism* 10, 2 (2012), 486âĂŞ489. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3693611/

[12] John A Hewson, Paul Anderson, and Andrew D Gordon. 2012. A Declarative Approach to Automated Configuration.. In *LISA*, Vol. 12. 51–66.

[13] Niklas Hollmann and Stefan Hanenberg. 2017. An Empirical Study on the Readability of Regular Expressions: Textual Versus Graphical. In *IEEE Working Conference on Software Visualization, VISSOFT 2017, Shanghai, China, September 18-19, 2017*. 74––84. https://doi.org/10.1109/VISSOFT.2017.27

[14] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *Dependable Systems and Networks With FTCS and DCC, 2008*. IEEE, 157–166.

[15] Spencer Krum, William Van Hevelingen, Ben Kero, James Turnbull, and Jeffrey McCune. 2013. *Pro Puppet* (second edition ed.). Apress.

[16] Puppet Labs. 2016. *Puppet 4.8 reference manual âĂŤ Documentation âĂŤ Puppet.* https://docs.puppet.com/puppet/4.8/reference/index.html

[17] James Loope. 2011. *Managing Infrastructure with Puppet: Configuration Management at Scale.* " O'Reilly Media, Inc.".

[18] David Lutterkort. 2008. AUGEAS - a configuration API. In *Proceedings of the Linux Symposium*, Vol. 2. 47–56. http://www.landley.net/kdocs/ols/2008/ols2008v2-pages-47-56.pdf

[19] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Eric Tanter, and Andreas Stefik. 2012. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. 683––702. https://doi.org/10.1145/2384616.2384666

[20] John McDonald. 2015. Paired T-Test, Handbook of Biological Statistics. (7 2015). http://www.biostathandbook.com/pairedttest.html

[21] John McDonald. 2015. Wilcoxon Signed-Rank Test, Handbook of Biological Statistics. (7 2015). http://www.biostathandbook.com/wilcoxonsignedrank.html

[22] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2004. Understanding and Dealing with Operator Mistakes in Internet Services.. In *In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Vol. 4. 61–76.

[23] Stephen Nelson-Smith. 2013. *Test-Driven Infrastructure with Chef: Bring Behavior-Driven Development to Infrastructure as Code.* " O'Reilly Media, Inc.".

[24] Milan Nosál and Jaroslav Porubän. 2012. Supporting multiple configuration sources using abstraction. *Open Computer Science* 2, 3 (2012), 283–299.

[25] Prechelt, Lutz. 2000. *Kontrollierte Experimente in der Softwaretechnik: Potential und Methodik.* Springer.

[26] Markus Raab. 2016. Elektra: universal framework to access configuration parameters. *The Journal of Open Source Software* 1, 8 (dec 2016). https://doi.org/10.21105/joss.00044

[27] Markus Raab. 2016. Improving System Integration Using a Modular Configuration Specification Language. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 152–157. https://doi.org/10.1145/2892664.2892691

[28] Markus Raab. 2016. Improving System Integration Using a Modular Configuration Specification Language. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 152–157. https://doi.org/10.1145/2892664.2892691

[29] Markus Raab and Gergö Barany. 2017. *Challenges in Validating FLOSS Configuration.* Springer International Publishing, Cham, 101–114. https://doi.org/10.1007/978-3-319-57735-7_11

[30] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering (TSE)* (2017).

[31] S. S. Shapiro and M. B. Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3/4 (1965), 591–611.

[32] Samuel Spiza and Stefan Hanenberg. 2014. Type names without static type checking already improve the usability of APIs (as long as the type names are correct): an empirical study. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*. 99––108. https://doi.org/10.1145/2577080.2577098

[33] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving Configuration Management with Operating System Causality Analysis. (2007), 237–250. https://doi.org/10.1145/1294261.1294284

[34] Blazej Swikecicki and Leszek Borzemski. 2018. *How Is Server Software Configured? Examining the Structure of Configuration Files.* Springer International Publishing, 217–229. https://doi.org/10.1007/978-3-319-67220-5_20

[35] James Turnbull. 2007. *Pulling Strings with Puppet.* Apress. https://doi.org/10.1007/978-1-4302-0622-4

[36] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An empirical study on the impact of C++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 760––771. https://doi.org/10.1145/2884781.2884849

[37] Claes Wohlin, Per Runeson, Martin HÃűst, Magnus C. Ohlsson, BjÃűrn Regnell, and Anders WesslÃĺn. 2000. *Experimentation in Sortware Engineering: An Introduction.* Kluwer Academic Publishers.

[38] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-designed Configuration in System Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 307–319. https://doi.org/10.1145/2786805.2786852

[39] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, USA.

[40] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 244–259.

[41] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4 (July 2015), 70:1–70:41. https://doi.org/10.1145/2791577

[42] Sai Zhang and Michael D. Ernst. 2015. Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/2771783.2771817