Early Detection of Configuration Errors by Learning Configuration Rules

Martin Schweighofer[a]

a TU Wien, Austria

Abstract Configuration errors are a major cause of system failures and therefore may also lead to financial loss. Thus, it is desirable that configuration errors get detected before runtime of a software system. One approach to achieve this is to automatically derive rules from the configurations of other deployed systems and to use these rules to verify the configuration of a system before it gets used. We performed a literature study and identified two frameworks proposed in recent research, EnCore and ConfigV, which follow the approach. EnCore focuses on integrating information of the system environment and deriving correlation rules concerning the configuration parameters and the environment information. ConfigV on the other hand uses a probabilistic type system to determine configuration parameter types and derives rules based on the correlation between the configuration entries and between their values. ConfigV afterwards orders the rules by importance based on rule graph analysis. Evaluating the frameworks shows that both approaches detect real-world configuration errors. The false positive rate is also acceptable for use in practice. Therefore, the discussed rule-learning frameworks show great promise for helping to prevent misconfigurations.

ACM CCS 2012

- *General and reference → Surveys and overviews;*
- *Computing methodologies → Rule learning;*
- *Software and its engineering → Software configuration management and version control systems; System administration;*

Keywords misconfiguration, configuration errors, machine learning, configuration verification

Seminar aus Programmiersprachen

Perspective The Empirical Science of Programming

Area of Submission Data mining and machine learning for programming

1 Introduction

Modern software systems usually have a large number of configuration settings [9]. These configuration parameters are used in order to guarantee the reusability and customizability of a software system. They may provide information about the environment of a running software system, which is needed for it to function properly (e.g. the location of a used service). Other configuration parameters influence non-functional properties of a software system like performance, security, reliability, and availability [11].

Thus, configuration errors may lead to failure of software systems. For example, in 2015, Facebook and Instagram became inaccessible because of a configuration change [6]. Such service downtimes may lead to high financial loss [11].

According to Xu et al., configuration settings are often not validated before runtime or even before they are used [10]. Without further measures, configuration errors then only get detected when the running system does not perform as expected. To prevent system failures, it is thus desirable that configuration errors get detected before runtime of the software system. One promising approach to achieve this is to infer rules from a set of (mostly) valid configuration files of other deployed systems, called training set. The inferred rules can then be used to detect anomalies in the configuration of a target system. By doing a literature study, we found two frameworks proposed in recent research work that follow this approach: EnCore by Zhang et al. [13] and ConfigV by Santolucito et al. [7]. The aim of this paper is to give the reader an overview of these approaches and how they work, as well as to discuss the differences, advantages and shortcomings of these solutions. Therefore, in Sections 2 and 3, we will describe the proposed frameworks individually. In Section 4, we compare and evaluate both solutions. In Section 5, we discuss related work in misconfiguration detection. In Section 6, we summarize the content of this paper and discuss possible further research.

2 EnCore

As stated in Section 1, configuration parameters often refer to resources or other information of the environment. For example, a software component may use a configuration setting which specifies the directory used for storing logs. If a regular file is set as value of the setting instead of a directory, this should be detected as an error. Furthermore, multiple configuration settings might also be dependent on each other. For example, a software system may use one configuration setting to specify the location of a used resource (e.g. file) and another setting to specify its owner. In case of a mismatch between the actual owner and the owner according to the configuration, this may result in a permission error. To detect such problems it is necessary to consider correlations between configuration parameters and the environment information. Zhang et al. thus proposed EnCore [13], a misconfiguration detection framework based on rule-learning, which considers environment information and correlations between configuration parameters. In order to learn configuration rules and detect anomalies in target systems based on these rules, EnCore generally performs four steps, which are described in more detail in the following.

2.1 Collecting data

The data collector retrieves all the relevant configuration and environment information of the systems of the training set. The output of this step is the raw data, which is analyzed in the next step.

2.2 Assembling data

First, the data assembler parses the configuration files of the training set and saves the configuration parameters in an intermediate representation. The parser is based on the configuration parser Augeas [3]. Furthermore, general information about the operating system environment (e.g. OS version) of each system is also stored.

EnCore then infers a type for every found configuration parameter. This is done in two steps: First, the potential types are determined by syntactic pattern matching. After that, for each type candidate, EnCore verifies semantically whether the type is fitting by checking the external resources. An overview of types is given in Section A.

After the types are determined, the data assembler augments every configuration parameter which references known environment resources with further information from the environment. These augmented attributes are then added to the intermediate representation.

For example, a configuration parameter whose value begins with a letter and afterwards contains an arbitrary amount of letters or numbers is recognized as being potentially a user name. This is semantically verified by examining the /etc/passwd file of the system, which contains its users. If it is indeed a user name, EnCore adds further attributes, e.g. one which names the group of the user, and stores them like other configuration parameters. Assuming the name of the configuration parameter is account, the name of the stored attribute could be e.g. account.group.

2.3 Inferring rules

To infer rules, EnCore uses a template-based approach, which enables the consideration of correlation. The approach is based on the previously assigned types. A template specifies a relation between two previously stored entries and their types. A template could for example specify that, given a parameter of type UserName and a parameter of type FilePath, the user owns the file. An overview of the available templates is given in Section A. For each template, EnCore tries every possible initialization of the template based on the entries and checks whether it is valid (e.g. whether the user in fact owns the file). In this case, the instance is treated as a candidate for a concrete rule. The rule candidates whose involved entries do not appear often enough in the training set or are not valid often enough (determined by a given threshold), are filtered out. Also filtered out are rule candidates which contain configuration entries with values that appear very often across the training set, as the resulting correlation does not contain a lot of information. The remaining rules are used in the next step to detect misconfiguration in the system to be examined.

2.4 Detecting anomalies

The data of the target system gets collected and assembled as described previously. Afterwards, the anomaly detector emits warnings for each configuration parameter with a name which has not been in the training set, for each violation of the learned correlation rules and also for each configuration value which has another type than the type which has been inferred for the training set. Furthermore, EnCore reports values of configuration entries which have not been seen before. In case this affects multiple entries, entries with less diverse values in the training set are ranked as more likely to produce system failures.

3 ConfigV

While EnCore is generally able to detect type violations, correlation violations, unknown configuration parameters and suspicious values, it is not able to detect other types of errors, particularly missing configuration parameters and ordering errors (i.e., when configuration parameters are denoted in a configuration file in the wrong order). It is also not able to determine more complex relations between integer configuration values, e.g. that one value must be larger than the result of the multiplication of two other values. The violation of such rules can, in practice, lead to e.g. performance issues (see e.g. [4]). In order to be able to detect such error types as well, Santolucito et al. propose the framework ConfigV [7], which is based on ConfigC by the same authors [8]. The learning process, which can be divided into three major steps (assembling data, inferring rules, analyzing the rule graph), outputs the rules which have to hold in a valid configuration. To verify the configuration of the target system, ConfigV then checks whether all the derived rules hold for that configuration. In the following, the three learning steps are explained in more detail.

3.1 Assembling data

The input of the learning process are the configuration files of the training set. In the first step, each configuration file gets parsed and the key-value pair of each configuration parameter gets stored in an intermediate representation. Furthermore, the type of each configuration parameter gets determined. Since one value of a parameter may not allow to infer its type (e.g. booleans may be encoded in a configuration language by 0 and 1, i.e. numbers, and/or 'true' or 'false', i.e. strings), ConfigV determines, for every configuration parameter and type, how many instances of the values of the parameter in the training set are potentially (syntax-wise) of the given type. This results in probabilistic type information for each parameter. Based on this information, ConfigV tries to infer a type, e.g. via a threshold on the number of occurrences of a potential type or the percentage it occurs.

3.2 Inferring rules

To infer rules, ConfigV first derives predicates from the set of parameters. These predicates have arity 2 and both supplied terms are sets of configuration parameters, the first being called source set, and the second being called target set. Generally, the predicate rules specify that given the elements of the source set occur in a configuration file, some relation between the source set and the target set has to hold. Predefined predicate types in ConfigV are the type predicates, the equality predicate, the order predicate, the missing predicate and the comparison predicates. An overview of the predicates is also given in Section B.

The type predicates state that a configuration parameter, if defined, has to be of a certain type and are inferred when the corresponding type for the parameter could get determined in the previous step. The equality predicate is inferred for every pair of configuration parameters of the same type and states that the two parameters have the same values. The order predicate is derived for each pair of configuration parameters in one configuration file and states that the first parameter in the pair is denoted before the second. The missing predicate is derived for every pair of configuration parameters and states that if the first parameter exists in one configuration file, the second also has to exist in the same file. For each integer comparison operator ($<$, $>$, $=$) and each pair of configuration parameters of type integer, a predicate is inferred which states that if the first parameter is defined, the comparison operation of the two parameter values has to yield true. Similarly, comparison predicates with three configuration parameters of type integer are derived, which state that if the first two parameters are defined, comparing the third value with the result of the multiplication of the first two values yields true.

Afterwards, for each predicate, its confidence, which is the percentage of how often its specified relation holds in the training set, is determined. The predicates whose confidence do not reach a certain threshold are filtered out, as well as predicates whose involved configuration parameters do not occur often enough. The remaining predicates determine the rules which have to hold in the target system.

3.3 Analyzing the rule graph

To further improve the output of the misconfiguration checking, the rules are analyzed using concepts from graph theory. A rule graph is a directed hypergraph, which is a graph that not only allows edges between vertices, but also between sets of vertices. The vertices of the rule graph are the configuration parameters. Edges are constructed for each predicate rule, directing from the source parameter set to the target parameter set, and are labeled with the predicate name and weighted with the confidence percentage.

Rule graphs are used in ConfigV to rank the determined rules regarding their likelyhood to actually specify constraints whose violation results in errors. In order to achieve this, the notion of a degree of a vertex is introduced. The degree of a vertex is the sum of all weights of the edges where the vertex is in the source set plus the sum of all weights of the edges where the vertex is in the target set. An example for the calculation of a degree of a vertex in a rule graph is given in Section C. The authors of ConfigV assume that rules involving configuration parameters with low degree are more likely depicting technical necessities than conventions in the industry. Therefore, their violation more likely results in errors and thus, ConfigV uses this metric to order the rules by importance.

4 Comparison and Evaluation

EnCore and ConfigV share some similarities. In particular, they infer types to avoid unnecessary computations as well as false positives, and derive rules in order to express correlations between configuration values. There are however some significant differences. For instance, the type inference mechanism differs greatly. The respective mechanism in EnCore has more focus on detecting resource types like files, users, etc. and semantically verifies whether the resources exist by checking the environment information, adding, in case of success, further information from the environment. ConfigV does not use any environment information at all and instead focuses on syntactically recognizing types like booleans and integers by using a probabilistic approach to infer the type based on all the values in the training set.

Similarly, the template-based approach of EnCore allows to recognize correlations between the resources of the system referenced to by configuration parameters. On the other hand, the focus of ConfigV is to recognize patterns derivable from the configuration files itself. It can detect missing entries, ordering errors and more complex integer relations, all of which EnCore is not able to detect. Thus, while EnCore is more suitable to detect problems with regards to the referenced environment resources, ConfigV is more likely to detect performance problems of the software system due to misconfiguration, since these are often caused by multiple integer configuration paramaters whose values violate recommended constraints [7].

Furthermore, ConfigV uses the notion of rule graph analysis to refine the output of the framework. In particular, the analysis is used to order the reported rule violations by the likelyhood they actually represent an error. EnCore does not use rule graphs.

The authors of both EnCore and ConfigV evaluate their proposed systems by using MySQL configuration files from several internet sources as a test set [7, 13]. The systems learn the rules beforehand from a larger training set of industry configuration files (187 in case of EnCore, 256 in case of ConfigV). Their evaluations show that both systems are generally able to accurately detect configuration errors in real-world configuration files. The authors of EnCore thereby show that some of the reported errors could not have been recognized without its incorporation of environment information, while the authors of ConfigV show that their probabilistic type system was able to filter out a significant amount of false-positives and that the rule graph analysis marks rules whose violation results in errors known by the authors as more important. They further show that ConfigV is able to detect errors concerning more complex integer parameter relations, which are not possible to detect in EnCore.

Bessey et al. note that the false positive rate of static analysis tools should not be higher than 30 percent, as otherwise users do not rely on the output of such tools [1].

The authors of EnCore report a false positive rate of 13 percent in their evaluation using MySQL configuration files, while the authors of ConfigV estimate the false positive rate to be 11-18 percent. Thus, both tools, while not yielding perfect results, seem promising to be useful in practice.

We, however, have to keep in mind that automatic misconfiguration detection approaches like EnCore and ConfigV cannot detect every type of misconfiguration. Yin et al. note that around 50 percent of configuration errors are legal misconfigurations, where the configuration parameters generally have valid values, but in reality do not represent the actual intention of the user [12]. To prevent such errors, Yin et al. suggest more user training or an improved configuration design.

5 Related Work

Some of the other work regarding misconfiguration detection focuses on languages to specify constraints on configuration parameters. For example, Huang et al. propose ConfValley [2], which provides a language to declaratively specify configuration constraints in cloud systems. As a more general solution, SpecElektra [5] is a modular configuration specification language, which allows the user to specify constraints on the configuration parameters and its values. The user can include custom checks via plugins. While the validation of the configuration is possible before runtime of the system, these solutions require a user to manually specify the constraints.

Another framework proposed to automatically detect misconfigurations before runtime is PCheck by Xu et al. [10]. It is based on analysis of the source code, while this paper focused on approaches to detect misconfiguration based on configurations of other systems. Other work focuses on diagnosing misconfiguration during or after a system is running. Xu et al. surveyed different strategies [11].

6 Conclusion

We discussed EnCore and ConfigV, two frameworks proposed in recent research work with the aim to automatically detect configuration errors based on learning rules from configuration of other systems. While the focus of EnCore mainly lies on incorporating information from the system environment and deriving correlation rules incorporating configuration parameters and environment resources, the aim of ConfigV is to detect more complex relationships between the configuration parameters and their values themselves. For both approaches, it has been shown that they are able to detect real-world configuration errors while having an acceptable false positive rate.

Due to the different focuses of the both frameworks, some types of configuration errors can only be recognized by one of the two frameworks. Further work could thus focus on integrating environment information into the approach followed by ConfigV. Another interesting topic for future work would be the automatic generation of configuration specifications for SpecElektra based on the output of the discussed frameworks.

References

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”. In: *Commun. ACM* 53.2 (Feb. 2010), pages 66–75. ISSN: 0001-0782. DOI: 10.1145/1646353.1646374.

[2] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. “ConfValley: A Systematic Configuration Validation Framework for Cloud Services”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 19:1–19:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741963.

[3] David Lutterkort. “Augeas—A configuration API”. In: *Proceedings of Linux Symposium*. 2008, pages 47–56.

[4] *my.cnf configuration in mysql 5.6.X*. https://serverfault.com/questions/628414/my-cnf-configuration-in-mysql-5-6-x. [Online; accessed 7-November-2018]. 2014.

[5] Markus Raab. “Improving System Integration Using a Modular Configuration Specification Language”. In: *Companion Proceedings of the 15th International Conference on Modularity*. MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pages 152–157. ISBN: 978-1-4503-4033-5. DOI: 10.1145/2892664.2892691.

[6] Jenni Ryall. *Facebook, Tinder, Instagram suffer widespread issues*. https://mashable.com/2015/01/27/facebook-tinder-instagram-issues/. [Online; accessed 7-November-2018]. 2015.

[7] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. “Synthesizing Configuration File Specifications with Association Rule Learning”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 64:1–64:20. ISSN: 2475-1421. DOI: 10.1145/3133888.

[8] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. “Probabilistic Automated Language Learning for Configuration Files”. In: *Computer Aided Verification*. Edited by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pages 80–87. ISBN: 978-3-319-41540-6.

[9] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. “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*. ACM. 2015, pages 307–319.

[10] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. “Early Detection of Configuration Errors to Reduce Failure Damage”. In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI’16. Savannah, GA, USA: USENIX Association,

2016, pages 619–634. ISBN: 978-1-931971-33-1. URL: http://dl.acm.org/citation.cfm?id=3026877.3026925.

[11] Tianyin Xu and Yuanyuan Zhou. "Systems approaches to tackling configuration errors: A survey". In: *ACM Computing Surveys (CSUR)* 47.4 (2015), 70:1–70:41.

[12] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. "An Empirical Study on Configuration Errors in Commercial and Open Source Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pages 159–172. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043572.

[13] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. "EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection". In: *SIGARCH Comput. Archit. News* 42.1 (Feb. 2014), pages 687–700. ISSN: 0163-5964. DOI: 10.1145/2654822.2541983.

A Types and Templates of EnCore

Table 1 An overview of the types in EnCore. Also included are the respective (simplified) patterns for the synctactic matching as well as the resources used for semantic verification. Taken from [13].

Types	Syntactic	Semantic
FilePath	/.+(/.+)*	File System
UserName	[a-zA-Z][a-zA-Z0-9_]*	/etc/passwd
GroupName	[a-zA-Z][a-zA-Z0-9_]*	/etc/group
IPAdress	[\d]{1,3}(.[\d]{1,3}){3}	N/A
PortNumber	[\d]+	/etc/services
FileName	[\w _-]+.[\w _-]+	File System
Number	[0-9]+[.0-9]*	N/A
URL	[a-z]+://.*	N/A
PartialFilePath	/?.+/(.+)*	File System
MIME Types	[\w/-.]+	IANA
Charset	[\w]+	IANA
Language	[a-zA-Z]2	ISO 639-1
Size	[\d]+[KMGT]	N/A
Boolean	Values Set	N/A
String	N/A	N/A

Table 2 An overview of the templates in EnCore. Taken from [13].

Template	Description
[A<AnyTypeA>] == [B<AnyTypeA>]	An entry should be equal to another entry of same type
[A<AnyTypeA>] = [B<AnyTypeA>]	One instance of an entry should equal to at least one instance of another entry of same type
[A<ExtBoolean>] -> [B<Boolean>]	An extended boolean indicates a boolean entry whose extended attribute has boolean value
[A<IPAddress>] < [B<IPAddress>]	An entry of IPAddress is a subnet of another entry
[A<FilePath>]+[B<FileName>]=><FilePath>	Concatenation of a file path entry with a partial file path entry forms a full file path
[A<String>] < [B<String>]	An entry is substring of another entry
[A<UserName>]<[B<GroupName>]	The user name belongs to the group name
[A<FilePath>] != [B<UserName>]	The file path is not accessible by the user specified in the entry
[A<FilePath>] => [B<UserName>]	The entry of UserName is the owner of the file path specified in the entry A
[A<Number>] < [B<Number>]	The number in one entry is less than that of the other entry
[A<Size>] < [B<Size>]	The size in one entry is smaller than that of the other entry

B Predicates in ConfigV

Figure 1 An overview of the predicates in ConfigV. The judgements show under which circumstances predicates are derived. Figure from [7].

$$\frac{k_1 :: bool}{isbool([k_1],[k_1]) :: Rule}\ \textsc{bool} \qquad \frac{k_1 :: int}{isint([k_1],[k_1]) :: Rule}\ \textsc{int}$$

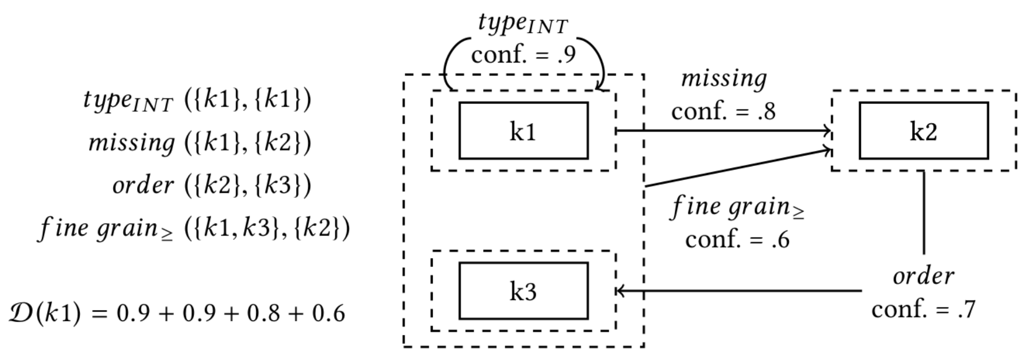
$$\frac{}{missing([k_1],[k_2]) :: Rule}\ \textsc{missing} \qquad \frac{k_1, k_2 :: int}{compare([k_1],[k_2]) :: Rule}\ \textsc{coarse\_grain}$$

$$\frac{k_1, k_2, k_3 :: int}{compare([k_1, k_2],[k_3]) :: Rule}\ \textsc{fine\_grain} \qquad \frac{k_1, k_3 :: size \qquad k_2 :: int}{compare([k_1, k_2],[k_3]) :: Rule}\ \textsc{fine\_grain}$$

$$\frac{k_1 :: \tau \qquad k_2 :: \tau}{eq(k_1, k_2) :: Rule}\ \textsc{eq} \qquad \frac{k1, k2 \in C \qquad k1 \neq k2}{ord(k1, k2) :: Rule}\ \textsc{order}$$

C Example Rule Graph

Figure 2 An example rule graph that was constructed from the predicates on the left (with arbitrary confidence values). The figure also demonstrates how to calculate the degree. Figure from [7].


$type_{INT}$ $(\{k1\}, \{k1\})$
$missing$ $(\{k1\}, \{k2\})$
$order$ $(\{k2\}, \{k3\})$
$fine\ grain_{\geq}$ $(\{k1, k3\}, \{k2\})$
$\mathcal{D}(k1) = 0.9 + 0.9 + 0.8 + 0.6$
$type_{INT}$
conf. = .9
$missing$
conf. = .8
$fine\ grain_{\geq}$
conf. = .6
$order$
conf. = .7
k1
k2
k3

About the author

Martin Schweighofer is master student at TU Wien, studying Software Engineering & Internet Computing. Contact him at e1426365@student.tuwien.ac.at.