Probabilistic Automated Language Learning for Configuration Files

Abstract. Software failures resulting from configuration errors have become commonplace as modern software systems grow increasingly large and more complex. The lack of language constructs in configuration files, such as types and grammars, has directed the focus of a configuration file verification towards building post-failure error diagnosis tools. In addition, the existing tools are generally language specific, requiring the user to define at least a grammar for the language models and explicit rules to check. In this paper, we propose a framework which analyzes datasets of correct configuration files and derives rules for building a language model from the given dataset. The resulting language model can be used to verify new configuration files and detect errors in them. Our proposed framework is highly modular, does not rely on the system source code, and can be applied to any new configuration file type with minimal user input. Our tool, named ConfigC, relies on an abstract representation of language rules to allow for this modularity. ConfigC supports learning of various rules, such as orderings, value relations, type errors, or user defined rules by using a probabilistic type inference strategy and defining a small interface for the rule type.

1 Introduction and System Overview

Configuration errors are one of the most important root-causes of modern software system failures [8, 9]. In practice, misconfiguration problems may result in security vulnerabilities, application crashes, severe disruptions in software functionality, and incorrect program executions [8, 10, 11]. Although several tools have been proposed to automate configuration error diagnosis after failures occur [3,5–7], these tools rely on a manual approach to understand and detect the failure symptoms. The main reasons for this are: 1) entries in configuration files are untyped assignments, 2) there is no explicit structure policy for the entries in configuration files, and 3) there are surprisingly few rules specifying the entries' constraints.

We propose an approach to the verification of configuration files which is based on learning rules about the language model for configuration files.

Figure 1 describes an overview of our system. We start with the assumption that we are given a number of correct configuration files belonging to the same category (for instance, MySQL or Apache). Such files follow similar patterns, which we exploit in a learning algorithm to build rules that describe a language model for the files. Since the "language" of configuration types is untyped and unstructured, we first parse the files and translate them into a more structured, intermediary representation. When running type inference on a configuration file, the type of a variable cannot always be fully determined from a single value. We address this problem by introducing so called *probabalistic types*. Rather than giving a variable a single type, we assign several types with their probability distributions. We can then use these more structured files as a training set to learn the rules. The learning algorithm is template-based to be easily extensible. We provide an initial set of templates and the learner learns some concrete instances

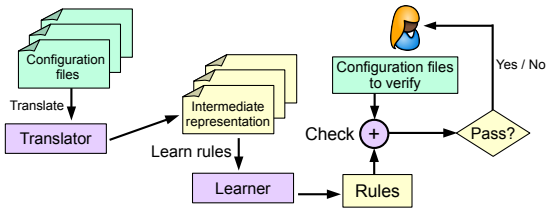

Fig. 1: ConfigC's workflow. The green boxes represent configuration files, including both correct general configuration files and users' input configuration files. The purple boxes are the components within ConfigC. The yellow boxes are results generated by ConfigC's components.

from the training set. These rules are used for detecting errors violating the learned constraints in the files given by the user.

As an illustration of a simple rule that we can learn, consider a template $X_1 \leq X_2$, where X_1 and X_2 are integer variables. The learner might derive the rule stating that `mysql.max_persistent` $\leq$ `max_connections`. There is a classification and taxonomy of configuration errors in the existing work on automated configuration troubleshooting [1, 9]. We provide templates for every class that ConfigC can handle: we consider integer constraints, ordering constraints, typing constraints, and constraints about correlated entries (such as "if X is present, Y has to appear as well"). Unfortunately, we cannot handle the class of errors that rely on the analysis of the whole operating system. Our language-based approach can only learn on sets of text files, not the system environment.

From a practical perspective ConfigC introduces no additional burden to the users: they can simply use ConfigC to check for errors in their configuration files. However, they can also easily extend the framework themselves. The system is designed to be highly modular. If there is a class of rules that ConfigC is not currently learning, the user can develop their own templates and learners for that class. The new learner can be added to ConfigC and this way it can check additionally a new set of errors.

Finally, from a systems perspective this is the first approach that *proactively* checks the correctness of configuration files. All previous work [3, 5–8, 10, 11] tries to identify the problem after the failure occured. Our approach isolates potential errors before the system failure occurs, e.g. before the installation. We can also see ConfigC as a tool that can run in conjunction with existing tools. Pre-analyzed configuration files are already free from language-based errors, and this way the workloads of post-failure forensics at the runtime is significantly reduced, thus making these tools truly practical.

To summarize, this tool paper makes the following contributions:

1. We designed and implemented a tool, ConfigC, that can learn a language model from an example set of correct configuration files, and we use the model to verify new configuration files.
2. We use probabilistic types to assign a confidence distribution over a set of types to a value.
3. In ConfigC we define a interface for describing a verification attribute in a learning context, making it easy to add new rules to the system.

2 Motivating Examples

When writing configuration files, users usually take already existing files and modify the files, with little knowledge of the system. The non-expert user can then easily introduce in errors. Even worse, the original file may already corrupted and the errors are propagated further. Below we show some real worlds examples of the errors commonly found in configuration files. All these examples are extracted from real-world reports The deep, domain specific knowledge needed to identify these error manually is strong motivation for a tool such as ConfigC.

Example 1: Ordering Errors. When configuring PHP to run with the Apache HTTP Server, the user writes, among others, the following lines:

```
extension = mysql.so
...
extension = recode.so
```

This file caused the Apache server to fail to start due to a segmentation fault error. When using PHP in Apache, the extension "mysql.so" depends on "recode.so" and the relative ordering of two of them is crucial. ConfigC would inform the user that "recode.so" should appear before "mysql.so", and return the error:

```
ORDERING ERROR: Expected "extension"recode.so"
BEFORE "extension""mysql.so"
```

Example 2: Entry Missing Errors. If the user wants to use OpenLDAP to enable her directory access protocol, she needs to use the password policy overlay. This is usually done through the following entries in the OpenLDAP configuration file:

```
include schema/ppolicy.schema
overlay ppolicy
```

When using the password policy overlay in OpenLDAP, we have to first include the related schema. Leaving out the "include" statement will cause the failure of this LDAP. Running ConfigC on such a misconfiguration file would return:

```
MISSING KEYWORD ERROR: Expected "overlay""ppolicy"
in the same file as: "include""schema/ppolicy.schema"
```

Example 3: Type Errors. If the user tries to install MySQL, she first needs to initiate the path for the log information generated by MySQL. A user may put the following code in the MySQL configuration file:

```
general_log = /var/log/mysql/mysql.log
```

However, the entry "general_log" should be an integer, not a string. In MySQL, there is another entry named "general_log_file" which is used to specify the log path. After ConfigC analyzes this configuration file, it correctly identifies the error:

```
TYPE ERROR: Expected a Int with P=1.0 for "general_log[mysqld]"
```

Example 4: Value Correlation Errors. When configuring PHP on MySQL, the user may write the following lines of entries in both the PHP and MySQL configuration files:

```
mysql's config
max_connections = 300
...
php's config
mysql.max_persistent = 400
```

This could cause MySQL to abort with the error information: “too many connections”. In this case, the “mysql.max_persistent” in PHP should be no larger than the “max_connections” in MySQL configuration file. Another rule we have implemented is learning inequality relations between integers. Running ConfigC on this combined configuration file would return: `INTEGER RELATION ERROR:`
`Expected "max_connections">="mysql.max_persistent"`

3 Learning the Rules

To learn rules, we first translate to the intermediate representation where each line of a configuration file is reduced to a keyword-value pair (k, v). Parsing is language dependent and users may provide extra help to the translator for their specific language, such as specifying a comment character. We must assign types to the keywords to guide the learning modules. With typed keyword-value pairs, we can run each learning module independent of each other. We learn a set of rules over a every file, then merge the rules of all files.

Introducing the types. Based only on a single example value of v we cannot fully determine the type of k. Consider for instance the following example:

```
foo = 300
bar = 300.txt
```

Most likely `foo` is an integer and we learn an equality rule, but it could also be a string. In this case we want to learn the rule $\texttt{foo} \in \mathsf{substrings}(\texttt{bar})$. We therefore assign a distribution of types to a value, an idea closely related to existentially quantified types [4]. We introduce *probabilistic types* to address this issue.

Let $\mathcal{T}$ be a set of basic types. In ConfigC set $\mathcal{T}$ contains strings, integers, file paths, sizes and IP addresses. A probabilistic type built from $\mathcal{T}$ is a list of pairs $[(\tau_1, p_1), \ldots, (\tau_n, p_n)]$ such that $\tau_i \in \mathcal{T}$, $0 \leq p_i \leq 1$ and $\Sigma p_i = 1$. These probabilities are updated each time a new example value for a keyword is encountered.

When a value has a probabilistic type, we generate rules for all its types. This means that by assigning `foo` a probabilistic type (e.g. $(\texttt{foo}, 300, [(\mathit{Int}, 90\%), (\mathit{String}, 10\%)])$ we now generate rules for both strings and integers. Once the type inference can uniquely determine the type, the probability of all other types is set to zero, and the associated rules are withdrawn.

Note that typing is also a system module than can be easily extended to support more types. In that case the user will need to provide rules for type inference and probability distributions for values where type inference is ambiguous.

Rule Learning. With every type we associate a set of templates, specific to this type. Once the input files are fully type-annotated, we generate rules that are instances of these templates. We always learn the largest set of rules that all correct configuration files satisfy. This way ConfigC can guarantee that, over the set of rules we consider, there will be no false negatives that could have been caught with the given learning set. The only case of a false negative can be when there was no evidence of such a rule in the learning set - we cannot generate rules from nothing.

4 Implementation and Evaluation

ConfigC is implemented in Haskell and takes full advantage of its polymorphism to make the system more modular. In particular, rules are represented as a type, where the type must support a particular interface (called a typeclass in Haskell) to be compatible with our system. By using language extensions (FlexibleInstances and MultiParamTypeClasses), this typeclass can be made polymorphic over the data structure as denoted by `Foldable t =>`. The user can then choose a data structure that is most natural to the rule they are implementing. For example, in our implementation, Missing Entries were easier to manipulate in lists, while Type Errors fit more naturally into a hashmap. This typeclass defines the three functions that each set of rules must implement to work with our system. Typeclasses and other features of Haskell means that our system consists of only 267 lines of code, with another 233 for the rule modules. With an average size of 58 lines of code for each rule module, this is evidence of how simple it is to extend ConfigC with new rules.

```
class Foldable t => RuleSet t a where
  learn :: IRConfigFile -> t a
  merge :: t a -> t a -> t a
  check :: t a -> IRConfigFile -> Error
```

Since we learn a set of rules on each file in isolation from the other, we have a pleasingly parallel situation. Haskell allows us to easily take advantage of this by using the parallel mapping library [2], both for translation to the intermediate representation, and for learning the rules on each file. The merge stage could also easily be parallelized, using a divide and conquer approach, but ConfigC runs fast enough over our learning set (28 files, 961 lines of code) that this has not been necessary.

The integer relation rule has an unusual implementation that makes use of function as first-class objects in Haskell. Rather than associated keywords with SMT formula, we directly associate them with a function of type (`Int->Int->Bool`). Since we need to compare rules over equality, we must then have a way to compare functions. This limits the types of functions we use to `(==)`, `(>=)`, `(<=)`. Although this is sufficient for most use cases, more fine-grained relations could be encoded with SMT formulas then passed to a solver.

4.1 Evaluation

To evaluate our tool, we take a subset of 20 benchmarks from an existsing dataset of configuration errors [1, 9] which are supported by our tool. Table 1 contains an evaluation summary. We do not report the running times, since they are negligible: even when running in the interpreter mode, files are analyzed instantaneously. We spent approximately 30 second on learning the rules. When we run the compiled version, we need for lerning and verification combined less than 5 seconds. Our focus is usefulness of the tool: its ability to detect configuration errors and the number of false positives. For every benchmark class we took five examples. The middle column represents the number of detected errors, while the right column represents the number of returned false positives per each benchmark.

Table 1: Benchmarks for misconfiguration detection

Error Type	Passing Tests	False Positives
Missing Entry	5/5	1, 0, 0, 0, 1
Type Error	5/5	0, 0, 0, 0, 0
Keyword Ordering	5/5	0, 1, 1, 0, 10
Value Relations	4/5	0, 1, 0, 0, 0

A benchmark passes a test if it reports an error on the source of the misconfiguration (it is not a false negative). We call false positives any reported error that was unrelated to the value of interest. It is worth noting that this is in fact a conservative estimate. Since these benchmarks are taken from online forums, there is no guarantee the files contain only a single error. Indeed, on some benchmarks, ConfigC found errors in the file that were similar to rules broken by other benchmarks.

We fail one benchmark in Value Relations because we do not yet support relations between file sizes of different units (Mb to Kb). In one Keyword Ordering benchmark, ConfigC reports a type error on the value of interest instead of a ordering error. This is a result of our context embedding in the translation to the intermediate representation - reordering the value puts it in a new context where the type is now also incorrect.

All but one false positive reports were integer relations. They are the result of overfitting on rules. ConfigC can learn overapproximating rules when the learning set does not show the full spectrum of possible values. Since integer relations have a larger space of relation than ordering relations for instance, ConfigC needs a larger learning set in order to eliminate false positives.

The false positive for the Value Relation was a Missing Entry error. This is a result of the fact that we cannot learn rules that are disjunctions. In this case, no socket is provided to [mysqld], failing a rule we had learned over the dataset. In fact, this is not a misconfiguration because a socket only needs to be provided to one (or both) [mysqld] or [wampsqld]. We reported an error since none of the files in the learning set had no socket associated with [mysqld]. In fact, since we do not support disjunctive rules, we could not have even learned such a rule - though in practice these seem to be uncommon.

5 Conclusions

In this paper, we introduce ConfigC, a highly-modular framework that allows verification of configuration files, even without a language model of the file. New verification properties require only a small amount of code and are not language specific, all indicating that ConfigC could be widely adopted by system administrators. Such a verification tool that scales in both performance and expressivity can revolutionize configuration file checking, reducing the cost of system maintaince and failure dramatically.

References

1. Misconfiguration dataset, https://github.com/tianyin/configuration_datasets
2. parallel-3.2.1.0: Parallel programming library, https://hackage.haskell.org/package/parallel-3.2.1.0/docs/Control-Parallel-Strategies.html
3. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Oct 2010)
4. Launchbury, J., Jones, S.L.P.: Lazy functional state threads. In: Programming Language Design and Implementation (PLDI). pp. 24–35. ACM Press (1993)
5. Su, Y., Attariyan, M., Flinn, J.: AutoBash: Improving configuration management with operating systems. In: 21st ACM Symposium on Operating Systems Principles (SOSP) (Oct 2007)
6. Wang, H.J., Platt, J.C., Chen, Y., Zhang, R., Wang, Y.: Automatic misconfiguration troubleshooting with PeerPressure. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
7. Whitaker, A., Cox, R.S., Gribble, S.D.: Configuration debugging as search: Finding the needle in the haystack. In: 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (Dec 2004)
8. Xu, T., Zhou, Y.: Systems approaches to tackling configuration errors: A survey. ACM Comput. Surv. 47(4), 70 (2015)
9. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In: 23rd ACM Symposium on Operating Systems Principles (SOSP) (Oct 2011)
10. Yuan, D., Xie, Y., Panigrahy, R., Yang, J., Verbowski, C., Kumar, A.: Context-based online configuration-error detection. In: USENIX Annual Technical Conference (USENIX ATC) (Jun 2011)
11. Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., Zhou, Y.: Encore: Exploiting system environment and correlation information for misconfiguration detection. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (Mar 2014)