Version Space Learning for Verification on Temporal Differentials

Mark Santolucito
Yale University
New Haven, CT, USA
mark.santolucito@yale.edu

ABSTRACT

Configuration files provide users with the ability to quickly alter the behavior of their software system. Ensuring that a configuration file does not induce errors in the software is a complex verification issue. The types of errors can be easy to measure, such as an initialization failure of system boot, or more insidious such as performance degrading over time under heavy network loads. In order to warn a user of potential configuration errors ahead of time, we propose using version space learning specifications for configuration languages. We frame an existing tool, ConfigC, in terms of version space learning. We extend that algorithm to leverage the temporal structuring available in training sets scraped from versioning control systems. We plan to evaluate our system on a case study using TravisCI configuration files collected from Github.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

Configuration Files, Verification, Machine Learning

ACM Reference format:
Mark Santolucito. 2017. Version Space Learning for Verification on Temporal Differentials. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17-DOC),* 5 pages.
https://doi.org/10.1145/3092703.3098238

1 INTRODUCTION

Configuration files provide easy access to the most critical parameters of software systems in order to quickly tune the behavior of a system. This expressive power in a single file creates a large surface for potential errors that can degrade performance or take down systems entirely. The verification task for configuration files cannot be addressed with traditional software analysis techniques because they take a less structured form than a program. As opposed to programs, configuration files rarely have formal specifications of a correctness, the source code of the underlying system may not be available, and the configuration may depend on the external environment such as the network or available hardware. To overcome the unique challenges in this domain, we propose a method for automated configuration file verification using a data set of labeled examples of correct and incorrect configurations.

In our previous work [11], we built the tool ConfigC to verify configuration files from a set of correct examples. ConfigC takes as a training set a set of correct configuration files and builds a set of rules describing a language model. This model is then used to check if a new configuration file adheres to those rules. If a file is incorrect, the tool identifies the location of error and reports what is incorrect. For example, when analysing MySQL configurations, ConfigC might output `Ordering Error: Expected "extension mysql.so"` before `"extension recode.so"`. While ConfigC's results are promising, the main weakness is that the learning can only use correct configuration files. Additionally, there is a relatively high false positive rate (marking correct files as incorrect).

Here we propose to extend that work, which only works on correct examples, to work on both correct and incorrect examples by building an SMT formula to find a classification model. By learning over both correct and incorrect examples, our hypothesis is that we will be able to decrease the high false positive rate observed in ConfigC, making the tool more useful to end users. In order to extend ConfigC, we first provide a new description of ConfigC in terms of version space learning [10]. This description allows us to understand how the error reports are produced in ConfigC. Any extension to ConfigC must maintain the *legibility* of learning - the tool must not only classify configuration files as correct or incorrect, but also identify the contributing factors (misconfigurations) to that classification.

Legibility is a difficult property to achieve in machine learning and an active area of research [8]. Although machine learning has been used in various software analysis techniques, such as programming-by-example [7] and invariant synthesis [4], this requires careful selection of the algorithms. Many popular machine learning algorithms, such as neural nets and n-gram models, notoriously lack legibility and so are not well-suited to to provide simple justifications for their classification results. The lack of justification for the results limits the applicability to software analysis, where providing feedback to a user is critical. For example, in the case of error detection, the system should not only report which files have errors, but also locate the errors so a user may fix them. By using non-probabilistic version space learning, we expect to have a highly level of legibility in our learning results.

Additionally, probabilistic approaches to machine learning cannot be guaranteed to be complete, that is they are not guaranteed to always find an error if one exists. We use version space learning to address the need for a formal completeness guarantee in an automated model generation for verification. Version space learning is a technique for logical constraint classification. As a logical

ISSTA'17-DOC, July 2017, Santa Barbara, CA, USA
2017. ACM ISBN 978-1-4503-5076-1/17/07...$15.00
https://doi.org/10.1145/3092703.3098238

constraint system, version space learning not only provides justification of its classification results, but is has the foundation to build a completeness proof without the need for probabilistic thresholds.

Additionally, our extension takes advantage of a commonly available, but underutilized structure of training sets for software analysis. While the previous training sets for ConfigC were an unsorted sampling of configuration files, typically we can obtain training sets that have a structure based on version history. Because most code does not exist in isolation, but changes over time with development, any training set of code from a version control system, like Github, has this rich temporal structure. This structure is a partial order over time, and can be used by our proposed algorithm for more effective learning.

To test this approach in practice, we plan to implement our algorithm to check for TravisCI[1] configuration errors. TravisCI is a continuous integration tool connected to Github that allows programmers to automatically run their test suite on every code update (commit). A user adds a configuration file to the repository that enables TravisCI and specifies build conditions, such as which compiler to use, which dependencies are required, and a set of benchmarks to test. This gives the developer confidence the code can always be automatically built correctly on a fresh machine. The algorithm will utilize the temporal ordering on code in Github histories to allow us to detect potential build errors without actually building, saving valuable programmer and server time.

A recent usage study [1] of TravisCI found that 15-20% of failed TravisCI builds are due to "errors" - which is the TravisCI name used to mean the configuration file was malformed and the software could not even be built. Using the data from [1], we can also learn that since the start of 2014, approximately 88,000 hours of server time was used on TravisCI projects that resulted in an error status. This number not only represents lost server time, but also lost developer time, as programmers must wait to verify that their work does not break the build. If these malformed projects could be quickly statically checked on the client side, both TravisCI and its users could benefit.

While ConfigC was applied to MySQL configurations, and we propose our next application as TravisCI, this approach can be used for any configuration language. TravisCI is just one example of an easily accessible set of labeled configuration files with a partial order.

Our main contributions are as follow:

(1) We give a description of our tool, ConfigC, in the context of version space learning.
(2) We propose a new algorithm to handle both incorrect and correct training data, as well as the internal structure of a training set.
(3) We propose a real-world application of this approach, and outline the steps needed for an implementation.

2 VERSION SPACE FOR VERIFICATION

Version space learning builds a logical constraint model for binary classification, which we use to test a configuration file for membership in the set of all correct files [10]. Traditional version space learning builds a model that tests membership using a series of disjunctions from a set of predefined hypotheses. In ConfigC, we instead restrict the model to use a series of conjunctions. In other words, rather than describing a correct file by allowable traits, we describe the required traits. This way, ConfigC can not only flag misconfigurations, but also give the points of failure for non-membership. We now show in detail how this process works.

Our definition of a configuration file, C, is a file that can be transformed to an ordered list of $(keyword, value)$ pairs, called $Line$s. ConfigC builds a model for a single file, $M(C)$, that is the set of all possible relations between the lines in the file. A relation is described by user provided templates. A template is a function taking some number of lines and determining a relation on the keywords of the lines ($Line^n \rightarrow Rel_{k1,..,kn}$). As an example, ConfigC might derive the relation `extension mysql.so` comes before `extension recode.so` by using the ordering template. ConfigC includes default templates for ordering, integer relations ($<, =, >$), type errors, and missing entries (two keywords must appear together irrespective of order).

It is assumed if a file is correct, all relations in that file are in the set of necessary relations, Nec, for any other file to be correct ($Correct(C) \Rightarrow \forall r \in M(C), r \in Nec$). In this way, the initial model is built by creating the strongest conditions for a correct file, called the *specific boundary* in version space learning. This model is then iteratively relaxed as more examples are seen, a process called *candidate elimination*. To relax the model, two sets of relations from two files are merged into a single consistent set.

When the system is run on a new file, one of the relations from the templates will not be satisfied. To provide an error message, the system only needs to report which relation is not satisfied.

Since we maintain the strongest condition for correctness, this approach will identify many correct files as incorrect, which we call a high false positive rate. This is a problem when a user is then asked to manually review many files for errors, when the files are in fact correct. However, the benefit to this approach is that, given the coverage of the training set, every misconfiguration will be detected.

2.1 Completeness

We demonstrate that ConfigC is complete but unsound. That is, we will always detect a misconfiguration, but may also report correct files as misconfigurations. We first define the set of incorrect files based on ground truth Inc_{gt}, and the set of files which are predicted by the model to be incorrect, Inc_{prd}. To define Inc_{gt}, we will introduce the set of relations that are necessary and breaking based on ground truth, Nec_{gt} and Br_{gt} respectively. The definition of Inc_{prd} follows from the description of the algorithm.

$$Inc_{gt} = \{\forall C, \exists r \in Nec_{gt}, r \notin M(C) \vee \exists r \in Br_{gt}, r \in M(C)\} \tag{1}$$

$$Inc_{prd} = \{\forall C, \exists r \in Nec, r \notin M(C)\} \tag{2}$$

Our definition of sound and complete are as follows:

$$\forall C \in Inc_{gt}, C \in Inc_{prd} \text{ Complete} \tag{3}$$

$$\nexists C \in Inc_{prd}, C \notin Inc_{gt} \text{ Sound} \tag{4}$$

Completeness follows from showing that $Inc_{gt} \subseteq Inc_{prd}$. The algorithm is complete if, for any file which breaks a relation for which we have a template (a misconfiguration) and have seen a correct example, the system will mark that file as incorrect. Since

[1]http://www.travis-ci.com

the model always maintains the strongest conditions for correctness for any file, the algorithm is clearly complete. Completeness corresponds to no false negatives. However, this algorithm is unsound, as some correct file may be marked as incorrect, i.e. ConfigC returns a false positive. We have a high false positive rate since the model is an over-approximation of the true model.

3 LEARNING FROM TEMPORAL PROPERTIES

ConfigC can provide justifications, but has a high false positive rate. However, ConfigC only analyzes correct configuration files, and does not consider any ordering between these correct files. We present a new algorithm to decrease the false positive rate in ConfigC by learning on both correct and incorrect examples, as well as temporal structure of these examples. This extension will build a logical formula representing the entire history of examples, and use a SMT solver to find a classification model.

While in ConfigC, configuration files were manually labeled in isolation from their system by an expert, TravisCI configurations are automatically labeled with respect to their containing repository. Because the correctness of TravisCI configuration files is dependent on the code it builds, we must model more than just the configuration file, C. We introduce a program summary P_t, which is a combination of C as well as the information relevant to the learning process from the repository. The subscript on P_t is a timestamp tag based on the ordered git commit history. In the case of TravisCI, this include the `.travis.yml` file, as well as code features that may affect build status, such as programming language and a list of imported libraries. The summary must be *sufficiently detailed*, that is, it must contain every piece of information that might lead to a build error.

However, most versioning systems do not limit version history to a single linear timeline. For example, Git features the ability to *branch* and create temporally simultaneous commit chains. To handle the start of a branch, we add a superscript to indicate the branch (x and y here). To handle the merge of two branches P_t^x and $P_{t'}^y$, step to P_{t+1}^x, where x is the mainline branch. We then say that $P_{t'}^y$ has no successor commit $P_{t'+1}^y$. Note the ability to handle branching implies this method is suitable to any training set which has a partial order.

From this summary we can then build a model $M(P_t)$, as in ConfigC, which is the full set of possible relations derivable from the program summary. Again, the user must provide templates for the learning process.

We will denote the build status returned from TravisCI when run on P_t with $S(P_t)$. Any files with a $Pass$ build status are correct, and any files with an Err build status are incorrect. In this application, we consider all non-erroring build status to be passing, denoted $Pass$ and otherwise Err. For brevity, we denote sequences of build statuses with the following notation:

$$S(P_t) = Pass \wedge S(P_{t+1}) = Err \Rightarrow S(P_{t,t+1}) = PE$$

In contrast with ConfigC, we now consider both incorrect and correct examples and so must introduce the *general boundary*. The general boundary is the dual of the specific boundary, and is the most relaxed requirement for a positive classification. We denoted specific boundary as the set of necessary relations Nec, and now denote the general boundary as the set of breaking relations Br. With this notation, we can formally express the requirement that the program summary is sufficiently detailed.

$$\forall S(P_t) = Err, \exists r \in M(P_t), r \in Br \tag{5}$$

From the above we know that if a build is erroring, then there must exist at least one error. We can also know that if a build is passing, then there must not exist any errors. That is, the model of a passing commit must not contain any rules which are breaking. Note we are not, however, guaranteed that any rules from a passing commit are necessary.

$$S(P_t) = Err \Rightarrow \exists r \in M(P_t), r \in Br \tag{6}$$

$$S(P_t) = Pass \Rightarrow \forall r \in M(P_t), r \notin Br \tag{7}$$

Additionally, when we commit a break (PE), we can localize the error to one of the relations that changed. Either we removed something that was necessary, or added something that was breaking. Note that this is an inclusive disjunction, since a erroring commit can break multiple things at once. When comparing multiple versions of configurations, it is important that we look at the difference between the models $M(P)$ and not just P. In the latter case, we would treat individual lines as error sources, when in fact we want to detect the relations between configuration settings. Expressed formally, where $\setminus$ is the set difference, we learn the following from a breaking commit:

$$\begin{aligned} S(P_{t,t+1}) = PE \Rightarrow& \\ &\exists r \in (M(P_t) \setminus M(P_{t+1})), r \in Nec \\ &\vee \exists r \in (M(P_{t+1}) \setminus M(P_t)), r \in Br \end{aligned} \tag{8}$$

In fact, to make full use of the partial order, we can generalize Eq 8 to $P_{t,t+n}$. For example, the pattern $Pass_0, Err_1, Pass_2, Err_3$ yields a PE pattern on $P_{0,1}$, $P_{2,3}$, and $P_{0,3}$. More complex chains could be built in the case of branching.

We then can combine Eq 6, 7, and 8 with conjunctions and ask a SMT solver for a model satisfying the formula. The resulting model will be the sets Nec, and Br, which can be used to check new configuration files. Since we used a similar model to ConfigC, we will still be able to provide justifications for the classification results.

While existential set operations can be expensive on large sets for an SMT solver, in our application this is not the case. Thanks to the practice of making incremental commits when using source control, these sets will be small and the SMT will be fairly cheap. However, for efficiency we must choose a maximum n such that $M(P_t) \setminus M(P_{t+n})$ is manageably small. The definition of small and selection of maximum n remains to be experimentally determined.

4 PROPERTIES OF TEMPORAL LEARNING

We have made two assumptions in Eq. 5; first that the summary P_t sufficiently detailed, i.e. contains every piece of information that might lead to a build error, and second that the model $M(P_t)$ will learn all relations that might lead to a build error, i.e. has a template for all types of errors. While these are strong assumptions, our algorithm is able to detect cases where these conditions are not met. If we cannot find a solution for Eq. 8, it means either P_t or $M(P_t)$ has been underspecified and is insufficiently detailed, and the definitions must be expanded accordingly.

In addition to the sufficient detail of P_t and $M(P_t)$, we must pick a trusted base. Since TravisCI may not always be backwards compatible, it is possible that between versions of TravisCI, two identical program summaries may have different build statuses. Since version space learning is (generally) intolerant of noise, we require that $P_t = P_{t'} \Rightarrow S(P_t) = S(P_{t'})$. If we allow for noisy data, for example in the case that it is not possible to identify a trusted base, this predicate will no longer hold. This situation would require a noise tolerant version space learning [5], though this may degrade the completeness guarantees.

Picking the right assumptions upon which to build a completeness and soundness proof remain future work, either with a trusted base, or in the noisy context. Choosing the right assumption and building these proofs are a valuable direction, as it will allow us to identify which parts of the model and training set are most important to ensure coverage of the system.

5 RELATED WORK

Language-support misconfiguration checking. One approach to statically checking configuration files is to avoid the difficult and unstructured nature of configuration files in the first place. To that end efforts have introduced special configuration languages that overcome the fundamental deficiencies in the existing untyped or low-level configuration languages. For example, network administrators often produce configuration errors in their routing configuration files. PRESTO [3] introduces configlets as a template language to automates the generation of device-native configurations. Other work uses Datalog to reason about routing protocols in a declarative fashion [9]. COOLAID [2] is a domain specific language to describe knowledge about network devices and services. In contrast to this approach, our main purpose is to automate configuration verification of existing configuration languages, rather than proposing new, more powerful languages. This is largely motivated by the unwieldy amount of existing configuration files in industry production.

Another language based approach is used by ConfValley, a specification language for administrators to provide constraints on their systems' configurations [6]. This approach requires that administrators manually write specifications - an error-prone and difficult process. In contrast, by using a learning approach, our work does not require users to manually write any specifications.

Learning. Using machine learning to tackle misconfiguration detection has also been explored [12–14]. As an example, EnCore [14] uses a rule learning approach, augmented with user provided templates, to improve the accuracy of their learning results. The learning process is guided by a set of predefined rule templates that enforce learning to focus on patterns of interest and allow the problem to scale over larger training sets. EnCore uses the templates to filters out configuration entries that are unlikely to cause errors to reduce false positives. The templates are also used to directly encode system environment information that would not be contained in the training set for their machine learning. Compared with EnCore, our approach does not rely on templates - everything is automatically generated. Furthermore, none of the above approaches use the temporal ordering on the training set inside the learning process.

White-box. In our approach we assume a "black-box" model where we cannot access the system source code. This is as opposed to a "white-box" model, where the source code is available and can leveraged in the configuration analysis. Although this is a strong assumption, PCheck [12] explores misconfiguration detection by adding configuration checking code directly in the system source code. The system is then emulating with various potential configurations of the system. One drawback to this approach is that for some systems (e.g. ZooKeeper) whose behavior is hard to emulate, PCheck cannot automatically generate the corresponding checking code. Due to the emulation based testing strategy, PCheck's scope is limited to reliability problems caused by misconfiguration parameters. In contrast, our work is a "black-box" approach and only requires a training set of configuration files to learn rules. By using the version space learning strategy of examples, we are also able to detect general misconfiguration issues that are outside the scope of emulation testing (e.g. memory or thread usage settings), including performance, security, availability and reliability.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1302327

REFERENCES

[1] Zaidman A. Beller M, Gousios G. 2016. Oops, my tests broke the build: An analysis of Travis CI builds with GitHub. PREPRINT. (2016). https://doi.org/10.7287/peerj.preprints.1984v1

[2] Xu Chen, Yun Mao, Zhuoqing Morley Mao, and Jacobus E. van der Merwe. 2010. Declarative configuration management for complex and dynamic networks. In *ACM CoNEXT (CoNEXT)*.

[3] William Enck, Patrick Drew McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert G. Greenberg, Sanjay G. Rao, and William Aiello. 2007. Configuration Management at Massive Scale: System Design and Experience. In *USENIX Annual Technical Conference (USENIX ATC)*.

[4] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *CAV*. 69–87.

[5] Tzung-Pai Hong and Shian-Shyong Tsang. 1997. A generalized version space learning algorithm for noisy and uncertain data. *IEEE Transactions on Knowledge and Data Engineering* 9, 2 (1997), 336–340.

[6] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: A systematic configuration validation framework for cloud services. In *10th European Conference on Computer Systems (EuroSys)*.

[7] Tessa A Lau, Pedro M Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration.. In *ICML*. 527–534.

[8] Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2016. Rationalizing neural predictions. *arXiv preprint arXiv:1606.04155* (2016).

[9] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative routing: Extensible routing with declarative queries. In *ACM SIGCOMM (SIGCOMM)*.

[10] Tom M. Mitchell. 1982. Generalization as search. *Artificial Intelligence* 18, 2 (1982), 203 – 226. https://doi.org/10.1016/0004-3702(82)90040-6

[11] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic Automated Language Learning for Configuration Files. In *CAV*. 80–87. https://doi.org/10.1007/978-3-319-41540-6_5

[12] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[13] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based online configuration-error detection. In *USENIX Annual Technical Conference (USENIX ATC)*.

[14] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.