Version Space Learning for Verification on Temporal Differentials

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

Ruzica Piskac
Yale University
New Haven, CT, USA
ruzica.piskac@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 avaible 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.

KEYWORDS

Configuration Files, Verification

ACM Reference format:
Mark Santolucito and Ruzica Piskac. 2017. Version Space Learning for Verification on Temporal Differentials. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, January 01–03, 2017 (ISSTA'17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Configuration files provide easy access to the most critical parameters of software systems in order to quickly tune the behavior 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. There are rarely formal specifications of a correct configuration, 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 address this issue, we propose a method for automated configuration file verification using a data set of labeled examples of correct and incorrect configurations.

In previous work, we used version space learning for configuration file verification. We extend that work which only works on correct examples, to work on temporally structured correct and incorrect examples by building an SMT formula to find a classification model. As a demonstration of this algorithm, we plan to implement it to automatically verify TravisCI configurations. The algorithm will utilize Github histories to allow us to detect potential build errors without actually building, saving valuable programmer and server time.

ISSTA'17, New York, NY, USA
2017. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Machine learning has been used in various software analysis techniques, such as programming-by-example (Lau et al. 2000), invariant synthesis (Garg et al. 2014), and error detection (Santolucito et al. 2016). However, many popular machine learning algorithms, such as neural nets and n-gram models, are not designed to provide simple justifications for their classification results. While effective in practice, the lack of justification for the results limits the applicability to software analysis, where justification is often critical. For example, in the case of error detection, the system should not only report which files have errors, but also locate the errors. This justification issue is also called *legibility* of machine learning.

Additionally, these probabilistic approaches to machine learning cannot be guaranteed to be complete, that is they will always find the error. 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 (Mitchell 1982). As a logical constraint system, version space learning can provide the justification that is difficult to obtain from other methods, but its main weakness is that it cannot handle noisy data.

In our previous work (Santolucito et al. 2016) we built a prototype to verify configuration files from a set of correct examples. The tool, 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, ConfigC might output `Ordering Error: Expected "extension mysql.so"` before `"extension recode.so"`. While this prototype'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).

In order to extend ConfigC, in this paper we provide a new description of ConfigC in terms of version space learning. This description will allow us to understand how the justifications are produced in ConfigC. We can then extend ConfigC with a new algorithm to decrease the false positive rate, while maintaining the ability to provide justification.

We propose an extension to ConfigC that allows it to handle both correct and incorrect files in the training set. Additionally, we extend ConfigC with the ability to use structure within the training set. While the previous training set from ConfigC was a random sampling of configuration files, many other training sets have a structure based on version history. The algorithm we propose takes advantage of this commonly available, but underutilized structure of training sets for software analysis. 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 ensures the tool can always be automatically built correctly on a fresh machine.

A recent usage study (Beller M 2016) 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 (Beller M 2016), 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.

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 (Mitchell 1982). 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 $Lines$. 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 features the following default templates.

Template	Arity	Relation
ordering	2	Before | After | None
coarse relation	2	< | == | >
fine-grain relation	3	< | == | >
type	1	String | Int | Boolean | IP
missing entry	2	Required | Not

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.

```
n = {}
for (c in files):
  n1 = M(c)
  n = merge(n, n1)
```

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.

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.

ConfigC is complete but unsound. 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.

$$\begin{aligned} Inc_{gt} &= \{\forall C, \exists r \in Nec_{gt}, r \notin M(C) \\ &\qquad \vee \exists r \in Br_{gt}, r \in M(C)\} \quad (1) \\ Inc_{prd} &= \{\forall C, \exists r \in Nec, r \notin M(C)\} \quad (2) \end{aligned}$$

Our definition of sound and complete are as follows:

$$\forall C \in Inc_{gt}, C \in Inc_{prd} \text{ Complete} \quad (3)$$
$$\nexists C \in Inc_{prd}, C \notin Inc_{gt} \text{ Sound} \quad (4)$$

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 an correct example, the system will mark that file as incorrect. Since 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.

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

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 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 effect 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*, which allows to simultaneous commit chains. To handle the start of a branch, add a superscript to indicate the branch, and restart the counter on a branch. 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:

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

In contrast with ConfigC, we now consider both incorrect and correct 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 \ is the set difference, we learn the following from a breaking commit:

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

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. In fact, the above implication generalizes to $P_{t,t+n}$, but for efficiency we must require that $M(P_t) \setminus M(P_{t+n})$ is manageably small. The definition of small here remains to be experimentally determined.

4 LIMITATIONS

We have made two assumptions in Eq. 5; first that the summary P_t sufficently 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 templete for all types of errors. While these are the strong assumptions - our algorithm is able to detect cases where these assumptions are not met. If we cannot find a solution for Eq. 8, it means either P_t or $M(P_t)$ has been underspecified. It is then the user's responsibility to expand the definitions accordingly.

Mark: [Future work is showing completness of this approach. what assumption do we need? maybe make the assumption in configC completness more explicit?]

4.1 Trusted Base

In addition to the completeness of P_t and $M(P_t)$, we must pick a trusted base. Since it is possible for TravisCI to have bugs, 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 $P_t = P_{t'} \Rightarrow S(P_t) = S(P_{t'})$.

Mark: [In future work we can try using noise tolerant learning?]

5 RELATED WORK

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

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

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)*.

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)*.

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

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)*.

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.

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

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

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

Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving configuration management with operating systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)*.

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)*.

Tianyin Xu and Yuanyuan Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.* 47, 4 (2015), 70.

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)*.

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)*.