Software Configuration Diagnosis – A Survey of Existing Methods and Open Challenges

Artur Andrzejak[1] and **Gerhard Friedrich**[2] and **Franz Wotawa**[3]

Abstract. As software systems become more complex and feature-rich, configuration mechanisms are needed to adapt them to different execution environments and usage profiles. As a consequence, failures due to erroneous configuration settings are becoming more common, calling for effective mechanisms for diagnosis, repair, and prevention of such issues. In this paper, we survey approaches for diagnosing software configuration errors, methods for debugging these errors, and techniques for testing against such issues. In addition, we outline current challenges of isolating and fixing faults in configuration settings, including improving fault localization, handling the case of multi-stack systems, and configuration verification at runtime.

1 Introduction

Tackling software configuration errors is recognized as an important research problem which has been investigated by many groups from academia and industry, e.g., see [51]. In a recent study [52], the authors report empirical findings on the impact of configuration errors in practice. In particular, a study of over 500 real-world configuration issues revealed that this type of problems constituted the largest percentage (31%) of high-severity support requests. Moreover, a significant portion of these issues (16% to 47%) rendered systems fully unavailable or caused severe performance degradation. Also other studies [30] and incident reports [5] confirm that detecting and correcting configuration errors in software is of a great importance for practical applications.

In this paper, we focus on providing an overview of current research in the area of software configuration diagnosis comprising fault detection, fault localization, and correction. Besides discussing research articles dealing with software configure errors, we further discuss open issues and challenges that are worth being tackled in future research activities. While the excellent survey [51] has a broader scope and also includes aspects such as configuration-free/easy-to-configure systems, hardening against configuration errors, automating deployment and monitoring etc., we consider in this paper primarily diagnosis aspects. We also cover the most recent state-of-the-art work like diagnosing cross-stack configuration errors [32]. In summary, this survey attempts to offer a compact and focused introduction to this research area, thus serving as a good starting point for further contributions.

Although, there has been work also dealing with configurations and configuration errors for systems comprising hardware and software, we focus on methods and tools that have been developed within the area of software configuration. Dealing with software configuration only allows for extracting and straightforwardly using information from programs, which would be hardly obtained when considering hardware. As a consequence, there are many approaches that work exclusively in the software configuration domain. Nevertheless, there are also approaches that can be generalized to serve diagnosis of system configuration as well. Especially, when it comes to large software comprising million lines of source code and also to cases where source code is not available, approaches have to follow a more black-box oriented approach. This approach also enables diagnosis in case of hardware or systems in general where hard- and software is investigated.

In more detail, given a program, its configuration parameters (or settings), and an execution environment, a *software configuration error* comes forward when the parameters assume incorrect values. The configuration parameters might specify multiple aspects of system behavior, including adaptation to execution environment (paths, network settings, ..), functionality (enabled/disabled components, logging, ...), performance and resource policies (cache sizes, number of threads, ..), security settings, and others. Consequently, erroneous configuration settings can cause failures of multiple types: complete crashes, partially disabled functionality, performance issues, inappropriate resource usage, or security threads. A frequent scenario of a configuration error are parameter values which do not fit to the specific execution environment. For example, we specified a path to a working directory of the application but the user executing the program do not have write access to this directory, causing the program to crash (or at least to terminate with an exception).

In the context of this survey, we consider the *configuration error diagnosis problem* in its most general form: detecting the root causes, i.e. isolating the configuration parameters with inappropriate values, and providing means for repair in terms of identifying correct values or value ranges for these parameters (or adapting the execution environment). This definition implies that we do not target diagnosis of "traditional" software bugs, since we assume that a repair is possible without code changes. Note that it might be difficult to decide whether a failure should be attributed to a configuration problem or a software bug, and this challenge remains one of the open issues (see Section 3). For example, if a failure-triggering sequence of statements in a faulty program is executed only because of a certain parameter setting, the subsequent failure might appear to be caused by a configuration error.

We organize this paper as follows: We first discuss in Section 2 previous research works dealing with software configuration diagnosis. In the following Section 3 we present open research challenges that have not been tackled so far. We discuss threats to validity in

[1] Heidelberg University, Germany, email: artur.andrzejak@informatik.uni-heidelberg.de
[2] University Klagenfurt, Austria, email: Gerhard.Friedrich@aau.at
[3] TU Graz, Institute for Software Technology, Austria, email: wotawa@ist.tugraz.at

Sec. 4. Finally, we summarize the content and the findings of this paper (Section 5).

2 Previous Work on Software Configuration Diagnosis

In this section, we discuss research work that has been published in the area of software configuration diagnosis. We obtained the papers searching relevant digital libraries from IEEE and ACM. We further focussed on the most recent work in this area not older than 10 years. Hence, we do not claim the survey to comprise all papers in the context of software configuration errors (for a more comprehensive collection see [51]). However, the presented papers are intended to give an overview of the current research directions in software configuration diagnosis and methods and techniques used for this purpose.

In order to present the discussed papers in an accessible way, we classify the paper accordingly to the following categories: (i) diagnosing single-layer configuration errors, (ii) diagnosing cross-stack configuration errors, (iii) diagnosing using configuration knowledge, and (iv) other aspects of software configuration diagnosis. *Single-layer configuration errors* are errors found in one-component applications like MySQL, Hive, or Spark. Typically, such applications have one common configuration file/database and are developed as an integral project. *Cross-stack configuration errors* occur in multi-component applications or software stacks like LAMP (Linux, Apache Web Server, MySQL, PHP, Wordpress/Drupal), J2EE, or MEAN.

The rational behind these categories is the following. Most previous work is available for diagnosing single-layer configuration errors and this case offers an opportunity for an overview of existing diagnosis approaches. Diagnosis of cross-stack configuration errors pose additional challenges. In some cases, the source code of stack components might not be available, precluding usage of general program analysis techniques. More frequently, cross-stack configuration errors are frequently caused by a mismatch between the configuration settings within separate components [32, 33]. To diagnose such issues, knowledge about the interactions between the components is needed.

In case of the availability of formal knowledge about configurations, i.e., configuration rules or constraints, diagnosis can be performed using this knowledge. Such formal knowledge bases may be applicable for single-layer or cross-stack applications.

Finally, there are other aspects that cannot be assigned to one of the former categories, for example testing configurable systems or optimization of software based on configuration parameters.

2.1 Diagnosing Single-Layer Configuration Errors

Single-layer programs are typically written in a single programming language and often the source code is available. Hence, static and dynamic program analysis techniques can be applied to obtain a mapping from configuration options to code regions. This information can be exploited for localizing the root cause behind configuration errors. Consequently, a lot of approaches for diagnosis configuration errors in such programs have been proposed.

Linking configuration options and code regions. Approaches in this group attempt to find a correspondence between a configuration option and code regions impacted by this option. Frequently, such techniques exploit static [43] or dynamic program slicing [14]. In program slicing, one attempts to find the set of all code locations which might influence a target statement (so-called seed), or all code locations which might be influenced by a seed statement. Hence, there approaches are mainly applicable in the software configuration setting and may not be generalizable to deal with hardware configuration diagnosis.

ConfAnalyzer [29] builds a map from each program point to the options that might cause an error at that point by static data-flow analysis. For diagnosis, it treats a configuration option as the root cause if its value flows into the crashing point. The approach does not require from users to install or use additional tools, but it can use logs and stack traces to reduce the rate of false positives.

ConfDiagnoser [57, 56] uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior that are represented by predicates to configuration options. When these predicates indicate behavior deviating from the one known for correct profiles, ConfDiagnoser lists the relevant configuration options as suspects.

Work [58] presents a technique and a tool to troubleshoot configuration errors caused by software evolution. The approach uses dynamic profiling, execution trace comparison, and static analysis to link the undesired behavior to its root cause - a configuration option which needs to be changed in the new software version.

ConfDoctor [7] is an approach based on static analysis to diagnose configuration defects. It does not require users to execute an instrumented program or to reproduce errors, which is an essential advantage compared to previous approaches. The only run-time information required is the stack trace of a failure. An evaluation on JChord, Randoop, Hadoop, and Hbase shows that the approach could successfully diagnose 27 out of 29 errors, with 20 of them ranked first.

Authors of [25] propose a lightweight dynamic analysis technique that automatically discovers a program's interactions, i.e., logical formulae that give developers information about how a system's configuration option settings map to particular code coverage. It is evaluated on 29 programs spanning five languages and could find precise interactions based on a very small fraction of the number of possible configurations.

Data flow analysis. ConfAid [3] applies dynamic information flow analysis techniques to track tokens from specified "configuration sources" and analyze dependencies between the tokens and the error symptoms, pinpointing which tokens are root causes.

Sherlog [53] uses static analysis to infer control and data information in case of a failure. It analyses source code by exploiting information from run-time logs and computes what must or may have happened during the failed run. One deficiency of this tool is that it may require guidance from developers about which function should be symbolically executed.

Paper [17] introduces Lotrack, an extended static taint analysis approach and tool to automatically track configuration options. It derives a configuration map that explains for each code fragment under which configurations it may be executed.

Supervised learning approaches. Relatively few authors propose to use machine learning approaches based on supervised learning (i.e. mainly classification). This can be explained by the fact that it is difficult to obtain or generate training data with appropriate structure and in sufficient amount. Similarly to the challenges of mutation testing, if training samples are generated, faults injected in the configuration files might not trigger a failure or have unrealistic properties. Also, since a configuration file might contain hundreds of options, a training set is likely to containt only few faulty cases per option, giving rise to the unbalanced class problem.

Authors of [41] use machine learning to predict whether a configuration error is responsible for a failure and if yes, what is the category of the error. To obtain training data, faults are injected into configuration files and the resulting error category is manually labeled.

Work [38] exploits statistical decision tree analysis to determine possible misconfigurations in data center environments. The authors further improve the accuracy of this approach via a pattern modification method.

Replay-based techniques. One category of well-known tools [44, 37, 20] are the replay-based diagnosis techniques. They treat the system as a black box to automatically run the system with possible configurations values without damaging the rest of the system until fixing the misconfiguration. This class of techniques relies on having a working configuration. Otherwise, it can not be applied. Besides, they require users with more domain knowledge.

Signature-based approaches. Another family of tools mine a large amount of configuration data from different instances to infer rules about options and use these rules to identify software misconfigurations.

EnCore [55] and CODE [54] belong to this category of work. EnCore takes into account the interaction between the configuration settings and the executing environment, as well as the correlations between configuration entries. It learns configuration rules from a given set of sample configurations and pinpoints configuration anomalies based on these rules.

Analogously, some tools such as Strider [42] or PeerPressure [40] adopt statistical techniques to compare values of configuration options in a problematic system with those in other systems to infer the root cause of a failure. All these techniques require substantial effort to collect the baseline data.

2.2 Diagnosing cross-stack configuration errors

Configuration options in multi-layer architectures (e.g., LAMP, J2EE, or MEAN "software stacks") might easily contradict each other or be hard to trace to each other. Therefore, configuration error diagnosis in such architectures is particularly challenging [51]. On the other hand, so far there are very few research approaches or tools targeting this scenario [33].

Sayagh and Adams [32] conducted an empirical study on multi-layer configuration options across Wordpress (WP) plugins, WP, and the PHP engine. They discover a large and increasing number of configuration options used by WP and its plugins. In addition, over 85% of these options are used by at least two plugins at the same time.

Sayagh *et al.* [33] perform a qualitative analysis of over 1,000 configuration errors to understand their impact and complexity. Based on this data they develop a slicing-based approach to identify error-inducing configuration options in heterogeneous software stacks. So far it is the only approach which attempts to provide a complete, end-to-end process for diagnosing cross-stack configuration errors.

Work [4] focuses on finding configuration inconsistencies between layers in complex, multi-component software. The proposed technique (based on static analysis) can handle software that is written in multiple programming languages and has a complex preference structure.

In [31] the authors target the identification of configuration dependencies in multi-tiered enterprise applications. It provides a method for analyzing existing deployments to infer the configuration dependencies in a probabilistic sense. This yields rank-ordered list of dependencies so that administrators can consult it and systematically identify the true dependencies.

Authors of [12] attempt to quantify the challenges that configurability of complex, multi-component systems creates for software testing and debugging. It analyzes a highly-configurable industrial application and two open source applications. They notice that all three applications consist of multiple programming languages, limiting the applicability of static analysis. Furthermore, they find out that there many access points and methods to modify configurations, and that the configuration state of an application on failure cannot be determined only from persistent data.

2.3 Rules, Constraints and Fixing their Violations

Once configuration knowledge can be described using constraints or rules they can be used for diagnosis as well. The use of such knowledge is neither restricted to single-layer nor cross-stack applications in general. Hence, methods and techniques based on rules and constraints, which can also be seen as models of the applications, would provide a more general account to solve the software configuration error problem. In this section, we distinguish methods for learning knowledge, fixing violations, and inconsistency detection between different software artifacts.

Learning constraints and rules. Several existing approaches extract configuration models [42, 40, 54, 50, 55] and leverage them for configuration debugging, mainly via detecting value anomalies and rule violations.

The categories of extracted data constituting the models typically include the primitive and semantic *data types* of configuration options (e.g., integer, file path, port number, URL), the *value ranges* of options (minimum and maximum integer values or a list of acceptable values), the *control dependencies* (i.e., usage of parameter Q relies on the setting of another parameter P), and *value relationships* (e.g., value of parameter S should be greater than that of parameter T). EnCore [55] additionally considers the properties of the execution environment as a part of their models.

CODE [54] takes a unique approach and uses dynamic execution information as the model content, namely sequences of (Windows) registry accesses and derived rules. Using these rules for efficient filtering of even large lists of events, CODE can detect not only configuration errors but also deviant program executions. It requires no

source code, application-specific semantics, or heavyweight program analysis.

SPEX [50] analyzes source code to infer configuration option constraints and use these constraints to diagnose software misconfigurations, to expose misconfiguration vulnerabilities, and to detect error-prone configuration design and handling.

Build-time configuration settings. Another category of work addresses configurations and their constraints used at compilation and build time. Such configurations determine whether certain product features (e.g. logging, debugging) are activated, or even which software components are included in the shipped product. The later aspect is relevant e.g., for software product lines.

Works [22], [23] propose a static analysis approach to extract (build-time) configuration constraints from C code. Despite of its simplicity, it has high precision (77% - 93% in the studied systems) and can recover 28% of existing constraints. A further study of the authors reveals that configuration constraints enforce correct runtime behavior, improve users' configuration experience, and prevent corner cases.

Fixing violations of configuration constraints. The problem of fixing a configuration that violates one or more constraints is addressed in [47, 48]. The authors introduce to this purpose the concept of a range fix, which specifies the options to change the ranges of values for these options. They also design and evaluate an algorithm that automatically generates range fixes for a violated constraint. Empirical studies shows that the range fix approach provides mostly simple yet complete sets of fixes and has a moderate running time in the order of seconds.

Configurable software (e.g., Linux OS, eCos) can have very high number of options (variables) and constraints. E.g., Linux has over 6,000 variables and 10,000 constraints; eCos has over 1,000 variables and 1,000 constraints. Such systems typically use variability modeling languages and configuration tools (called *configurators*). Examples of variability languages include Linux Kconfig, eCos CDL, and feature models. With variability modeling languages and configurators, errors can be detected early, but users still have to resolve the errors, which is also not an easy task: the constraints in variability models can be very complex and highly interconnected. Therefore, researchers have proposed automated approaches that suggest a list of fixes for an error. A fix is a set of changes that, when performed on the configuration, resolve the current error. However, the recommended fixes in these approaches are sometimes large in number and size. For example, fix lists for eCos configurations contain up to nine fixes, and some fixes change up to nine variables.

In this context, work [39] proposes a method to reduce the size and complexity of error fixes by introducing a concept of *dynamic priorities*. The basic idea is to first generate one fix and then to gradually reach the desirable state based on user feedback. To this end, the approach (1) automatically translates user feedback into a set of implicit priority levels on variables, using five priority assignment and adjustment strategies and (2) efficiently identifies potentially desirable fixes that change only the variables with low priorities.

Detecting inconsistencies between code, documentation, and configuration files. Configuration options are widely used for customizing the behavior and initial settings of software applications, server processes, and operating systems. Their distinctive property is that each option is processed, defined, and described in different parts of a software project - namely in code, in configuration file, and in documentation. This creates a challenge for maintaining project consistency as it evolves. It also promotes inconsistencies leading to misconfiguration issues in production scenarios.

Confalyzer [30] uses static analysis to extract a list of configuration option from source code and from associated options documentation. Confalyzer first marks configuration APIs in the configuration classes. Then it identifies calls to these APIs in the program by building a call graph and obtains option names by reading values of parameters of these calls.

PrefFinder [11] proposed by Jin *et al*., uses static analysis and dynamic analysis techniques to extract configuration options and stores them in a database for query and use.

The SCIC approach [4] exploits Confalyzer to implement the functionality of extracting configuration options in the key-value model and the tree-structured model.

Work [6] proposes an approach for detection of inconsistencies between source code and documentation based on static analysis. It identifies source code locations where options are read and for each such location retrieves the name of the option. Inconsistencies are then detected by comparing the results against the option names listed in documentation.

2.4 Other Aspects

There are other papers dealing with diagnosis of software configuration errors not falling into the previous categories like testing, end-user support and performance optimization, which we discuss in this subsection.

Testing of highly configurable systems. Paper [18] presents an initial study on the potential of using statistical testing techniques for improving the efficiency of test selection for configurable software. The study aims to answer whether statistical testing can reduce the effort of localizing the most critical software faults, seen from user perspective.

Authors of [19] analyze program traces to characterize and identify where interactions occur on control flow and data. They find that the essential configuration complexity of these programs is indeed much lower than the combinatorial explosion of the configuration space indicates.

Work [36] proposes S-SPLat, a technique that combines heuristic sampling with symbolic search to explore enormous space of configurations for testing of software product lines.

A more general approach for testing configurable systems including software is combinatorial testing [15, 16]. There the underlying assumption is that it is not necessarily one configuration parameter that reveals a fault but a certain combination of parameters. Combinatorial testing assures to compute all combinations for any arbitrary subset of configuration parameters of arity k. In the context of combinatorial testing, the resulting test suite is said being of strength k. There are many algorithms and tools for combinatorial testing [13]. For a survey on combinatorial testing we refer the interested user to [26].

Configuration and debugging support for end-users. A technique to detect inadequate (i.e., missing or ambiguous) diagnostic messages for configuration errors issued by a configurable software system is proposed in [59]. It injects configuration errors and uses natural language processing to analyze the resulting diagnostic messages. It then identifies messages which might be unhelpful in diagnosis or even negatively impact this process.

Authors of [49] study configuration settings of real-world users from multiple projects and reveal patterns of unnecessary complexity in configuration design. The authors also provide a few guidelines to reduce the configuration space. Finally, the existing configuration navigation methods are studied in terms of their effectiveness in dealing with the over-designed configuration.

Work [28] introduces ConfSeer, a system which recommends to users suitable knowledge base articles which are likely to describe user's current configuration problem and its fix. To this end, ConfSeer takes the snapshots of configuration files from a user machine as input, then extracts the configuration parameter names and value settings from the snapshots and matches them against a large set of KB articles. If a match is found, ConfSeer pinpoints the configuration error with its matching KB article. The described system powers the recommendation engine behind Microsoft Operations Management Suite.

Optimizing performance via configuration settings. In [24], a rank-based approach to efficient creation of performance models is introduced. Such models can be exploited for finding an optimally performing configuration of a software system.

Authors of [10] conducted an empirical study on four popular software systems by varying software configurations and environmental conditions, to identify the key knowledge pieces that can be exploited for transfer learning for constructing performance models of configurable software systems.

Paper [35] proposes a multi-objective evolutionary algorithm to find the optimal solutions and addresses the configuration optimization problem for software product lines.

Finally, the work described in [27] employs random sampling and recursive search in a configuration space to find optimally performing configurations for an anticipated workload in software product lines.

2.5 Survey Summary

There are lots of papers dealing with configuration diagnosis of single layer applications often employing program analysis techniques but also making use of machine learning or replay methods. In case of more complicated applications comprising interacting and configurable software components there have been less papers dealing with concrete solutions. One approach that can be used in both cases of software is to make use of formalized knowledge about configurations, i.e., the configuration parameters, their domains, and rules specifying limitations and relationships among parameters. It would be interesting to investigate whether classical approaches to diagnosis of knowledge-bases like [8, 45, 9, 34] can also be successfully applied for configuration diagnosis. Other aspects, discussed in this section include testing configurations, end-user support, and performance optimization.

3 Challenges in Configuration Diagnosis

Based on the survey of papers presented in the previous section, we are able to identify several still open challenges. A general challenge that immediately arises is to distinguish whether an application failure is due to a fault in the configuration setup or code defect in the program. This is a common problem when applying configuration debugging tools, which usually assumes a certain cause. If we want to come up with a general approach for software configuration diagnosis, we have to adapt diagnosis to identify the underlying root cause.

A method that is able to separate these causes would take the current configuration, the program, the description of the execution environment, and the passing/failing tests as input. Based on these inputs the possible causes of a failure are provided as output. In order to come up with such an approach, it is necessary to have a close look at various configuration diagnosis problems, given consequently raise to the another challenge, i.e., providing an open repository of various configuration diagnosis problems that can be accessed by researchers in this field.

Such a general repository for software configuration diagnosis should include a larger set of different programs from single-layer to cross-stack applications together with configuration errors coming from different sources, test suites, and ideally also configuration knowledge bases. The repository should cover programs of different sizes and from different domains capturing currently available software to allow comparing different configuration diagnosis methods and techniques.

Besides these two general challenges, there are other challenges that are more specific to the applications (single-layer versus cross-stack) or the tasks to be tackled (i.e., fault localization and repair versus fault detection). In the following, we illustrate some of these more specific challenges in detail.

Diagnosis of single-layer applications Despite the fact that there have been various methods already published in this domain, there are still some open issues.

- Transfer techniques from functional fault localization: In case of software debugging, there are various methods available going beyond program analysis including spectrum-based fault localization [1, 2] among others. In this approach, code regions are ranked (essentially) according to the number of times there are executed by passing or by failing tests (intuition: if a code line is executed primarily by failing tests, it is more likely to contribute to a failure). For a detailed look at current debugging techniques we refer the interested reader to Wong et al.'s survey [46]. In particular spectrum-based fault localization offers superior performance compared to static and dynamic program analysis applied to debugging. The open research question that is, whether spectrum-based fault localization can be efficiently used for software configuration diagnosis as well.
- Study and exploit the trade-off between the type of data from users required for diagnosis (as well as the effort of obtaining this data, e.g., via instrumentation) and the achieved accuracy. The research goals that would go into this direction include:
  - For each type of diagnosis data (from static analysis to diagnosis data dynamically created from instrumentation and also

for combinations) understand and quantify the degree of likely penalties (e.g., in terms of accuracy) of using only this data for diagnosis. Specifically, characterize error types which can be or *cannot be* diagnosed for each type of diagnosis data (when using state-of-the art debugging approaches).

– For each "class" of diagnosis data, attempt to improve the corresponding state-of-the art diagnosis methods in terms of types of errors they are able to debug. This can be done e.g., by an in-depth analysis why they fail for some error types and by providing substrates/replacements for the missing diagnosis data.

Diagnosing of cross-stack configuration errors In the case of cross-stack applications, there is not so much work available. Important open research challenges include:

- Exploit work on consistency checking to detect potential inconsistencies between different stack layers.
- Leverage existing work on extraction of rules and constraints to model dependencies *between* layers. Then use the techniques for discovery and fixing of constraint violations to diagnose (and possibly repair) cross-stack configuration errors.
- As a further application of extracted rules, configurator-like tools (as used for configuring operating systems) could be used for safe configuration of cross-stack systems.
- Create models of expected behavior (given a current global configuration) of each layer from the perspective of each layer. Divergences in the behavior might indicate potential configuration inconsistencies or errors. For example, given the current configuration of a database-layer (specifying n_1 database connections), also the PHP-layer should allow n_1 database connections. However, if the expected behavior of PHP-layer, based on its own configuration, allows only $n_2 < n_1$ database-connections, then an inconsistency between these two behavioral models is indicated.

It is worth noting that it is quite important which dependencies or interaction between layers can be observed or recorded. Moreover, in the context of these challenges the application of model-based approaches for diagnosing (configuration) knowledge-base, e.g., [8, 45, 9, 34], might be worth being considered.

Testing-related challenges and goals In case of testing, we are interested in detecting faults caused by configuration settings. There the motivation is to improve testing approaches specifically for detecting faults in system configurations ideally during software development. To clarify the meaning of "software testing" in context of configuration (errors) we should consider that an application failure in this context does not necessarily imply that there is a defect in code (as in traditional testing). Such a failure rather indicates that:

- There is a mismatch between the state of the application environment (operating system, file system, hardware, location of input data, libraries, network properties, remote components, etc.) and the configuration settings. This implies that a test for this type of error must take into consideration the environment.
- There is an inconsistency between configuration values, either within a single layer or between layers in a multi-layer application. The corresponding tests might be independent of the application environment, but are probably more comprehensive if this is also taken into account.

Consequently, this discussion gives rise to the following goals:

- Attempt automated test generation that considers the state of the application environment and the configuration settings (maybe implicitly). Such tests would adapt to environment changes and target only the above-mentioned mismatch between environment and configuration. In order to avoid confusion with the meaning of traditional testing, we might call this "configuration verification" step instead of testing.
- Generate tests that verify only the *consistency of configurations between layers* of a multi-stack system. In this case a test failure should indicate only an inconsistency, not a lack of adaptation to the production environment. For example, a test could only verify the consistency of configurations across layers, not execute the whole application.
- Generate tests which verify the correctness of application's behavior independently of the configuration settings. For example, an application should produce the same behavior independently of the exact path to input/output/libraries, number of used threads (in some range), used compiler (or its flags) etc.
- Generate tests that improve the outcome of fault localization. There it would be necessary to identify those tests that can distinguish different computed root causes (see e.g., [21]).

4 Threats to Validity

Several threats to validity of this paper exist. The main one is the risk of omitting important contributions to this field. To mitigate this risk, we have created lists of relevant works using several processes described below. We then merged and pruned the results according to the rank of the publishing venue and originality (i.e. works proposing a novel or distinctive approach were included even if published in a workshop). In the first literature collection process we searched for publications containing word "configuration" which were published in selected high-quality venues (ICSE, ASE, ISSTA, FSE, ISSRE, ICSME, ICPC, IEEE Trans. Software Eng., and some others) in the last five years; for each found publication, we verified via abstract whether a publication indeed targets configuration error (diagnosis). In the second process, we read the related work sections of the previously identified works, and created a list of discussed there which are of relevance (here, also less prestigious venues were considered). Finally, we screened the survey [51] for checking that no important contribution is omitted.

Another threat to validity is the possibility to misinterpret any of the discussed works (e.g. due to different understanding of terms), and state here inaccurate claims. To reduce this risk, we have studied each described contribution in a depth sufficient to avoid a misinterpretation. Besides of this, information from related work section to verify our interpretation was used where available.

5 Conclusion

In this paper, we presented a survey on methods and techniques used for detecting, localizing, and correcting faults in the context of software configurations. We distinguished the different cases of software

configuration diagnosis for single-layer and cross-stack applications as well as methods used in case of available configuration knowledge and further aspects. From the survey we were able to identify some still open challenges and research questions including distinguishing different variants of potential root causes, the lack of repositories of application-cases for validating and comparing research results as well as the need for new fault localization and testing methods.

The motivation for this paper is to provide a solid basis for future research in this area and to identify some important challenges in software configuration diagnosis worth being tackled. We also indicated some relationships with work on diagnosis of configuration knowledge bases and other approaches of software debugging that might stimulate this field. Because of the growing interest in providing programs comprising a stack of other programs that themselves can be configured, we see a growing need for research in this area.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund, 'A practical evaluation of spectrum-based fault localization', *Journal of Systems and Software*, **82**(11), 1780–1792, (2009).

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund, 'Spectrum-based multiple fault localization', in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pp. 88–99. IEEE Computer Society, (2009).

[3] Mona Attariyan and Jason Flinn, 'Automating Configuration Troubleshooting with Dynamic Information Flow Analysis', in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pp. 1–11, Vancouver, BC, Canada, (2010). USENIX Association.

[4] Farnaz Behrang, Myra B. Cohen, and Alessandro Orso, 'Users Beware: Preference Inconsistencies Ahead', in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 295–306, New York, NY, USA, (2015). ACM.

[5] Jon Brodkin. Why Gmail Went Down: Google Misconfigured Load Balancing Servers. `https://goo.gl/Hdga7H`. Accessed: 5 June 2018.

[6] Z. Dong, A. Andrzejak, D. Lo, and D. Costa, 'ORPLocator: Identifying Read Points of Configuration Options via Static Analysis', in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 185–195, (October 2016).

[7] Z. Dong, A. Andrzejak, and K. Shao, 'Practical and accurate pinpointing of configuration errors using static analysis', in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 171–180, (September 2015).

[8] A Felfernig, G Friedrich, D Jannach, and M Stumptner, 'Consistency-based diagnosis of configuration knowledge bases', *Artificial Intelligence*, **152**(2), 213–234, (2004).

[9] A. Felfernig, M. Schubert, and C. Zehentner, 'An efficient diagnosis algorithm for inconsistent constraint sets', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **26**(1), 53–62, (2 2012).

[10] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal, 'Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis', in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pp. 497–508, Piscataway, NJ, USA, (2017). IEEE Press.

[11] Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson, 'PrefFinder: Getting the Right Preference in Configurable Software Systems', in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pp. 151–162, New York, NY, USA, (2014). ACM.

[12] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson, 'Configurations Everywhere: Implications for Testing and Debugging in Practice', in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pp. 215–224, New York, NY, USA, (2014). ACM.

[13] Sunint Kaur Khalsa and Yvan Labiche, 'An orchestrated survey of available algorithms and tools for combinatorial testing', in *25th International Symposium on Software Reliability Engineering*, pp. 323–334, (2015).

[14] Bogdan Korel and Janusz Laski, 'Dynamic Program Slicing', *Information Processing Letters*, **29**, 155–163, (1988).

[15] D. R. Kuhn, R. N. Kacker, and Y. Lei, 'Combinatorial testing', in *Encyclopedia of Software Engineering*, ed., Phillip A. Laplante, Taylor & Francis, (2012).

[16] D. Richard Kuhn, Renee Bryce, Feng Duan, Laleh Sh. Ghandehari, Yu Lei, and Raghu N. Kacker, 'Combinatorial testing: Theory and practice', in *Advances in Computers*, volume 99, 1–66, Elsevier, (2015).

[17] Max Lillack, Christian Kästner, and Eric Bodden, 'Tracking Load-time Configuration Options', in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pp. 445–456, New York, NY, USA, (2014). ACM.

[18] Dusica Marijan, 'Improving Configurable Software Testing with Statistical Test Selection', in *Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems*, ForMABS 2016, pp. 5–8, New York, NY, USA, (2016). ACM.

[19] J. Meinicke, C. P. Wong, C. Kästner, T. Thüm, and G. Saake, 'On essential configuration complexity: Measuring interactions in highly-configurable systems', in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 483–494, (September 2016).

[20] James Mickens, Martin Szummer, and Dushyanth Narayanan, 'Snitch: Interactive Decision Trees for Troubleshooting Misconfigurations', in *Proceedings of the 2Nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, pp. 8:1–8:6, Cambridge, MA, (2007). USENIX Association.

[21] Nica Mihai, Nica Simona, and Wotawa Franz, 'On the use of mutations and testing for debugging', *Software: Practice and Experience*, **43**(9), 1121–1142, (2013).

[22] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, 'Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study', *IEEE Transactions on Software Engineering*, **41**(8), 820–841, (August 2015).

[23] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki, 'Mining Configuration Constraints: Static Analyses and Empirical Results', in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 140–151, New York, NY, USA, (2014). ACM.

[24] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel, 'Using Bad Learners to Find Good Configurations', in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 257–267, New York, NY, USA, (2017). ACM.

[25] ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S. Foster, and Adam A. Porter, 'iGen: Dynamic Interaction Inference for Configurable Software', in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 655–665, New York, NY, USA, (2016). ACM.

[26] Changhai Nie and Hareton Leung, 'A survey of combinatorial testing', *ACM Computing Surveys*, **43**(2), (January 2011).

[27] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund, 'Finding Near-optimal Configurations in Product Lines by Random Sampling', in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 61–71, New York, NY, USA, (2017). ACM.

[28] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain, 'ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection', *Proc. VLDB Endow.*, **8**(12), 1828–1839, (August 2015).

[29] Ariel Rabkin and Randy Katz, 'Precomputing Possible Configuration Error Diagnoses', in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 193–202, Washington, DC, USA, (2011). IEEE Computer Society.

[30] Ariel Rabkin and Randy Katz, 'Static Extraction of Program Configuration Options', in *Proceedings of the 33rd International Conference on*

Software Engineering, ICSE '11, pp. 131–140, New York, NY, USA, (2011). ACM.

[31] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury, 'Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications', in *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pp. 169–178, New York, NY, USA, (2009). ACM.

[32] M. Sayagh and B. Adams, 'Multi-layer software configuration: Empirical study on wordpress', in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 31–40, (September 2015).

[33] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams, 'On Cross-stack Configuration Errors', in *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pp. 255–265, Piscataway, NJ, USA, (2017). IEEE Press.

[34] Kostyantyn Shchekotykhin, Gerhard Friedrich, Patrick Rodler, and Philipp Fleiss, 'Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation', in *ECAI '14*, pp. 813–818, (2014).

[35] K. Shi, 'Combining Evolutionary Algorithms with Constraint Solving for Configuration Optimization', in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 665–669, (September 2017).

[36] S. Souto, M. D'Amorim, and R. Gheyi, 'Balancing Soundness and Efficiency for Practical Testing of Configurable Systems', in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 632–642, (May 2017).

[37] Ya-Yunn Su, Mona Attariyan, and Jason Flinn, 'AutoBash: Improving Configuration Management with Operating System Causality Analysis', in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pp. 237–250, Stevenson, Washington, USA, (2007). ACM.

[38] T. Uchiumi, S. Kikuchi, and Y. Matsumoto, 'Misconfiguration detection for cloud datacenters using decision tree analysis', in *Network Operations and Management Symposium (APNOMS), 2012 14th Asia-Pacific*, pp. 1–4, (September 2012).

[39] Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang, 'SmartFixer: Fixing Software Configurations Based on Dynamic Priorities', in *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pp. 82–90, New York, NY, USA, (2013). ACM.

[40] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-min Wang, 'Automatic Misconfiguration Troubleshooting with PeerPressure', in *In OSDI*, pp. 245–258, (2004).

[41] Mengliao Wang, Xiaoyu Shi, and K. Wong, 'Capturing Expert Knowledge for Automated Configuration Fault Diagnosis', in *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*, pp. 205–208, (June 2011).

[42] Yi-min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, and Chun Yuan, 'STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support', in *In Usenix LISA*, pp. 159–172, (2003).

[43] Mark Weiser, 'Program slicing', *IEEE Transactions on Software Engineering*, **10**(4), 352–357, (July 1984).

[44] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble, 'Configuration Debugging As Search: Finding the Needle in the Haystack', in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pp. 6–6, San Francisco, CA, (2004). USENIX Association.

[45] Jules White, David Benavides, Douglas C. Schmidt, Pablo Trinidad, Brian Dougherty, and Antonio Ruiz Cortés, 'Automated diagnosis of feature model configurations', *Journal of Systems and Software*, **83**(7), 1094–1107, (2010).

[46] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa, 'A survey on software fault localization', *IEEE Trans. Software Eng.*, **42**(8), 707–740, (2016).

[47] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, 'Range Fixes: Interactive Error Resolution for Software Configuration', *IEEE Transactions on Software Engineering*, **41**(6), 603–619, (June 2015).

[48] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki, 'Generating Range Fixes for Software Configuration', in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 58–68, Piscataway, NJ, USA, (2012). IEEE Press.

[49] 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*, ESEC/FSE 2015, pp. 307–319, New York, NY, USA, (2015). ACM.

[50] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy, 'Do Not Blame Users for Misconfigurations', in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 244–259, Farminton, Pennsylvania, (2013). ACM.

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

[52] 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, pp. 159–172, New York, NY, USA, (2011). ACM.

[53] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy, 'SherLog: Error Diagnosis by Connecting Clues from Run-time Logs', in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pp. 143–154, New York, NY, USA, (2010). ACM.

[54] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar, 'Context-based Online Configuration-error Detection', in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, pp. 28–28, Portland, OR, (2011). USENIX Association.

[55] 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 *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 687–700, Salt Lake City, Utah, USA, (2014). ACM.

[56] Sai Zhang, 'ConfDiagnoser: An Automated Configuration Error Diagnosis Tool for Java Software', in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 1438–1440, Piscataway, NJ, USA, (2013). IEEE Press.

[57] Sai Zhang and Michael D. Ernst, 'Automated diagnosis of software configuration errors', in *ICSE'13, Proceedings of the 34th International Conference on Software Engineering*, San Francisco, CA, USA, (May 2013).

[58] Sai Zhang and Michael D. Ernst, 'Which Configuration Option Should I Change?', in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 152–163, New York, NY, USA, (2014). ACM.

[59] Sai Zhang and Michael D. Ernst, 'Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors', in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pp. 12–23, New York, NY, USA, (2015). ACM.