Ctest4J: A Practical Configuration Testing Framework for Java

Shuai Wang
University of Illinois Urbana-Champaign
Urbana, USA
swang516@illinois.edu

Xinyu Lian
University of Illinois Urbana-Champaign
Urbana, USA
lian7@illinois.edu

Qingyu Li
University of Illinois Urbana-Champaign
Urbana, USA
qingyul2@illinois.edu

Darko Marinov
University of Illinois Urbana-Champaign
Urbana, USA
marinov@illinois.edu

Tianyin Xu
University of Illinois Urbana-Champaign
Urbana, USA
tyxu@illinois.edu

ABSTRACT

We present Ctest4J, a practical configuration testing framework for Java projects. Configuration testing is a recently proposed approach for finding both misconfigurations and code bugs. Ctest4J addresses the limitations of configuration testing scripts from prior work, including lack of parallel test execution, poor maintainability due to external dependencies, limited integration with modern build systems, and the need for manual instrumentation of configuration API. Ctest4J is a unified framework to write, maintain, and execute configuration tests (Ctests) and integrates with multiple testing frameworks (JUnit4, JUnit5, and TestNG) and build systems (Maven and Gradle). With Ctest4J, Ctests can be maintained similarly to regular unit tests. Ctest4J also provides a utility for automated code instrumentation for common configuration API. We evaluate Ctest4J on 12 open-source projects. We show that Ctest4J effectively enables configuration testing for these projects and speeds up Ctest execution by 3.4X compared to prior scripts. Ctest4J can be found at https://github.com/xlab-uiuc/ctest4j.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Configuration testing, Software testing, Software reliability

ACM Reference Format:
Shuai Wang, Xinyu Lian, Qingyu Li, Darko Marinov, and Tianyin Xu. 2024. Ctest4J: A Practical Configuration Testing Framework for Java. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663799

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663799

1 INTRODUCTION

Configuration testing [10, 14] is a recently proposed approach for rigorously testing software configurations, similar to how software code is tested today. The key motivation is to test production system configurations before deploying them. Configuration testing connects configurations to software tests so that configuration changes can be tested in the context of code affected by the changes. A configuration test (Ctest) is a test that takes as input a system configuration and checks the configuration with the code. In many projects, configurations are key-value pairs that map configuration parameters to their values. Prior work [10] has shown that configuration testing outperforms previous approaches [1, 11, 16–18] for detecting failure-inducing configurations, including sophisticated misconfigurations and valid configurations that trigger dormant software bugs. Regression test selection [10, 12] and test case prioritization [2] have been developed to make configuration testing more efficient for continuous delivery and deployment.

However, despite the active research on configuration testing from several groups [2, 4, 6, 10, 12, 14], including configuration tests for fuzzing [4] and unsafe parameter detection [6], there is no practical, systematic framework for configuration testing. Prior research developed *ad hoc* scripts [7], which are very limited and deficient for practical use cases—they do not support parallel test execution, have poor maintainability due to external dependencies (requiring a file that specifies the mapping between configuration parameters and the tests that use them), work only for JUnit4 and Maven, and require manual instrumentation of the configuration API. Such deficient support makes it harder to adopt configuration testing in practice and even hampers research. For example, all prior papers [2, 4, 10, 12, 14] evaluated configuration testing on a fixed set of five or six open-source projects.

We present Ctest4J, a practical configuration testing framework for Java projects. Ctest4J provides new annotations so that developers can write and maintain Ctests similarly to regular unit tests. It connects the configuration values under test with the corresponding Ctests. Ctest4J also provides automated code instrumentation for common configuration API. The instrumentation is required to enable Ctests. Ctest4J supports the most popular Java-based test frameworks (JUnit4, JUnit5, TestNG) and build systems (Maven, Gradle). In sum, Ctest4J provides the following features:

- **Parallel test execution.** Ctest4J supports running Ctests with different configurations in parallel, addressing the limitation of sequential-only execution of prior Ctest scripts (due to a design limitation of sharing static configuration objects). Also, Ctest4J incurs a low runtime overhead; a Ctest runs only ~3% slower on average than a regular unit test.
- **Maintainability.** Ctest4J provides source-code annotations to make Ctests easy to maintain. Ctest4J allows developers to specify the mapping between Ctest and configuration parameters as annotations inside the test code, without creating a dependency on separate, external files. For backward compatibility, Ctest4J supports original mapping files for projects that use prior scripts.
- **Automation.** Configuration API instrumentation is a necessary step to enable Ctests. Ctest4J uses AspectJ to automate the instrumentation, thereby easing the adoption of configuration testing for existing projects. With our automation, enabling Ctests takes only 20 lines of code on average across 12 Java projects.

Ctest4J is available at https://github.com/xlab-uiuc/ctest4j and released in the Maven Central Repository.

2 USAGE

We present a high-level overview of configuration testing with Ctest4J. More details are in the code documentation on GitHub.

2.1 Writing Ctests

Ctest4J provides two source-code annotations for developers to mark Ctest classes and methods:

- `@CtestClass` marks that a class is a Ctest class;
- `@Ctest` marks that a method is a Ctest method;

These annotations specify the configuration parameters of the corresponding Ctest(s) in a class or a method. Making the configuration parameter usage explicit (1) substantially aids debugging (as developers gain insights into which configuration parameters are utilized by a Ctest), (2) enhances the capabilities of test selection [10, 12] and prioritization [2], and (3) enables more efficient Ctest fuzzing [4].

Both annotations can specify configuration parameters used by the Ctest(s) as a list of parameters and a regular expression to match the parameters. For backward compatibility with the original Ctest scripts [7], `@CtestClass` can also specify a file that contains the mapping between configuration parameters and Ctest methods. Lists, regexes, and files can be provided together, and Ctest4J unions the mappings into one final mapping. Besides these annotations, Ctest4J's Ctest runner needs to be specified within the Ctest class. For example, Ctest4J's `CtestJUnitRunner` is added through the usual `@RunWith` annotation in JUnit4.

Figure 1 is a simplified Ctest example from Hadoop YARN. We annotate the existing class `TestFSDownload` with the Ctest4J's runner `CtestJUnitRunner` (line 1) and `@CtestClass` (line 2), and the test method `testDownload` with `@Ctest` (lines 8-10). `@CtestClass` specifies the class-level configuration parameter used by all the Ctest methods in the class. The class-level configuration parameters mostly come from the test setup and teardown executions. In this example, the method `getRecordFactory()` (line 6) uses the parameter `yarn.ipc.record.factory.class`, making it a class-level configuration parameter. `@Ctest` specifies the method-level configuration parameters used in the execution of the Ctest method

```
1  @RunWith(CtestJUnitRunner.class)
2  @CtestClass({"yarn.ipc.record.factory.class"})
3  public class TestFSDownload {
4    private Configuration conf = new Configuration();
5    static final RecordFactory recordFactory =
6      RecordFactoryProvider.getRecordFactory(null);
7
8    @Ctest(regex="fs.(client.resolve.remote.symlinks|" +
9           "permissions.umask-mode|local.block.size|" +
10          "AbstractFileSystem.file.impl)")
11   public void testDownload() {
12     // Create FileContext with parameters in @Ctest
13     FileContext files =
14       FileContext.getLocalFSFileContext(conf);
15     ...
16     // Start downloading
17     FSDownload fsd = new FSDownload(files, ...)
18     Path path = fsd.download(...);
19     ...
20     // Check whether the download is done
21     assertTrue(path.isDone())
22     ... // Check other properties of the downloaded file
23   }
24 }
```

Figure 1: A Ctest in YARN with Ctest4J. The configuration parameters in the code snippet will be instantiated by values from the configuration under test (not shown in the figure).

body. In this example, `testDownload` uses four method-level configuration parameters, specified for illustration through a regular expression. These parameters are used by various methods, e.g., `createFileSystem()` method invoked by `getLocalFSFileContext()` uses `local.block.size`. During Ctest execution, Ctest4J instantiates every configuration parameter used by each Ctest method with the configuration under test, e.g., a production configuration.

2.2 Configuration API Instrumentation

To enable Ctests in a Java project, developers need to instrument the configuration API with Ctest4J so that Ctest4J can instantiate Ctests with the configuration under test at runtime. Ctest4J also provides APIs to track the usage of configuration parameters during the execution of Ctests; the tracking is important for debugging, maintenance of the input configuration parameters for each Ctest, and adequacy measurement (e.g., coverage of configuration parameters of a Ctest suite).

Ctest4J focuses on common configuration API patterns in Java projects, many of which use a unified configuration class with two basic API abstractions, configuration GET and SET APIs [5, 8–10, 13, 15, 19, 20]. The GET APIs of the form "`<T> get(String parameter)`" take a parameter name and return a value; SET APIs of the form "`void set(String parameter, <T> value)`" set the value of the given parameter with the input `value`. Configuration APIs built on top of the common `java.util.Properties` and `org.apache.commons.configuration` all follow such a pattern.

2.2.1 Instrumenting Configuration API. Without Ctest4J, all test executions would use only the default configuration provided with the project. Ctest4J modifies the execution so that tests run with the configuration under test (e.g., a production configuration). Ctest4J instruments the configuration API with `connectProdConfig`, which connects the configuration under test to the configuration objects used by the Ctests. The `connectProdConfig` method is a static method that takes the configuration SET API as input; typically,

`connectProdConfig` is added to the constructor of the configuration class to initialize the configuration object with the configuration under test. The following snippet shows how to instrument the configuration API of Apache Hadoop YARN.

```
1  public Configuration() {
2      this(true);
3  +   Ctest4J.connectProdConfig(
4  +       (name, value) -> set(name, (String) value));
5  }
```

For projects that do not have a unified configuration class where Ctest4J can modify the SET API, Ctest4J supports a lazy instrumentation mode that modifies the configuration GET API to use the configuration under test.

2.2.2 Tracking Configuration Parameters. Ctest4J also provides an API (a static `markParamAsUsed` method) to track the usage of configuration parameter values during the execution of Ctests. Ctest4J calls `markParamAsUsed` upon the invocation of a configuration GET API, as shown in the following snippet (from Hadoop).

```
1  public String get(String name) {
2  +   Ctest4J.markParamAsUsed(name);
3      String[] names = handleDeprecation(deprecationContext.get
           (), name);
4      String result = null;
5      for(String n : names) {
6  +       Ctest4J.markParamAsUsed(n);
7          result = substituteVars(getProps().getProperty(n));
8      }
9      return result;
10 }
```

For projects that have multiple different configuration GET APIs, `markParamAsUsed` is expected to be placed for every API to ensure the completeness of tracking.

Ctest4J provides an AspectJ [3] based utility to instrument the configuration API (by specifying the fully qualified name of the APIs), if source-code changes are not preferred.

2.3 Running Ctests

Running a Ctest is similar to running a regular unit test. For example, with Maven, Ctests can be run with Maven Surefire (`mvn test`). Ctest4J provides three modes to run Ctests:

- `debug`: run the Ctest with the default configuration and check whether all the required configuration parameters are used during the test execution. This mode helps in developing and debugging Ctests;
- `prod`: run the Ctest with the configuration under test;
- `default`: run the Ctest with the configuration under test and check whether all the required configuration parameters are used during the test execution.

Ctest4J supports running Ctests in parallel with different configuration files. Ctest4J also supports input configuration through command-line arguments. If no configuration file or command-line argument is specified, Ctest4J runs the Ctest with the default configuration; in this case, a Ctest falls back into a regular unit test.

Ctest4J implements parameter-aware Ctest selection [10]. We plan to develop advanced test selection algorithms such as uRTS [12] and test case prioritization algorithms [2] in Ctest4J.

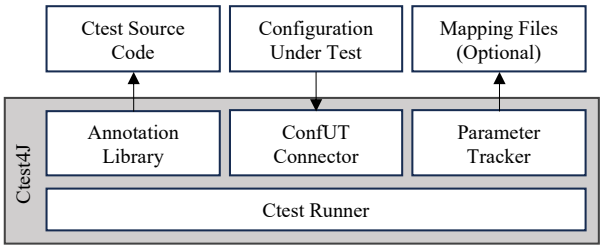

Figure 2: Overview of Ctest4J.

3 IMPLEMENTATION

The current Ctest4J implementation has ~5000 lines of Java code. Ctest4J takes three inputs: (1) the Ctest code, (2) the configuration under test (ConfUT), and (3) optionally a mapping file between the parameters and the Ctest that uses them (the original Ctest scripts [7] required such files). Ctest4J processes the test annotations (with explicit parameter list or the file name) and instantiates the execution of Ctests. Figure 2 shows the four main components of Ctest4J. We next briefly describe their implementations.

3.1 Annotation Library

The annotation library processes the annotations in the test code. The annotations are conceptually similar to the `@Test` annotation for regular unit tests. `@CtestClass` and `@Ctest` mark that the annotated class or method, respectively, is a Ctest. As described in §2.1, both annotations specify the configuration parameters of the Ctest using a list or a regex. To get a precise set of configuration parameters from a regex, Ctest4J does not accept patterns with match-any operators `".*"` or `".+"`, which could match too many parameters.

`@CtestClass` can also specify configuration parameters from a mapping file in JSON that has two fields: class-level parameters (a list of configuration parameters required by all Ctests in the class) and method-level parameters (a map between Ctest method names and configuration parameters required by the method).

3.2 ConfUT Connector

The configuration connector connects the ConfUT with Ctests, effectively to run each test with the configuration under test (rather than the default configuration). The original Ctest scripts [7] implemented the connector by writing the ConfUT to a dedicated configuration file and changing the code for the initialization of the configuration object to read the dedicated file and instantiate the Ctest for execution. However, the dedicated file was shared among all Ctests, and Ctests could not run concurrently with different configurations. As different Ctests may be suitable for testing different scenarios, some projects (e.g., HDFS) have various configuration files (e.g., in `test/resources`) for different tests.

Ctest4J's configuration connector directly writes the ConfUT into the configuration object via the configuration SET APIs, with no dedicated configuration file. For each Ctest run, it creates a map with each configuration parameter and its value from the ConfUT. To create configuration objects during Ctests run, the instrumented configuration API invokes the Ctest4J's `connectProdConfig` method. The connector uses the SET API to instantiate each configuration parameter with the corresponding value. The design enables Ctest4J to support parallel execution of Ctests with distinct configurations.

3.3 Parameter Tracker

The parameter tracker monitors the usage of configuration parameters during Ctests runs. For a given Ctest class, the tracker manages two levels of parameter usage list: (1) class-level list records the configuration parameters used by all Ctest methods within the class, including the `Before` and `After` methods; and (2) method-level list records the configuration parameters used by each Ctest method. Invocations of the instrumented configuration GET APIs call the tracker. To record the parameter in the class- and method-level lists, the tracker distinguishes whether the test execution is in the shared, class-level setup and teardown (`BeforeClass`, `Before`, `After`, and `AfterClass` methods) or in the method-level body execution.

3.4 Ctest Runner

The Ctest4J runner launches the Ctests using the ConfUT connector and the parameter tracker. For each test class, the runner first checks if the class is annotated with `@CtestClass`. If not, the runner executes the test as a regular unit test but issues a warning about the missing Ctest annotation. Otherwise, the Ctest4J runner executes the test class as a Ctest and proceeds to extract the required configurations for both the Ctest class and its methods from the annotations.

For each Ctest class, the runner creates a new connector and a new tracker. The isolation is important, as it ensures that the execution of one Ctest class does not interfere with the others.

The runner fails Ctests under any of the following conditions:

- *missing parameter usage*: the Ctest fails if the tracker identifies a required parameter that was not used during the Ctest run;
- *exceptions or errors*: the Ctest fails if it encounters an exception or error during execution (as for a regular unit test);
- *timeout*: the Ctest fails if its execution fails to finish within a specified timeout (as for a regular unit test).

3.5 Integration with Testing Frameworks

We integrated Ctest4J with JUnit4, JUnit5, and TestNG. To support JUnit4, we implement the Ctest runner as a custom JUnit4 runner that extends the `BlockJUnit4ClassRunner` class and implements the `CtestRunner` interface. We integrated Ctest4J with JUnit5 as a JUnit5 extension and with TestNG as a TestNG listener. To integrate with a new testing framework, one needs to implement the `CtestRunner` interface, which involves invoking the connector and tracker methods in the target framework's runner or listener.

4 EVALUATION

We evaluate Ctest4J using 12 open-source Java projects, including all five projects used in prior Ctest work [2, 4, 10, 12] (Alluxio, HBase, HCommon, HDFS, ZooKeeper) and seven new projects (Figure 3). These projects use different testing frameworks (eight JUnit4, three JUnit5, one TestNG) and different build systems (ten Maven, two Gradle). We report our experience of enabling Ctests using Ctest4J for the 12 projects and the performance of Ctest4J.

4.1 Enabling Ctests

We enabled Ctests for 12 mature, widely used Java projects using Ctest4J. The main effort is to understand each project's configuration API to instrument the configuration API (see §2.2). It took us

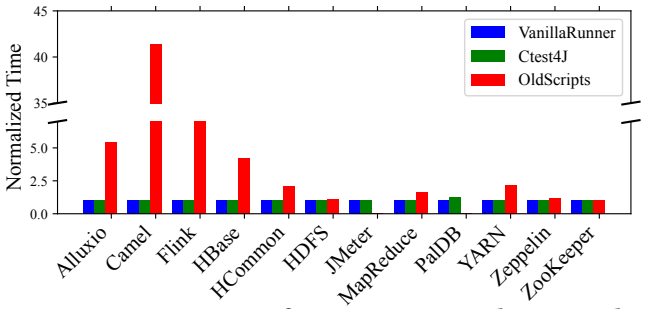

Figure 3: Execution time of running Ctests with Ctest4J, the Ctest scripts [7] (OldScripts), and the vanilla runner such as JUnit4 (VanillaRunner), normalized by VanillaRunner.

on average around one hour to find the correct configuration API for each project. We were able to use our Ctest4J's AspectJ utility to instrument the configuration API. Note that no paper author is a developer on any evaluated project, so we expect developers more familiar with the configuration API of their projects to add Ctest4J's instrumentation even faster. With the instrumentation in place, we transform existing tests that use configuration parameters into Ctests following the original approach [10]. We write scripts to automatically add the Ctest4J annotations in the Ctest code.

4.2 Performance

We measure the Ctest running time using Ctest4J and compare it with the original Ctest scripts [7]. We also measure the overhead of Ctest4J by comparing the running time with and without Ctest4J (using default configuration). Note that we only use the original scripts to run ten projects because the scripts do not support Gradle. We use the default parallelism configured in the projects. Ctest4J supports parallel execution, not requiring tests to run sequentially.

Figure 3 shows that Ctest4J can speed up the Ctest execution by up to 41.3X times (3.4X on average), compared to the original Ctest scripts. The speedup mostly comes from the parallel execution of Ctest4J, while the original scripts need to run Ctests one by one. Therefore, for projects that configure high parallelism for test execution (e.g., Alluxio, Camel, and Flink), the speedup is significant; for projects that run tests sequentially, the difference is smaller.

The overhead of Ctest4J is negligible. It mainly comes from the configuration usage tracking and additional checking logic in Ctest4J that checks whether all the required parameters are used during test execution. In Figure 3, the execution time of Ctest4J compared with VanillaRunner is up to 1.27X and 1.03X on average.

5 CONCLUSION

We present Ctest4J, a practical configuration testing framework for Java. Ctest4J can help Java projects enable configuration testing with modest manual effort and low runtime overhead. Ctest4J provides direct support for writing and maintaining configuration tests. We aim to broaden configuration testing research and reduce the barrier to adopting configuration testing in practice.

Acknowledgements. We thank the reviewers for their useful comments. This work was supported in part by NSF grants CCF-1763788, CCF-1956374, and CNS-2145295. We acknowledge support from Meta, Microsoft, and Qualcomm.

REFERENCES

[1] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. https://doi.org/10.1145/3368089.3409727

[2] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. https://doi.org/10.1145/3460319.3464810

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*.

[4] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems. In *Proceedings of the 46th International Conference on Software Engineering (ICSE'24)*. https://doi.org/10.1145/3597503.3623315

[5] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-time Configuration Options. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14)*. https://doi.org/10.1109/TSE.2017.2756048

[6] Sixiang Ma, Fang Zhou, Michael D. Bond, and Yang Wang. 2021. Finding heterogeneous-unsafe configuration parameters in cloud systems. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*. https://doi.org/10.1145/3447786.3456250

[7] OpenCtest. 2020. Research Artifact for "Testing Configuration Changes in Context to Prevent Production Failures". https://github.com/xlab-uiuc/openctest.

[8] Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. https://doi.org/10.1145/1985793.1985812

[9] Mohammed Sayagh, Zhen Dong, Artur Andrzejak, and Bram Adams. 2017. Does the Choice of Configuration Framework Matter for Developers? Empirical Study on 11 Java Configuration Frameworks. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM'17)*. https://doi.org/10.1109/SCAM.2017.25

[10] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*.

[11] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)*. https://doi.org/10.1145/2815400.2815401

[12] Shuai Wang, Xinyu Lian, Darko Marinov, and Tianyin Xu. 2023. Test Selection for Unified Regression Testing. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*. https://doi.org/10.1109/ICSE48619.2023.00145

[13] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[14] Tianyin Xu and Owolabi Legunsen. 2019. Configuration Testing: Testing Configuration Values Together with Code Logic. *CoRR* abs/1905.12195 (July 2019). https://doi.org/10.48550/arXiv.1905.12195

[15] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP'13)*. https://doi.org/10.1145/2517349.2522727

[16] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR)* 47, 4 (July 2015). https://doi.org/10.1145/2791577

[17] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static Detection of Silent Misconfigurations with Deep Interaction Analysis. In *Proceedings of the 36th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'21)*. https://doi.org/10.1145/3485517

[18] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*. https://doi.org/10.1145/2644865.2541983

[19] Sai Zhang and Michael D. Ernst. 2013. Automated Diagnosis of Software Configuration Errors. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. https://doi.org/10.1109/ICSE.2013.6606577

[20] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. 2021. An Evolutionary Study of Configuration Design and Implementation in Cloud Systems. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. https://doi.org/10.1109/ICSE43902.2021.00029

Received 2024-01-29; accepted 2024-04-15