# A review on *An Automated Approach to Estimating Code Coverage Measures via Execution Logs*

*Author:*
Mehrad Haghshenas

*Date:*
October 5, 2023

*This is a short review on Chen, B., Song, J., Xu, P., Hu, X., & Jiang, Z. M. (2018, September). An automated approach to estimating code coverage measures via execution logs. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 305-316).*

## Introduction

Tricentis' report indicates that $1.7 trillion financial losses were incurred by software failures; emphasizing a crucial need for increasing the quality of software systems through various methods such as software testing. In software testing, a widely used metric is *code coverage*, which refers to the extent of executed source code. To obtain this metric various tools have been introduced; however, current code coverage tools face limitations. In this article, an innovative solution *LogCoCo* has been suggested, that estimates code coverage from execution logs. We will present three strengths and three areas for potential improvement based on the provided information.

### Strengths

1. **Practicality and Reduced Overhead:** Traditional code coverage tools often involve complex setup, intrusive instrumentation, performance overhead, and incompleteness. LogCoCo, on the other hand, offers an innovative solution to a persistent problem in software testing. Accordingly, LogCoCo leverages readily available execution logs, significantly reducing the engineering challenges associated with configuration and deployment. This approach makes LogCoCo a more feasible choice, especially in large-scale systems.

2. **Wider Application Context:** Traditional tools often struggle to represent real-world scenarios accurately. A key advantage of LogCoCo is its ability to extend code coverage beyond the boundaries of unit and integration testing. Likewise, by utilizating execution logs, LogCoCo can capture the behavior of systems in field-like environments.

3. **Prioritizing & Higher Speed:** LogCoCo offers a fresh perspective on code coverage by utilizing execution logs. It is known that having test cases which cover the methods prone to failure is considered a higher priority in software testing. It is mentioned in the article that more often than not, logging statements are inserted into risky methods. Therefore, LogCoCo does not treat all codes equally and treats risky methods with a higher priority. In addition, LogCoCo can automatically pinpoint the problematic code regions much faster.

Overall, existing state-of-the-art code coverage tools, e.g., Jacoco, excessively instrument the *system under text* and produce the code coverage criteria. The excessive instrumentation guarantees accurate measurements of code coverage. However, problems like deployment challenges and performance overhead are imposed. LogCoCo on the other hand, is easy to setup and imposes little performance overhead by analyzing the execution logs. However, given that developers selectively instrument certain parts of the source code by adding logging statements, the estimated code coverage measures may be inaccurate.

## Weaknesses

1. **Generalizability:** LogCoCo primarily focuses on server-side systems with extensive logging. To enhance its applicability, research studies can explore ways to adapt LogCoCo to mobile applications and client/desktop-based systems with limited or no logging. This expansion would make LogCoCo a more versatile tool. One way to do this expansion can be through focusing on developing automated tools or techniques that *strategically* insert logging statements into source code where there are limited logging available. This technique can also be used to reduce the amount of *May* labels.

2. **Expanding Call Graph Depth:** The methods labeled with Must-not were mentioned to not always be accurate. To address the limitation of call graph depth (capped at 20 levels) due to memory constraints, we can explore more efficient data structures or algorithms for call graph construction. I don't have a concrete idea in this domain, but such a method has been used in model checking; for example in using BDDs (Binary Decision Diagrams) for representing boolean formulas.

3. **Dynamic Analysis for Polymorphism:** Addressing the issue of static analysis limitations related to polymorphism could be addressed by incorporating dynamic analysis techniques. By considering the actual types of objects during runtime, LogCoCo could provide more accurate coverage results in scenarios involving polymorphism.