CATMA: Conformance Analysis Tool For Microservice Applications

Clinton Cao
Delft University of Technology
The Netherlands

Simon Schneider
Hamburg University of Technology
Germany

Nicolás E. Díaz Ferreyra
Hamburg University of Technology
Germany

Sicco Verwer
Delft University of Technology
The Netherlands

Annibale Panichella
Delft University of Technology
The Netherlands

Riccardo Scandariato
Hamburg University of Technology
Germany

ABSTRACT

The microservice architecture allows developers to divide the core functionality of their software system into multiple smaller services. However, this architectural style also makes it harder for them to debug and assess whether the system's deployment conforms to its implementation. We present CATMA, an automated tool that detects non-conformances between the system's deployment and implementation. It automatically visualizes and generates potential interpretations for the detected discrepancies. Our evaluation of CATMA shows promising results in terms of performance and providing useful insights. CATMA is available at https://cyber-analytics.nl/catma.github.io/, and a demonstration video is available at https://youtu.be/WKP1hG-TDKc.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Automated static analysis**; **Dynamic analysis**.

KEYWORDS

microservices, static analysis, dynamic analysis, software testing, empirical software engineering

ACM Reference Format:
Clinton Cao, Simon Schneider, Nicolás E. Díaz Ferreyra, Sicco Verwer, Annibale Panichella, and Riccardo Scandariato. 2024. CATMA: Conformance Analysis Tool For Microservice Applications. In *Proceedings of International Conference on Software Engineering (ICSE'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software systems following the microservice architectural paradigm have their core functionality split into multiple smaller components. These microservices (or just *services*) of a microservice application (MSA) communicate via lightweight communication protocols such as REST APIs or message brokers. The services of an MSA can be developed, maintained, and deployed independently, paving the way for an increasing trend in the adoption of this architectural style. Despite these benefits, MSAs pose a challenge in gaining a comprehensive overview due to their inherently decoupled and distributed nature. Consequently, debugging faults is a time-consuming process because the localization of the root cause is challenging. According to studies, developers usually take several days to debug and find the cause of a fault [7, 15]. Many approaches for the automatic extraction of architectural representations of MSA have been proposed [1, 4, 8, 11], thus addressing the challenge of gaining an overview of the applications' architecture. Some approaches combine static and dynamic analysis to build the architectural models. Also, multiple fault localization techniques for MSAs have been proposed [5, 16], which use dynamic analysis to identify faults and pinpoint the root cause in code. However, to the best of our knowledge, no work compares the results from static and dynamic analysis rather than merging them. Also, none of the existing fault localization approaches offer explainability in the form of possible interpretations for the faults.

In this paper, we present CATMA, a novel tool designed to analyze and compare statically and dynamically obtained architectural models. CATMA autonomously identifies potential non-conformances between these models, generating easily accessible visualizations for users and providing concise interpretations. These interpretations reduce the number of lines in source code that users need to scrutinize when investigating a non-conformance. We tested CATMA on four open-source MSAs and conducted a preliminary usability study with two participants. The results indicate that the tool effectively supports developers during the localization and debugging of non-conformances, demonstrating its usefulness and potential in the debugging landscape for microservices.

2 RUNNING EXAMPLE

The software engineering team of ZYX Inc. is working on their new web application for selling tech products. They embrace the microservice architectural style as this allows them to split up into smaller groups and work independently on the core functionalities of their application. Each member follows the best practices of software engineering; using static analysis to detect faults and testing each functionality before its deployment. After finishing the development, they deploy the application to test it out. To their surprise, they notice that the monitoring service does not receive any metrics data. They are unsure of the cause of this discrepancy since a static analysis tool correctly detects the line of code that implements the transmission of metrics data and does not raise any

ICSE'24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

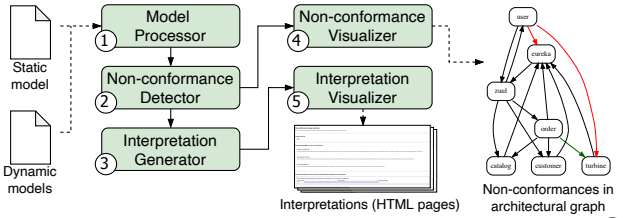

Figure 1: CATMA's workflow. Input models are processed(①) and non-conformances between them detected (②) and visualized (④). Each non-conformance is visualized (⑤) and possible interpretations for it are generated (③).

```
(.venv) (base) ➜  GitHub_catma git:(main) python3 CATMA.py \
> --static_model_path ./data/ewolff_microservice/ewolff_microservice_static_model.json \
> --dynamic_models_path ./data/ewolff_microservice/dynamic_models/ \
> --output_path ./output/
Reading configuration file...
Processing static model...
Processing dynamic model...
Detecting non-conformances...
Detecting non-conformances: 100%|| 13/13 [00:00<00:00, 97892.19it/s]
Detecting non-conformances: 100%|| 13/13 [00:00<00:00, 326502.71it/s]
Detected 2 static non-conformances and 1 dynamic non-conformances between implementation and deploymen
t of system!
Generating non-conformance interpretations...
Generating non-conformance visualizations...
Generating interpretation visualizations...
```

Figure 2: Commandline invocation of CATMA.

warnings. They spend several days analyzing different log files, but have no luck in finding the underlying cause. They scratch their heads and start wondering whether there is a tool that provides:

- detection of discrepancies between the implementation and deployment of MSAs,
- a high-level overview of such discrepancies, and
- descriptions of the potential root causes.

3 CATMA

Worfklow. Figure 1 depicts CATMA's workflow. First, the *Model Processor* ① reads the input models (static and dynamic) to extract architectural components. The obtained data is passed on to the *Non-conformance Detector* ②, which checks whether there are any discrepancies between static and dynamic models. If a non-conformance is detected, it is forwarded to both the *Interpretation Generator* ③ and the *Non-conformance Visualizer* ④. The latter (④) collects all detected non-conformances and generates a visualization of the system's architecture that shows the non-conformances. The former (③) generates a set of possible interpretations for each detected non-conformance, which describe potential causes. These interpretations are forwarded to the *Interpretation Visualizer* ⑤, which generates HTML pages that visualize the interpretations. CATMA is designed to be modular. Each component can be replaced or expanded to fit the user's needs. The tool is invoked via the command line (see Figure 2).

Detecting Non-conformances. As static models, CATMA accepts dataflow diagrams (DFDs) like the ones introduced by Schneider and Scandariato [10]. As dynamic models, state machines inferred from HTTP events logs are expected. They are created using a similar model-inference approach as presented by Cao et al. [2]. The *Model Processor* extracts services and connections between them from both input models. They represent the application's architecture and are used to detect non-conformances. In the DFD, nodes and edges depict the services and information flows between them, respectively. We can, therefore, directly extract the nodes and edges. In a state machine, services and their corresponding relations are represented differently; each transition in a state machine indicates which services in the system have communicated with each other. Thus, nodes and edges are extracted from the transitions of the state machines. The Model Processor creates a set of nodes and edges for both input models, where edges are represented as *"service X → service Y"* and denote the communication relationship between the two services.

Non-conformances are detected by identifying differences between the sets of nodes and edges. The *Non-conformance Detector* iterates through the sets and checks for each item whether it is present in both corresponding sets. We define *static non-conformances* as nodes or edges missing from the static model (compared to the dynamic model) and *dynamic non-conformances* as those missing from the dynamic model. Each item is tagged according to this comparison, i.e., indicating whether it is present in both, only the static, or only the dynamic model. The tagged sets of nodes and edges are passed to components ③ and ④.

The *Non-conformance Visualizer* is responsible for creating a graphical representation of any detected non-conformances. It generates a PlantUML (plantuml.com) file which presents the nodes and edges as a graph and where a coloring scheme highlights any found non-conformances. Model items observed in both models are colored black, items only observed in the static model (dynamic non-conformances) are colored green, and items only observed in the dynamic model (static non-conformances) are colored red.

Interpreting Non-conformances. CATMA generates a set of possible interpretations for each detected non-conformance. These interpretations are visualized in an HTML page by the *Interpretation Visualizer*. The HTML page helps users analyze the potential causes of non-conformances. CATMA presents a specific set of interpretations for both types of non-conformance. The generated HTML pages contain (1) the type, definition, and involved services of the non-conformance, (2) the set of possible interpretations, and (3) additional details that support the understanding of the non-conformance. In the following, (2) and (3) are described further.

Providing Interpretations of Non-Conformances. A set of high-level textual interpretations is provided, which describe possible underlying causes of the detected non-conformances. The interpretations are meant to serve as possible starting points to debug found non-conformances. Currently, the generation is based solely on the type of non-conformance, i.e., whether it is static or dynamic. We formulated a text describing possible interpretations for both types of non-conformances, and the corresponding one is presented to the user. As the basis for these interpretations, we collected known causes of non-conformances from the literature (e.g., [6, 13]). These causes range from standard programming errors made in software development to common causes for issues encountered by developers of MSAs. As an example, misconfiguration of services is a common cause of dynamic non-conformances in MSAs. When services are not properly configured, they become undiscoverable by other services, leading to missing expected runtime behaviors. CATMA uses this information as a basis for the generation of one interpretation for a dynamic non-conformance. For the collection, we disregarded non-conformances rooted in hardware-related issues, e.g., due to non-deterministic behavior because of multi-threading or similar effects. The textual descriptions of possible interpretations provided for a static non-conformance are shown in Figure 3. Our future work will predominantly focus on this part of the tool,

Potential Interpretations For the Non-Conformance

Implicit Call via Third-Party Services
Communication flow has been detected between the two services but the responsible line of code is not detected in the source code. It could be that the call is implicitly triggered by a third-party service that is used in the implementation (e.g. via an annotation that is used by a framework or that code is injected during run-time).

Unintentional Endpoint Exposure
The communication flow detected between the two services could be caused by an endpoint that is exposed unintentionally by the developer. It could be the case that code has been refactored and the endpoint was not removed during refactoring, or that the endpoint was used for testing purposes and was not removed afterwards.

Code Located Outside of Default Source Location
The line of code that is responsible for triggering the flow of communication between the two services is not located in the default source-code folder of the project. It could be the case that the line of code was unintentionaly introduced in a different folder (e.g. resources folder).

Figure 3: Example set of textual interpretations.

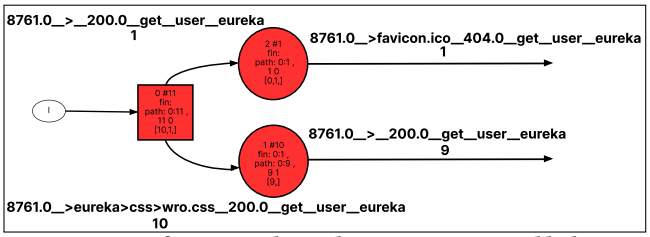

Figure 4: Part of state machine showing unexpected behavior.

specifically on implementing a more intelligent generation of applicable interpretations. In this regard, we will analyze indicators for each cause of non-conformances. These indicators will then be used to decide whether a cause is plausible or not for a given non-conformance. This will lead to the generation of a tailored set of possible interpretations for each found non-conformance. The already carried-out analysis of the related literature provides the basis for this future work.

Additional details. The generated HTML page also presents additional details that could aid the user with the understanding of the detected non-conformance. In the case of static non-conformances, a state machine is visualized that depicts the unexpected sequential communication behavior detected between the involved services. The most frequently occurring calls between the involved services are presented in a human-readable format right after the state machine model. This insight can be used to understand why such calls were made between the involved services. Figures 4 and 5 show an example of a state machine and the most frequent calls, respectively. In the case of a dynamic non-conformance, we instead leverage the traceability information contained in the static model to point to the code that shows the expected behavior. Specifically, the page presents (1) the line of source code responsible for triggering the missing runtime event, (2) the sequence of events that should trigger the missing runtime event, and (3) human-readable call details extracted for the previous point. Figure 6 provides a snapshot of this set of details. Furthermore, the state machines learned for each involved service are presented on the HTML page.

4 TOOL EVALUATION

Performance Analysis. We evaluated CATMA's performance in terms of time to detect non-conformances in MSA. For this, we selected 4 DFDs of open-source MSAs from the dataset created by Schneider et al. [9], deployed these MSAs, and created state machine models for them. Then, we ran CATMA on the obtained models and measured the time of the analysis. Table 1 presents the time for analysing the 4 selected MSAs (averaged over 10 executions

Frequently occurring endpoint calls extracted from the dynamic model learned for the link between user and eureka:

Endpoint: /
- Port: 8761.0
- Call status code: 200.0
- Call direction: from user to eureka
- Call frequency: 38

Endpoint: /eureka/css/wro.css
- Port: 8761.0
- Call status code: 200.0
- Call direction: from user to eureka
- Call frequency: 10

Figure 5: Most frequent calls for unexpected behavior.

Sequences that occurred in the dynamic model that should produce run-time behaviour for link between order and turbine

- user →(implicit) zuul → order → turbine
- zuul → order → turbine

For the occurred sequences, these are the unique sequence of endpoints (parameters) that were used in the sequence

- Sequence: user → zuul → order → turbine
  - Call started with "/order/line". Then followed by call with "/line".
  - Call started with "/order/". Then followed by call with "/".
  - Call started with "/order/18". Then followed by call with "/18".
  - Call started with "/order/11". Then followed by call with "/11".
- Sequence: zuul → order → turbine
  - Call started with "/line".
  - Call started with "/18".
  - Call started with "/3".
  - Call started with "/5".
  - Call started with "/11".
  - Call started with "/form.html".
  - Call started with "/".

Figure 6: Example details for dynamic non-conformance.

Table 1: CATMA's performance statistics on multiple MSAs

Name	#LOC	# Services	# Detected Non-Conformances (static / dynamic)	Avg. Runtime (seconds)
Springboot-Microservice[1]	879	9	0 / 16	4.3
Microservice Sample[2]	3117	7	2 / 1	3.0
Spring PetClinic[3]	3990	12	1 / 26	78.6
Piggy Metrics[4]	9977	17	3 / 11	53.9

per MSA). This evaluation allows us to quantify the benefits of utilizing CATMA compared to manual analysis. The data clearly demonstrates that CATMA significantly accelerates the analysis process. While developers often invest days in resolving issues (as reported in [15]), our tool accomplishes the same task in a matter of minutes. Thus, CATMA can substantially reduce the time spent on debugging issues, offering a valuable resource for developers.

User Study. We conducted a small-scale user study to investigate CATMA's usefulness. We report an initial assessment of this user study based on a think-aloud interview setup with two participants. The participants were recruited from the lab of one of the authors and have no relation to the work done for CATMA. The participants got an introduction to MSAs and were allowed to interact with CATMA before the start of the interview. During the interview, we asked several questions that would provide us insights on what are the most useful elements presented in the output generated by CATMA. A complete transcript of the interview can be found at [3]. The following points summarize the most useful elements from CATMA's output: (1) the model-based visualization that shows where non-conformances are detected, (2) the set of possible interpretations providing the potential causes for the corresponding non-conformance, (3) the ability to jump from

[1]https://github.com/shabbirdwd53/springboot-microservice
[2]https://github.com/ewolff/microservice
[3]https://github.com/spring-petclinic/spring-petclinic-microservices
[4]https://github.com/sqshq/piggymetrics

Table 2: Trade-off between the size and correctness.

Avg. # Edges	Avg. # Nodes	Avg. Recall	Avg. Specificity	Avg. Balanced Accuracy
1	1	0.0	1.0	0.5
127	99	0.368	0.998	0.683
790	646	0.904	0.990	0.947
1982	1843	0.920	0.986	0.953
4714	4570	1.0	0.978	0.989

the dynamic model (state machine) back to the source code, (4) static non-conformances provide insights on the security implication of the system, and (5) the type of the non-conformances: static non-conformances provide insights on the security implication.

Correctness of Dynamic Models. As the state machines approximate the provided log data, it is helpful to understand the trade-off between the correctness and the size of the model as it could influence the detection of non-conformances; a small state machine generalizes too much and introduces inaccuracies, a large state machine captures all possible behavior but might be hard to understand and process. To evaluate this aspect, we use a technique similar to the one proposed by Walkinshaw et al. [12]. Table 2 presents the average results computed from a 10-fold cross-validation experiment. As expected, we see that smaller state machines introduce more inaccuracies, leading to lower balanced accuracy scores. This suggests that smaller state machines do lead to more inaccuracies in the detection of non-conformances. Furthermore, we see that the accuracy scores do not significantly increase when we learn a much larger state machine. This suggests that large state machines do not introduce significantly more inaccuracies and there is no need to opt for the largest model for the detection of non-conformances.

5 RELATED WORK

Several approaches in related literature combine static and dynamic analysis for architecture reconstruction of MSAs. *MicroArt* presented by Granchelli et al.[4], *MiSAR* presented by Alshuqayran et al. [1], and *μTOSCA* presented by Soldani et al. [11] all extract the list of microservices statically by parsing deployment files. Connections between them are detected dynamically by leveraging service discovery services that exist in the analyzed applications or by injecting different monitoring tools. *VMAWV* presented by Ma et al. [8] instead queries existing service discovery services to retrieve the list of services and uses static analysis to detect connections. While these approaches combine static and dynamic analysis, none of them compare complete architectural models obtained via the two techniques. Since our approach performs this comparison to identify non-conformances, we believe it to be novel in this regard.

The approach *DOMICO* by Zhong et al. [14] also detects non-conformances between system representations of different stages in the development process, however, they compare the intended design (UML) against the actual implementation (static model).

6 CONCLUSION & FUTURE WORK

We present CATMA, a tool for automatically conducting conformance analysis of MSAs. It detects possible non-conformances by computing differences between a statically and a dynamically obtained architectural model of the MSA. Found non-conformances are visualized in an easily accessible way. Further, a set of possible interpretations is generated, showing the non-conformances' potential causes. In a preliminary evaluation, CATMA showed promising results in terms of performance as well as usability. In our evaluation, CATMA identified a non-conformance in an open-source MSA on GitHub. A misconfiguration in the Hystrix [5] monitoring dashboard prevented stream data from being visualized as intended in the implementation. This is a good example of a non-conformance between the intended and observed behaviors of the MSA. We notified the developers and our fix was accepted [6]. Hence, CATMA has already shown its first –albeit small– impact on MSA.

As future work, we will extend CATMA with a more intelligent technique for selecting suitable interpretations for found non-conformances. Further, the approach would benefit from further validation activities concerning its usefulness and possible enhancements. We plan an extended user study with developers in which they identify non-conformances with the help of CATMA. Finally, we will investigate the feasibility of using other types of models as input and the detection capabilities of other non-conformances.

ACKNOWLEDGMENTS

This work was partly funded by the European Union's Horizon 2020 program under grant agreement No. 952647 (AssureMOSS).

REFERENCES

[1] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2018. Towards Micro Service Architecture Recovery: An Empirical Study. In *ICSA*.
[2] Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. 2022. Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster. In *ARES*.
[3] Clinton Cao, Simon Schneider, Nicolás Diaz Ferreyra, Sicco Verwer, Annibale Panichella, and Riccardo Scandariato. 2023. Appendix for 'CATMA: Conformance Analysis Tool for Microservice Applications'. https://doi.org/10.6084/m9.figshare.23942214.v2
[4] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. 2017. MicroART: A software architecture recovery tool for maintaining microservice-based systems. In *ICSAW 2017*.
[5] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *ESEC/FSE*.
[6] Abdelhakim Hannousse and Salima Yahiouche. 2021. Securing microservices and microservice architectures: A systematic mapping study. *Comput. Sci. Rev.*
[7] Valentina Lenarduzzi and Annibale Panichella. 2021. Serverless Testing: Tool Vendors' and Experts' Points of View. *IEEE Software*.
[8] Shang Pin Ma, I. Hsiu Liu, Chun Yu Chen, Jiun Ting Lin, and Nien Lin Hsueh. 2019. Version-Based Microservice Analysis, Monitoring, and Visualization. *APSEC*.
[9] Simon Schneider, Tufan Özen, Michael Chen, and Riccardo Scandariato. 2023. microSecEnD: A Dataset of Security-Enriched Dataflow Diagrams for Microservice Applications. In *MSR*.
[10] Simon Schneider and Riccardo Scandariato. 2023. Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java. *JSS*.
[11] Jacopo Soldani, Giuseppe Muntoni, Davide Neri, and Antonio Brogi. 2021. The mTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*.
[12] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *EMSE*.
[13] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Aakash Ahmad, and Ali Rezaei Nassab. 2021. On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study. In *EASE*.
[14] Chenxing Zhong, He Zhang, Huang Huang, Zhikun Chen, Chao Li, Xiaodong Liu, and Shanshan Li. 2023. DOMICO: Checking conformance between domain models and implementations. *Software: Practice and Experience* (2023).
[15] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *TSE*.
[16] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *ESEC/FSE*.

[5] https://github.com/Netflix/Hystrix
[6] https://github.com/ewolff/microservice/pull/30