Towards Log-driven Testing through Transformers: A Preliminary Study

Raffaele Della Corte[a,b], Roberto Pietrantuono[a,b]
[a]*DIETI - Università degli Studi di Napoli Federico II*
Via Claudio 21, 80125, Naples, Italy
[b]*CINI - Consorzio Interuniversitario Nazionale per l'Informatica*
M.S. Angelo, Via Cinthia, 80126 Naples, Italy
{raffaele.dellacorte2, roberto.pietrantuono}@unina.it

***Abstract*—Software testing is one of the most important engineering tasks during the software development life cycle, since it supports the identification of potential issues in the code and allows reducing the cost of maintenance. This is especially true in regression testing, where the system is tested with the aim of identifying regressions generated in new software versions. However, regression test suites are usually developed with no knowledge about how the system will be used in the field and, in turn, they may fail in highlighting interesting testing scenarios, potentially leading to the deployment of a faulty software version. When an interesting scenario takes place, e.g., a system failure due to some regressions, understanding how to induce similar system behaviors for testing purposes may be a hard task, requiring time-consuming analysis of event logs, system traces, and system documentation.**

In this paper, we propose an approach to automatically generate sequences of service invocations in service-based software inducing system behaviors similar to the ones observed in the field. The approach leverages transformers to learn from both event logs and system traces how to generate a sequence of service invocations exposing behaviors similar to those of interest observed in the field. Generated sequences can be used by practitioners to design/augment regression test suites. A preliminary implementation of the approach has been used on a publicly available dataset, encompassing both event logs and system traces of a microservices-based system.

***Index Terms*—logging, tracing, transformers, testing**

I. Introduction

Today's software development features frequent updates and short release cycles to be able to adapt and quickly respond to changes of the environment and of requirements. Service-based software, particularly microservices applications developed in a DevOps style, are a prominent example of these commonly adopted paradigms.

Regression testing is a key activity for such systems, since it continuously checks for possible breakages introduced over subsequent releases. A large body of literature has been devoted to regression testing [1], and more recently in continuous integration (hence DevOps-like) environments, especially addressing Test Selection & Prioritization (TS&P) from an existing test suite [2]–[4]. The use of Machine Learning (ML) has significantly spread also in this area, with strategies exploiting historical data to train models relating test cases to their potential outcome (e.g., to select/prioritize likely-to-fail tests) [5]–[7].

However, often times such data exploited to select/prioritize or also to generate new interesting test cases is taken from static sources, such as test and/or code metrics from past tests execution extracted from versioning systems like `Git`. Although good results have been obtained, a large amount of information coming from the field, which is often immediately available in modern agile practices (e.g., continuous integration/deploy, DevOps) through continuous monitoring that is part of the process, is mostly neglected and not adequately exploited. We claim that the behavior observed in the field is much more informative than information gathered at development time, as it encompasses the interaction with the end user and with the real environment not available in-house. While field data are used for tasks like field failure and root cause analysis or for performance optimization, their exploitation for testing is still rather limited.

In this work, we preliminarily define an *ex-vivo* testing strategy, aimed to explore if and how we can use operation-time information gathered through logs to extrapolate behaviors of interests of the system (represented by features), and then use it to augment the in-house regression test suite. The proposed strategy aims to first learn the association (i.e., the joint distribution) between the observed behavior of interests for tester (encoded through features extracted from logs, e.g., number of occurred 500 http status code) and the sequences of services invocation. We leverage a pre-trained generative Large Language Model (LLM), fine-tuned to our task, to learn this association. Then, in the inference phase, the model is queried to generate new sequences of invocation given the features extracted from a log of interest as input. For instance, testers may require to generate sequences producing (logs with) a high number of 500 http status code, or high latency values.

The paper reports about preliminary results, by which we show that the newly generated sequences, given an observed log with features we want to reproduce (i.e., given a "label"), often corresponds to sequences in the test set whose logs are very similar to the one given as input.

II. Related Work

Regression testing has been investigated for long time [1] [8]. Techniques for regression testing aim to select (and/or prioritize) which of the existing test cases need to be re-executed after a change. The aim is to check if changes (e.g., new releases) introduced bugs in the code. With the spread of agile practices, e.g., Continuous Integration (CI) and DevOps, the importance of regression testing increased, as it is the main means to continuously check the code during its evolution.

Research on test selection and prioritization in CI environments is very active. Various techniques have been proposed to efficiently identify those test cases that exercise the changed code [2], [3]. Machine Learning has been used too, with the aim of learning associations between failing tests (or potentially failing code/commits) and information potentially correlated with it (e.g., coverage). For instance, Pang *et al.* [5] demonstrated that simple unsupervised learning algorithms, such as *k*-means, based on coverage information, can be used with good results. However, since code coverage collection is costly at large scale [9], researchers also proposed *static* approaches that use program analysis to identify the code parts potentially affected by a change, so as to select only tests exercising that part. Cruciani *et al.* [10] proposed a black-box similarity-based approach to test reduction that can scale up to very large test suites by using big data techniques.

Besides test selection, several strategies exist for regression tests prioritization. The aim is to execute first the most likely failing tests to spot bugs earlier [11]. The use of ML has also gain spread for test prioritization in the last years [12]. Tonella *et al.* [13] first applied a Case-Based Ranking algorithm to rank tests according to coverage, complexity metrics and historical data. Lenz *et al.* [6] proposed a ML strategy that leverages test-related data to support various tasks, including test prioritization. Busjaeger and Xie [4] reduce the problem of test prioritization to that of learning-to-rank (LTR): their model learns from a training set made up by past changes and by the tests observed for each of them. Lachman *et al.* [14] apply the LTR SVM-Rank algorithm to black-box prioritization starting from test cases and failure reports. Spieker *et al.* [15] propose the first TS&P approach using reinforcement learning with a shallow neural network agent.

Contrarily to this body of work, our strategy differs in two aspects: *i)* rather than features collected over past test execution or test artifacts, code changes or failure history, we claim that the information collected from the operational phase should be exploited to derive tests whose outcome better predicts what will be observed in the field over successive releases. This is today possible thanks to the continuous monitoring and feedback available in modern development processes (like DevOps) and architectures (e.g., in microservice applications). *ii)* although we can use the proposed strategy to select/prioritize existing tests, our main aim is to augment the existing regression test suite, enriching it with new field-aware test cases. With a specular strategy, the model we train could be used to exclude tests (hence reduce test suites).

III. Proposal

An overview of the proposal is shown in Fig. 1. The strategy requires two types of input: *event logs* and *system trace data*. **Event logs** are a byproduct of the software system execution, since they are naturally emitted during operation. Event logs are sequences of text lines – typically stored in log files – reporting about the runtime behavior of a system [16], including entries highlighting the occurrence of system failures. **System trace data** accounts for various information about the use of services composing a system, allowing to track source-destination of the services invocation, along with the response codes, latency, etc. Our approach takes advantage of both data sources in order to learn the association between service invocation chains $\mathcal{X}$, embedded via n features $\{x_1, x_2, \ldots, x_n\} \in \mathcal{X}$, and produced logs $\mathcal{Y}$, described by m features $\{y_1, y_2, \ldots, y_m\} \in \mathcal{Y}$ (thus, we learn the joint probability distribution $P(\mathcal{X}, \mathcal{Y})$) via a generative model. Once we have the model, we query it to generate new sequences of service invocations X' given the label, namely given a subset of k features of interest of a log Y: $Y' = \{y'_1, y'_2, \ldots, y'_k\} \in \mathcal{Y}$, representing a behavior of interest that we want to approximately reproduce, e.g., logs with services failures or with high latency. In other words, we ask for $P(\mathcal{X}'|\mathcal{Y}')$. A pre-processing stage is performed to map each trace, representing a particular system invocation, with the event log entries generated during that invocation. One of the obtained datasets (i.e., Training set in Fig. 1) is then used to fine-tune a *transformer*-based large language model (LLM), pre-trained on English, for the task of generating sequence of service invocations from event logs.

Pre-processing. Given a set of logs and traces, pre-processing aims to map each trace $\mathcal{T}_i$ (identified by a trace ID i) with the log generated during the invocations captured by that trace. To this aim, this stage (i) extracts the time window of each trace (i.e., the time difference between the last and the first event of the trace) as well as the *sequence of services* involved in the invocation represented by the trace, (2) groups together the log entries that fall in the extracted time window (hereinafter *raw log*), (3) pareses the raw log entries in order to extract their templates and parameters, and (4) labels the raw log entries with the extracted services sequence. After these steps, we have a dataset including, for each trace ID, *i)* the services sequence involved in the trace – denote as $\mathcal{X}$; *ii)* the raw log entries mapped with the trace, the templates and parameters associated with the raw log, as well other information extracted from the trace and logs, i.e., latency (in seconds) of the entire interaction described by the trace, the list of endpoints involved in the trace, the amount of log entries for each log severity level (e.g., INFO, WARNING, ERROR) and the number of `http` response codes (e.g., 200, 300, 500) – all these features being denoted as $\mathcal{Y}$. An example of entry of the dataset is shown in Fig. 2[1]. The extraction of raw log templates and parameters is performed leveraging the DRAIN tool [17], which allows to automatically identify the

[1]Some fields are truncated in the figure due to space limitations.

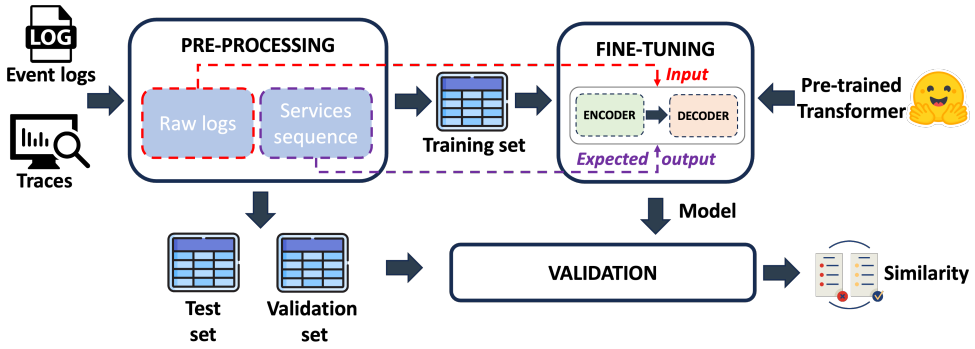

Fig. 1. Proposed strategy.


```
"trace_id":"c124e30fb40651dc",
"involved_services":"webservice1--redisservice1--mobservice2--
redisservice2--redisservice2--logservice2--redisservice2--logservice1--
dbservice1--dbservice1--redisservice1",
"involved_endpoints":"http://0.0.0.1:9379/web_login_service--
http://0.0.0.1:9386/set_key_value_into_redis?keys=a3036736-da17-11eb-
9811-0242ac110003&value=--...{TRUNCATED}",
"latency_sec":1.284347,
"raw_logs":"2021-07-01 10:54:21,907 | INFO | 0.0.0.1 | 172.17.0.3 |
webservice1 | c124e30fb40651dc | the list of all available services are
redisservice1: http://0.0.0.1:9386, redisservice2: http://0.0.0.2:9387\n
...{TRUNCATED}",
"logs_templates":"<:NUM:>-<:NUM:>-<:NUM:> <:NUM:>:<:NUM:>:<:NUM:>,<:NUM:>
| <:*:> | <:IP:> | <:IP:> | webservice1 | <:*:> <:*:> <:*:> <:*:> <:*:>
<:*:> <:*:> <:*:> <:*:> <:*:> <:*:> <:*:> <:*:>\n ...{TRUNCATED}",
"logs_templates_parameters":"['2021', '07', '01', '10', '54', '21',
'907', 'INFO', '0.0.0.1', '172.17.0.3', 'c124e30fb40651dc', '|', 'the',
'list', 'of', 'all', 'available', 'services', 'are', 'redisservice1:',
'http:\/\/0.0.0.1:9386,', 'redisservice2:', 'http:\/\/0.0.0.2:9387']\n
...{TRUNCATED}",
"200":10,
"500":0,
"300":1,
"WARNING":0,
"INFO":7,
"ERROR":0
```


Fig. 2. Example of entry in the dataset.

variable fields of logs and substitute them with placeholders, potentially identifying log templates and related parameters (i.e., the values assumed by the variable fields). Finally, the dataset is split into Training (10%), Test (10%) and Validation (80%) sets, with the first one used for fine-tuning of our model, and the remaining ones for the validation stage.

Transformers-based LLM. The model we use is T5-small, a reduced version of T5, a transformer-based sequence-to-sequence model. T5-small is an encoder-decoder model architecture, characterized by six layers in each encoder and decoder, eight attention heads, and by about 60 million parameters. T5 models each problem into a text-to-text format, and it is pre-trained on the colossal clean crawled corpus (C4) dataset (consisting of hundreds of gigabytes of clean English text scraped from the web), as well as on a mixture of unsupervised and supervised tasks, such as natural language inference, question answering, etc. The model works by receiving a text input containing the desired task to be performed as the prefix, and it produces an output in text format.

Fine-tuning. T5-small can be fine-tuned for different tasks like summarization, classification, and translation. In this stage we fine-tuned the pre-trained model (obtained through the Hugging Face transformer library) on the task of generating sequence of service invocations from raw logs. This task can be modeled as a *translation* task, where the input is represented by the raw logs, i.e., the interesting log entries observed in operation, while the expected output is the services sequence, i.e., the sequence of invocations generating that log entries, which can be potentially leveraged to reproduce similar system behaviors with respect to the ones reported in the logs.

We performed the fine-tuning training leveraging the Training set generated during the pre-processing stage. The Training set is further split in 90% for training and 10% for testing, with the latter used to evaluate the model performance after each epoch of training. We trained our model for overall 18 epochs, with a 0.00001 learning rate and 0.01 weight_decay, while the *Levenshtein distance* is used to evaluate the model performance after each epoch. In our context, this metric allows evaluating the distance between two services sequences, i.e., compare the actual services sequence (the label) and the generated one.[2]

IV. Preliminary Results

In order to validate the proposed strategy, we leverage the GAIA dataset [18], which encompasses both event logs and system trace data collected during the operation of MicroSS, a microservice-based business simulation system. The dataset accounts for around 7M log entries and more than 3M traces (with overall 28M trace events). Following our proposal, we apply the pre-processing stage to the GAIA dataset, obtaining a pre-processed dataset with 1M entries (like the one shown in Fig. 2). In this preliminary study we considered only a subset of the pre-processed dataset, which encompasses 116k randomly selected entries. Those entries are then split in Training (8k entries), Test (8k entries) and Validation (100k entries) sets. The fine-tuning stage is applied using the Training set (as described in Section III), obtaining the T5-small transformer tuned for our translation task, i.e., generate services sequence from raw logs. The generated Test and Validation sets, along with the fine-tuned transformer, are then used for the validation stage. The idea underlying this stage is to verify if our model is able to generate service sequences exposing behaviors similar to those observed in the raw logs provided as input. We leverage the raw logs in the Validation set (VS) to perform the comparison, selecting the entries that better match the services sequence generated by our model.

Fig. 3 depicts the validation stage, where each entry of the Test set (TS) is analyzed via three main steps: (i) given an entry of the Test set, its raw logs field (Raw logs TS in Fig. 3) is submitted to the fine-tuned transformer, which generates the related services sequence; (ii) the obtained sequence is

[2]To directly apply the metric, we replaced the service names with single characters, e.g., the sequences `webservice-redisservice-mobservice` and `webservice-dbservice-mobservice` become `A-B-C` and `A-D-C`, with a Levenshtein distance equal to 1.

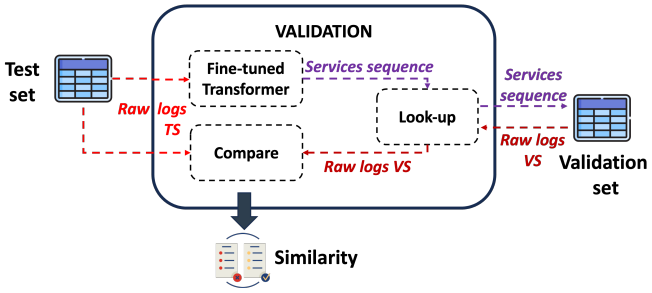

Fig. 3. Validation stage.

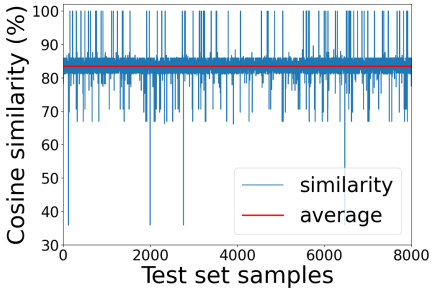

Fig. 4. Cosine similarity results.

used for a look-up operation into the Validation set, which aims to find an entry having the same services sequence (or the most similar one according to the Levenshtein distance) and retrieve its raw logs field (Raw logs VS in Fig. 3); (iii) the Raw logs TS and Raw logs VS are then compared to obtain an estimation of their similarity, through the *cosine similarity*.

Given two logs to compare, the comparison step first converts them into term-frequency vectors, each one containing the frequency of the terms composing the log. Let A and B denoting the two vectors, their cosine similarity is defined as $(A \bullet B)/(||A|| * ||B||)$, where the numerator and denominator represent the dot product and the Euclidean norm of A and B, respectively. The obtained value is in the [0, 1] range, with 0 indicating totally different logs and 1 that logs are identical.

Fig. 4 depicts the cosine similarity (in percentage terms) for all the samples contained in the Test set, along with the average similarity. It can be noted that most of the similarity values fall between 80% and 90% (with some spikes under 40%), as highlighted by the 83.54% average value. This represents a promising result for the proposal since the raw logs obtained from the Validation set by using the service sequences generated by our model are not so far from the one used as input for the model. Therefore, the sequences of service invocations generated by our model can be leveraged to reproduce behaviors similar to ones of interest observed in the field, and then used to design/augment regression test suites for the target system.

V. Conclusion

The paper proposed an ex-vivo testing strategy to explore how to leverage information gathered from event logs to generate interesting system behaviors through a pre-trained generative LLM. The strategy has been applied on a publicly available dataset drawn from a service-based software execution, exhibiting promising results. The next step is to validate the strategy on a new target service-based system, for which we need to repeat the above steps and then exercise the system with the generated services sequence, collect the logs, and compare with the ones given as input. Future work includes the enhancement of the model via a further pre-training stage on a corpus of logs before fine-tuning.

Acknowledgment

This work was supported in part by the European Union's Horizon 2020 program under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS", and by the SERENA-IIoT project funded by Ministero dell'Università e della Ricerca and European Union (Next Generation EU) under the PRIN 2022 program (project code 2022CN4EBH).

References

[1] S Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[2] Eric Knauss et al. Supporting continuous integration by code-churn based test selection. In *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, RCoSE, pages 19–25. IEEE, 2015.

[3] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *2015 International Symposium on Software Testing and Analysis*, ISSTA, pages 211–222, New York, NY, 2015. ACM.

[4] B. Busjaeger and T. Xie. Learning for test prioritization: An industrial case study. In *24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, FSE, pages 975–980, New York, NY, 2016. ACM.

[5] Y. Pang, X. Xue, and A. S. Namin. Identifying Effective Test Cases through K-Means Clustering for Enhancing Regression Testing. In *12th International Conference on Machine Learning and Applications*, pages 78–83. IEEE, 2013.

[6] A. R. Lenz, A. Pozo, and S. R. Vergilio. Linking software testing results with a machine learning approach. *Engineering Applications of Artificial Intelligence*, 26(5):1631–1640, 2013.

[7] A. Bertolino et al. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1–12, New York, NY, USA, 2020. ACM.

[8] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, apr 2001.

[9] S. Elbaum, G. Rothermel, and J. Penix. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 235–245, New York, NY, 2014. ACM.

[10] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino. Scalable approaches for test suite reduction. In *41st International Conference on Software Engineering*, ICSE, pages 419–429. IEEE, 2019.

[11] M. Khatibsyarbini et al. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018.

[12] V. H. S. Durelli et al. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 68(3):1189–1212, 2019.

[13] P. Tonella, P. Avesani, and A. Susi. Using the Case-Based Ranking Methodology for Test Case Prioritization. In *22nd IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2006.

[14] R. Lachmann et al. System-level test case prioritization using machine learning. In *15th IEEE International Conference on Machine Learning and Applications*, pages 361–368. IEEE, 2016.

[15] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *26th ACM SIGSOFT Int'l Symposium on Software Testing and Analysis (ISSTA)*, pages 12–22, New York, NY, 2017. ACM.

[16] M. Cinque, R. Della Corte, and A. Pecchia. An empirical analysis of error propagation in critical software systems. *Empirical Software Engineering*, 25(4):2450–2484, 2020.

[17] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.

[18] CloudWise. GAIA dataset. https://github.com/CloudWise-OpenSource/GAIA-DataSet.