

An Architecture for a Reliability Tool Applied to Distributed Systems

Caio Guimaraes Herrera[1(✉)], Vitor Silva Montes[1], Simone R. S. Souza[1], Ricardo Ribeiro dos Santos[2], and Paulo Sergio Lopes de Souza[1]

[1] University of Sao Paulo (ICMC/USP), Sao Carlos, SP, Brazil
caioherrera@usp.br
[2] Federal University of Mato Grosso do Sul (UFMS), Campo Grande, MS, Brazil

Abstract. Contemporary software systems, such as IoT (Internet of Things) systems, present new challenges to the development process and require new strategies for quality assurance. Reliability engineering aims to detect faults in applications, often in productive environments, providing benefits to the development activity. However, due to the inherent characteristics of these systems, complementary approaches to detect faults beyond traditional testing are needed. The Tricorder methodology evaluates resource consumption profiles as a heuristic for quality assurance, and experimental studies indicate its effectiveness in detecting new faults. This work investigates the use of the Tricorder in the context of reliability engineering to support the validation of contemporary systems, particularly IoT systems. An architecture for a reliability tool is proposed, inspired by software testing concepts. A reference architecture, RefTEST, is used to support the proposed tool. Results indicate that the Tricorder methodology can be implemented as a reliability tool, and its adoption allows for identifying anomalies that are not easily detected by traditional testing methods.

Keywords: Software reliability · distributed systems · fault forecasting · internet of things

1 Introduction

Software development is constantly evolving and growing in complexity. With the distributed nature of modern systems, DevOps practices [11] and the field of release engineering [5] have become crucial in ensuring the longevity and reliability of software. Moreover, the discipline of reliability engineering includes strategies for continuously monitoring and testing systems, even after reaching production environments [17], as software reliability is one of the main factors affecting software quality [10].

Well-structured tests are essential to detect faults in the early stages. However, some characteristics associated with low testability software [20] can hinder fault detection before the software reaches a production environment, creating an opportunity for complementary approaches to traditional software testing.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
O. Gervasi et al. (Eds.): ICCSA 2025, LNCS 15649, pp. 332–348, 2025.
https://doi.org/10.1007/978-3-031-96997-3_21

When considering low-testability software, continuous testing also presents new challenges. Data available in testing environments might not fully represent what can be found in production [19]. Acquiring and reproducing input data from production environments might be limited for security reasons [19]. These challenges highlight the need for testing strategies that differ from conventional approaches.

The Tricorder methodology [12] is a complementary approach that does not rely on input and output data to evaluate a System Under Test (SUT) but rather on defining resource consumption profiles. By detecting the changes in the consumption profiles, it is inferred that the SUT has an anomaly, which could be related to a fault being inserted in the code. The methodology can identify an anomaly automatically without the tester's mediation, although the latter is responsible for investigating the detected anomaly to see if it correlates to an actual fault. Tricorder has been evaluated in a set of studies and applications [1, 4, 12]. These studies indicate that the methodology effectively detects artificially inserted faults. Currently, Tricorder works using scripts that are modified to adapt to new systems when they are evaluated.

The Tricorder methodology is in line with the fault lifecycle techniques of software reliability engineering: **1)** fault prevention, **2)** fault removal, **3)** fault tolerance, and **4)** fault/failure forecasting [10]. In this context, our proposal can be classified as **fault/failure forecasting**.

A software tool must be available to implement Tricoder. Consequently, there is a need to establish a software architecture that can account for the Tricorder features while also allowing its use as a reliability tool. Moreover, this architecture should also allow for its use in low testability applications, such as IoT (Internet of Things) systems.

A reference architecture for developing software testing tools, such as RefTEST [13], can guide this process. Since the Tricorder methodology follows a non-conventional approach for evaluating the SUT, as it does not rely on specific features of the SUT implementation, nor does it need to characterize test cases based on input and output data, adapting the Tricorder methodology to RefTEST is a challenge.

This paper presents an architecture proposal, based on RefTEST, for a tool that implements the Tricorder methodology to evaluate software reliability. This work also discusses how flexible the reference architecture is and how we can design tools that follow a non-conventional approach. Moreover, this paper first attempts to adapt the Tricorder methodology into a tool while considering its use in a low testability scenario, exemplified by an IoT systems model.

The remainder of this text is organized as follows: Section 2 presents the context and motivation of this study. Section 3 introduces the Tricorder methodology. Section 4 covers the related work regarding the architectures of software testing tools. Section 5 shows the proposed architecture. Finally, Sect. 6 highlights and discusses the main results of this paper, and Sect. 7 concludes this study with final remarks and future work.

2 Background

The evolution of software development environments and the continuous increase in software complexity have made modern software ecosystems multidisciplinary. Beyond the programmer role, it is necessary to have testers, QA (Quality Assurance) analysts, and product/project managers for the development cycle. DevOps practices [11], such as CI/CD (Continuous Integration/Continuous Deployment) and version control tools, are essential to ensure the quality of the software being deployed to production.

The discipline of release engineering [5] includes these concepts, adding concerns regarding software deployment, longevity, and reliability to the traditional software engineering activity to ensure that the developed systems are predictable and reliable. More specifically, reliability engineering [17] ensures the availability, security, and integrity of computer systems, even after reaching production environments. To do so, routines such as monitoring, testing, and validation need to keep happening after a software is deployed [17].

It is still equally important to have well-structured test routines to evaluate changes and new software versions. From unit and integration tests up to system and acceptance tests, a set of test cases can be carried out from the first steps of developing a new feature or product. However, some characteristics of the software being developed, or the context in which it belongs, can hinder the detection of faults at early stages. In these scenarios, the software testability concept is crucial and needs to be considered.

Software testability can be described as the probability that software will fail at the subsequent test execution, provided it has a fault [20]. Low testability software can accumulate many characteristics that offer a challenge for the early detection of faults. Some of them that are of interest to this study are:

(1) **Data unpredictability**: several application contexts, such as IoT systems, rely almost exclusively on input data from sensors in contact with a real environment, exposed to noise and hardware failures, which can cause a divergence between data used in controlled environments for development and testing, and production data;
(2) **Low to no human interaction**: systems built to run in embedded systems are commonly set to run on areas with low connectivity and almost zero interaction with humans. This context can make fault detection take longer than in a scenario of high human interaction; and
(3) **Input and output data as stream**: applications with this characteristic rely on a continuous and real-time data transmission commonly associated with a high volume of data. Replicating this behavior in testing environments can be costly and unfeasible [19].

To ensure quality in the developed software and improve fault detection in scenarios of low testability, a complementary approach is necessary to conventional testing when the application is running in a production environment, in order to evaluate the software reliability. Since the mentioned features do not allow for the construction of a test oracle, there is a need for testing alternatives that do not rely solely on I/O data.

The Tricorder methodology, described in the next section, should not be confused with the *Tricorder* platform, proposed by Sadowski et al. [15], designed to help developers with program static analysis. Despite both aiming to improve software quality, the projects are not related, as our approach is based on software reliability metrics to automate anomaly detection.

3 The Tricorder Methodology

Tricorder [12] is a complementary testing methodology designed to detect potential faults in production software by exploring the resource consumption profiles of the application. One of the advantages of this methodology in comparison to more traditional approaches is that it does not rely on test cases, nor does it depend on input and output data. It simply uses the resource consumption profiles of the application to infer whether there is an anomaly in its behavior. This process is done automatically, without human intervention, although it is the tester's responsibility to decide whether a resource consumption anomaly and a software fault are correlated.

A conventional testing process, as shown in Fig. 1, aims to define a set of test cases consisting of inputs collected from the SUT's domain and their expected outputs for a particular SUT to be tested against it. The result of this testing activity is sent to an Oracle, which is responsible for determining whether or not the test cases detected an actual fault. It is said that the testing activity succeeded when a fault was found and failed otherwise [3].

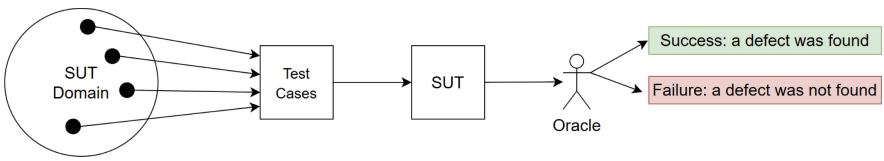

Fig. 1. Traditional testing process.

However, since the Tricorder methodology does not use inputs and outputs to detect anomalies, there is no need for test cases. The SUT domain is used instead to define workloads as different combinations of input parameters that are used to express the different known behaviors of the application. These workloads are executed against a version of the application, said to have no faults, to be used as a reference. This methodology will produce a set of reference resource consumption profiles, which will later be compared with the ones generated by the SUT with the same workloads. The result of this activity will indicate whether an anomaly was detected. The tester is assigned to identify whether that anomaly corresponds to an actual code fault, as shown in Fig. 2.

The complete list of steps to apply the Tricorder methodology is described as follows [12]:

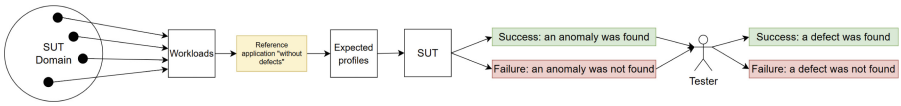

Fig. 2. Tricorder's testing process.

1. There should be a reference application **P**, which is assumed to have no faults. The running of **P** will be triggered **N** times;

2. For each execution of **P**, data regarding consumption of the available resources (such as RAM, processors, etc.) will be collected and stored;

3. The collected data is organized as a set of resource consumption profiles that will be grouped, according to a similarity metric, in an incremental manner, with each newly generated profile added to the grouping;

4. Then, a new version of the application, **P'**, which will be the system-under-test (SUT), is executed **N** times;

5. For each execution of **P'**, steps (2)-(4) will be executed all over again, collecting data and generating profiles of the resource consumption;

6. The new profiles will be added to the previous set, and the grouping process will be executed again incrementally; and

7. If a new group is formed with only **P'** data, it is assumed that an anomaly was detected, which could represent the presence of a fault.

The chosen algorithm for the grouping of the resource consumption profiles is based on the DAMICORE methodology [16], which is a generic approach to grouping data of any type by using Normalized Compression Distance (NCD) as a similarity metric between two data points. The DAMICORE workflow can be described in three steps [16]:

– **Computing the NCD metric**: generate a distance matrix by computing the NCD value between each pair of data points;

– **Simplification**: the distance matrix is transformed into a rootless binary tree, where each data point is presented as a leaf node, and their similarity degree is reflected as the length of the edges between them; and

– **Grouping**: the data grouping is hierarchically performed within the binary tree, causing the original data set to be partitioned;

To validate the effectiveness of fault detection, experimental Python scripts were developed to implement the Tricorder methodology. This experimental model is represented in Figs. 3 (representing the **P** application) and 4 (representing the **P'** application).

The **Monitoring** module [12], executes different workloads associated with the SUT. The workloads should be as close as possible to a real scenario in which the application should run. To ensure effective grouping in laboratory experiments, previous work usually sets up the SUTs to execute each workload at least 30 times to generate fitted resource consumption profiles. The Python library **PSUtil** was used to retrieve consumption data (specifically regarding CPU usage, number of allocated bytes in memory, and number of I/O operations) every 100 milliseconds [12]. A profile gathers all the data collected in each workload execution.

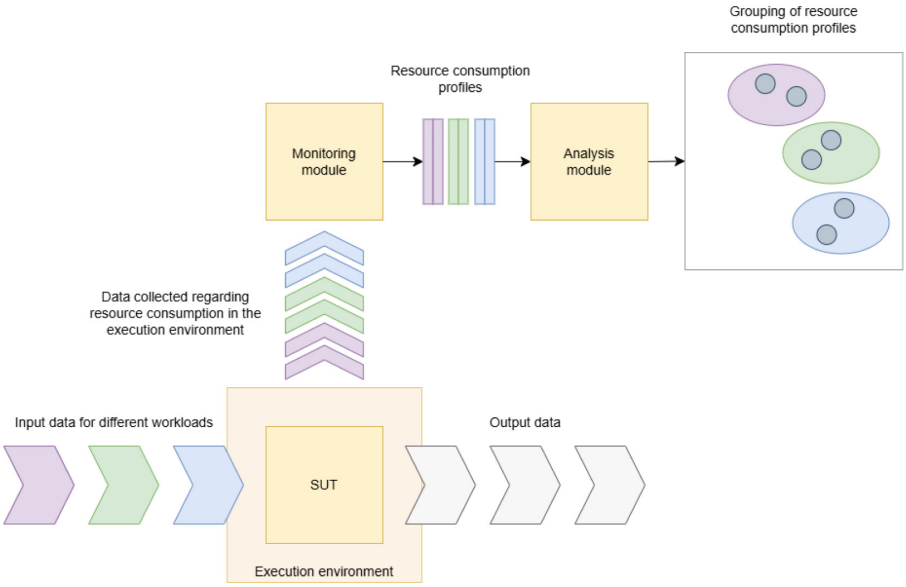

Fig. 3. Tricorder expected behavior when running P.

The **Analysis** module groups the collected data using DAMICORE [16]. Any newly generated group with executions of **P'** indicates that **P'** changed the application's resource consumption profile, pointing out an anomaly. From there, the researcher (or tester) needs to evaluate the anomaly and identify whether there is an actual fault in the SUT or the new profile is expected due to a change in the code.

The Tricorder methodology has been used in many scenarios and applications. While [12] explored the methodology in a Client-Server application, [1] evaluated the performance when exploring Machine Learning algorithms, and

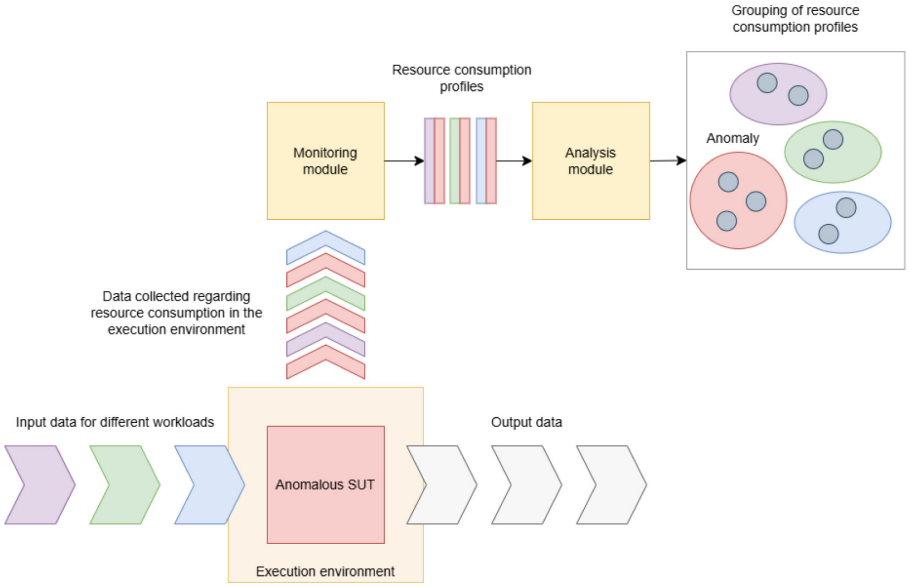

Fig. 4. Tricorder expected behavior when running P'.

[4] explored different application domains, such as Fluid Simulation and Video Processing. The results show that Tricorder can detect artificially inserted faults in every domain by grouping resource consumption profiles.

Tricorder's current implementation lacks the robustness of a real tool. It demands a significant effort to adapt its scripts, as the tool modules and source code were initially built to be used as experimental resources.

4 Architectures for Software Testing Tools

In the scientific literature, there is a plethora of proposed tools and approaches that deal with the activity of software testing, even for contexts that share the characteristics of low testability software, such as IoT systems [6]. However, most are not publicly available for replication and usage in different contexts [8].

One strategy that can improve the adoption of a new tool is using a reference architecture. Relying on a software architecture can improve the software maintainability and life cycle, and also reduce development time [7]. Using software architectures during the development process can encourage reusability of components [9].

The proposal of software testing architectures before an actual implementation of a tool is usual among researchers. A cloud architecture was proposed, with an actual implementation, for executing static tests as a service [2], in which users can upload the applications to be tested, execute their tests, and download their results.

Zhang [22] presents an architecture for software testing in web service applications based on a service-oriented testing framework proposed by Zhu [23], accompanied by a prototype implementation based on the ontology of software testing, demonstrating the feasibility of the proposed architecture.

Wu [21] presents CCTS, an architecture for a software testing tool to test event-based micro-services through consumer-driven contracts and state models.

Nakagawa [13] highlights the importance of standardization in testing tool proposals. The authors proposed RefTEST, a reference architecture specialized for constructing software testing tools, based on RefASSET, an architecture proposal focused on wider software engineering environments.

The RefTEST architecture was used as the baseline for the development of software testing tools, such as the one described by Silva [18], which presents a software testing tool for concurrent systems. Parizi [14] also proposes a testing tool, RAMBUTANS, based on RefTEST, which intends to test aspect-oriented programming.

By compiling the accumulated knowledge contained in the scientific literature for software testing (represented by Software Architectures, Testing Processes, Testing Tools, and Testing Ontologies), RefTEST suggests the definition of four modules (Fig. 5) [13]. Each module has a list of functionalities to encompass the expected features of a software testing tool:

- The **Test Artifact** defines the artifact to be tested by the tool and provides all the features related to its management.
- The **Test Case** defines the test cases to be executed against the Test Artifact, as well as the features for their management.
- The **Test Criterion** defines the test criterion used by the tool and provides the features related to its management.
- The **Test Requirement** defines the test requirements for the tool and provides the features for their management.

By using RefTEST as a baseline, this study aims to provide an architecture to describe a reliability tool capable of implementing the Tricorder methodology. Moreover, we aim to instantiate the proposed architecture into a conventional software scenario and also an IoT scenario to study its adaptability and adherence.

5 Mapping Tricorder to RefTEST Architecture

Aiming to understand the process and define the behavior for the new tool, as well as the necessary parameters to be provided by the tester, we describe a proposal of a workflow based on the Tricorder methodology and its current implementation.

The proposed tool can be adapted to encapsulate the guidelines provided by the RefTEST architecture. For each module, we need to specify the necessary

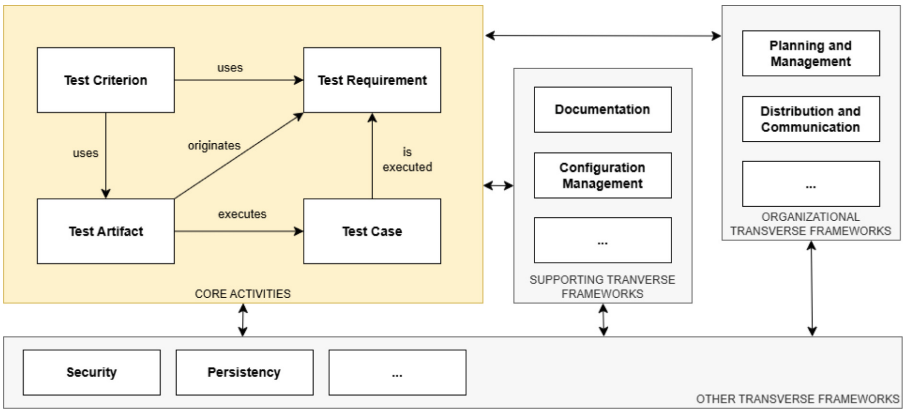

Fig. 5. Core aspect of the RefTEST architecture. Adapted from Nakagawa [13]. (Color figure online)

functional requirements for the tool being developed. In addition, we can derive adequate classes and methods from this definition. The RefTEST architecture also proposes a set of transverse frameworks that can be used to address other activities that need to be implemented for the tool to work in a real environment. These frameworks improve and support the established core activities, even being functional on their own, by implementing functionalities related to **supporting activities** - such as Documentation and Configuration Management -, **organizational activities** - such as Planning and Management - or other general activities, related to Security and Data Persistence, for example. The frameworks can also be used in a new tool proposal to encapsulate non-core activities properly.

An initial workflow was designed to represent all the steps the tool should take to implement the Tricorder methodology properly. The workflow provides insights into the stages of the activity and the parameters that a tester should provide to use the tool effectively. Since the Tricorder methodology is heavily dependent on clustering algorithms, the reliability tool was organized into three steps: training, validation, and testing.

The **training** step considers a version of the application that is admittedly fault-free to be used as a reference. From this version, some executions will be triggered and monitored by the tool, which will collect data regarding the usage of the available resources and generate a set of resource consumption profiles. From there, iterating through each profile, the tool will run a clustering algorithm until all profiles are used. The generated grouping will be saved as a reference and used in the following steps (Fig. 6).

Validation is a double-check step in which the tool will again receive the reference application and generate new resource consumption profiles to compare with the ones generated in the Training steps. The stages of running the application and grouping the results will be executed again. Still, this time, the

training groups will represent an initial set in the grouping stage, where the new profiles will be aggregated. The final grouping is expected to be the same as the original one. If not, the tool might need parameter changes, or the target application is unsuitable due to a non-deterministic behavior (Fig. 7).

Test is the final step. A new version of the SUT will generate new resource consumption profiles, which will be grouped with the Training/Validation profiles. This step is similar to the validation step, but now the main goal is to investigate if a new group containing only profiles from the Test step is generated. If so, an anomaly is detected (Fig. 8).

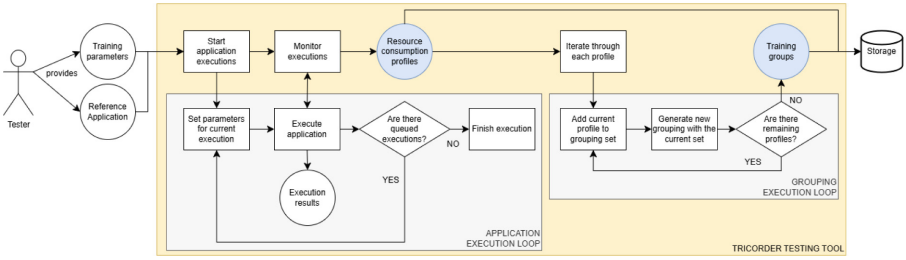

Fig. 6. Workflow overview for the Training step of the proposed tool.

The rectangles in Figs. 6, 7 and 8 represent block of actions, circles denote data and information (blue are in external storage), diamonds indicate conditional statements, and hexagons show conclusions to be taken by the tester depending on a condition.

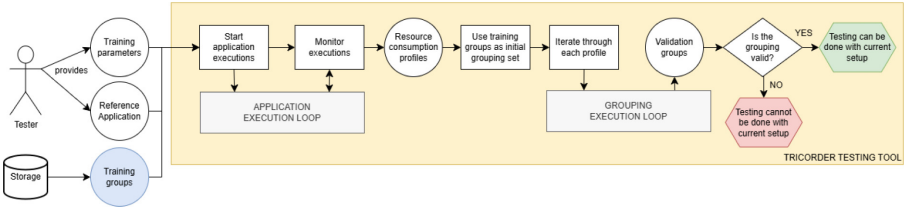

Fig. 7. Workflow overview for the Validation step of the proposed tool.

The non-conventional approach in the Tricorder methodology provides additional challenges to the use of a reference architecture based on standard software testing tools and processes. Concepts and functionalities must be adapted to this new context while keeping the architecture functional and structured as intended. This application illustrates and encourages the use of RefTEST architecture in non-conventional testing scenarios.

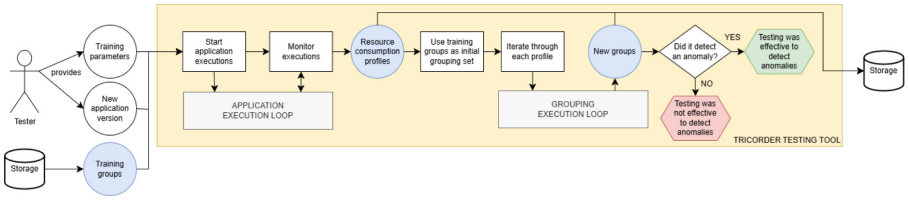

Fig. 8. Workflow overview for the Test step of the proposed tool.

By using RefTEST as the reference architecture, this study aims to provide a set of concepts and functionalities needed for the proposed tool to work correctly. The conceived architecture also needs to be instantiated in a more practical sense so that its implementation can be fully envisioned. This instance would require the definition of use cases, functional and non-functional requirements, class diagrams, etc. For this study, however, we aim to reach a proposal on how the tool should be organized, what modules should be developed, and how they would communicate with one another. The technical details, requirements, and use cases will not be covered in this work due to size constraints.

6 Results and Discussion

Using RefTEST as the baseline and the established workflows as guidance for the behavior of the proposed tool, we present the adapted concepts for each module. It is noteworthy that the purpose of our tool is to validate a system in use, with the goal of identifying the presence of defects when the software is in its production environment. The software testing concepts used for the development cycle were adapted to the context of our reliability engineering tool.

- **Test Artifact**: In the context of the Tricorder methodology, we can associate the artifact to be tested with a **new application version**, considering that there is a baseline version as the reference.

- **Test Case**: The different test scenarios to explore the artifact to be tested can be described as different **workloads** in the Tricorder methodology. The method can work without the definition of multiple workloads, as standard execution should be enough to build a resource consumption profile of the application. However, by including different workloads, testers can ensure that possible different resource demands of the application can be mapped and detected by the tool as allowed occurrences and not as false positives.

- **Test Criterion**: As the Tricorder methodology does not rely on code structure or implementation, the concept of coverage cannot rely on these features. However, since we have a Training step before using the tool for the actual testing, we collect a set of profile groups representing the expected application behaviors regarding resource consumption. At the Test step, the test criterion

is the coverage of the groups found in the Training. Also, during the Test, we should be able to identify resource consumption profiles belonging to each group in the Training step.

– **Test Requirement**: Similarly to the described Test Criterion, since the Tricorder methodology is not concerned about internal aspects of the SUT implementation, nor does it rely on input and output data, we can set, as the test requirement, the existence of resource consumption profiles that only belong to the predefined groups. An anomaly is detected if some resource consumption profiles are not grouped into an existing group.

The proximity of each RefTEST core concept to the concepts of the Tricorder methodology indicates that not only is the reference architecture versatile, but it is also feasible to guide the development of a tool for the Tricorder methodology.

Aside from the concepts that coordinate the behavior of a software testing tool, it is also necessary to persistently store and retrieve data (in the case of Tricorder, the resource consumption profiles and groupings), from one step to another in the workflow, and even from different test sessions. These features are also observed in RefTEST's architecture specification due to the definition of a transverse framework for **Data persistence.**

In order to accommodate all the necessary functions for the methodology while also taking into account the previously defined concepts, we can derive a generic overview of architecture to implement Tricorder, as shown in Fig. 9. The user interface module must provide an UI that allows the tester to interact with the system and trigger the execution of the monitoring and analysis modules. The data persistence module ensures that all the data needed during this process is consistent, reliable, and stored persistently.

Both analysis and monitoring modules are responsible for leveraging the core concepts of RefTEST, highlighted in yellow and shown on Fig. 5, to execute their function successfully. Moreover, to properly isolate resource consumption data collection from any interference, the monitoring module should be able to execute the SUT by demand in a dedicated environment, which might differ for each use case of the tool when instantiating its architecture with real resources.

This first proposal is generic and should be defined before being applied in any scenario. One of the first attempts to instantiate it for a typical case of a regular application to be tested is described in Fig. 10, where both the analysis and monitoring modules run as back-end applications in isolated containers, while the UI module is represented by a front-end web application, running locally on the tester's machine. The data persistence module stores the needed data in a SQL database.

A second attempt to adjust the architecture in a specific scenario focuses on IoT systems. For example, suppose that the software to be monitored runs at the perception layer. In that case, we might deal with hardware constraints and resource limitations that could prevent a monitoring module from being fully executed according to the specifications. Thus, an additional module might be needed to simplify the hardware overhead, as proposed in Fig. 11.

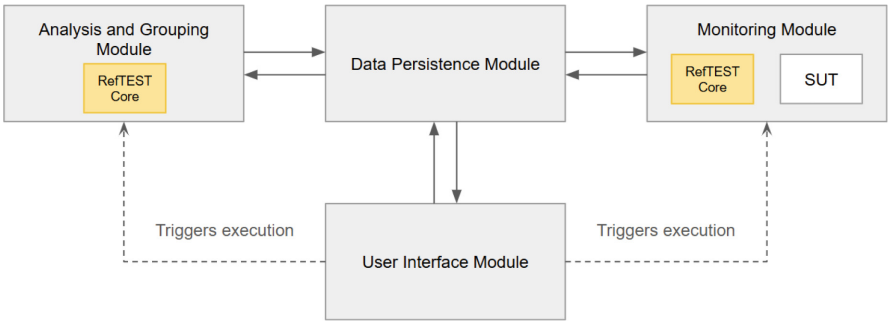

Fig. 9. Generic architecture of the Tricorder methodology.

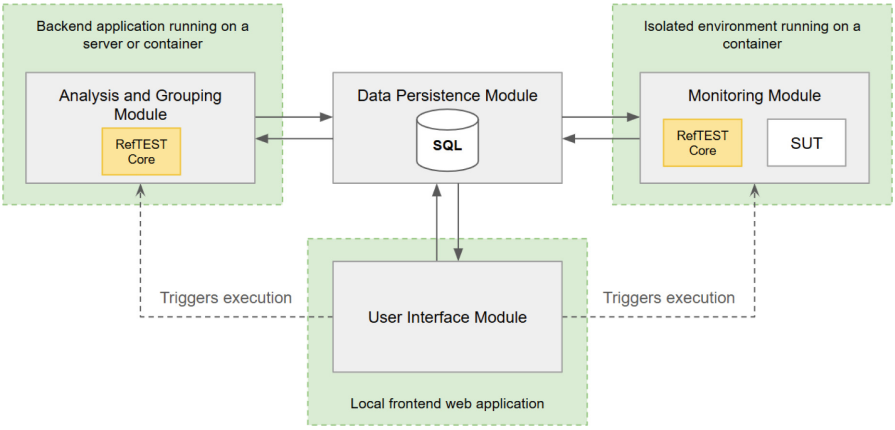

Fig. 10. Common definition for the architecture of Tricorder's proposed tool.

A monitoring module is used only to collect the current resource consumption data and send it to an external MQTT server as a publisher. The tool's persistence module can act as a subscriber and store the collected data in real time.

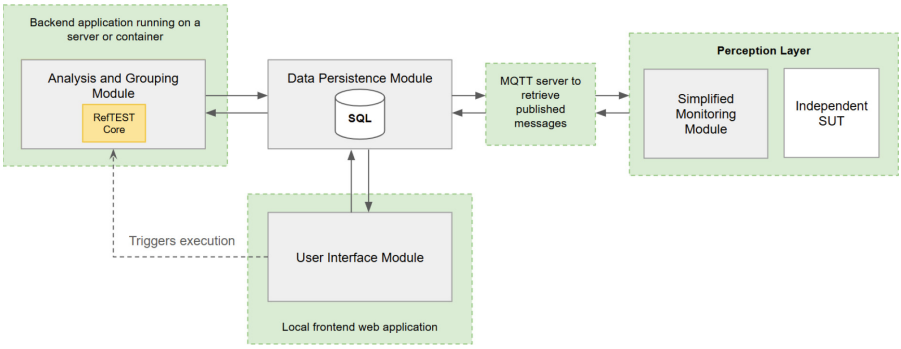

Fig. 11. IoT definition for the architecture of Tricorder's proposed tool.

6.1 An Use-Case IoT Scenario for the Tool

Suppose that an IoT application runs constantly on an ESP32 microcontroller, attached to a set of sensors that monitor aspects of the environment in which this circuit is exposed, such as temperature, humidity, air quality, etc. There is a low-level code on the perception layer for processing the collected data and sending them to another application running on a computer as application layer software. Both pieces of code (perception and application layers) are validated through traditional software testing methods, such as unit tests, integration tests, system tests, etc.

Assume that there is a fault in the code for a new version of the software on the perception layer, which causes a memory leak. The software can start to malfunction after some time running, caused by the memory consumption on the microcontroller. Since hardware resources are scarce in IoT systems, issues such as these cannot be overlooked. At the same time, traditional testing may not be enough to detect this fault, as it becomes more evident as the software runs.

Through the Tricorder methodology, such a scenario would be feasible to detect since a memory leak would represent a deviation from the expected resource consumption profile. By being trained on a more stable and reliable version of the software, the tool would identify a new and distinct group of profiles for RAM consumption, which represents an anomaly in the latest software.

Based on the proposed mapping of the Tricorder to a software testing reference architecture, we can draw the following results:

1. Proof of adherence to the Tricorder methodology as a future reliability tool, complementary to the in-production software testing activity, through the use of RefTEST;

2. Instantiation of different propositions for the original architecture in different scenarios, including an example of a low testability software, such as IoT; and

3. Contextualization of a possible scenario in which the Tricorder can be used to detect faults that traditional testing approaches would not detect.

7 Conclusion and Future Work

This paper presents an architecture for a reliability tool that implements the Tricorder methodology, offering additional support to the testing activity through a non-conventional approach. RefTEST was chosen as the reference architecture, where we extend its use by adapting its concepts to encompass all the necessary steps for non-conventional testing activity.

The work reinforces that a reliability tool based on RefTEST can successfully implement the Tricorder methodology. This work is an important validation, as it is the first step towards implementing the proposed tool. In addition, the proposal contributes to the idea that RefTEST is versatile and can be used in non-conventional scenarios, which could expand its adoption.

This study also approximates the proposed architecture from a real-world scenario, defining the technical details for each module in different scenarios, such as IoT systems. Beyond that, we characterize a use-case scenario in which such a tool can bring benefits for fault detection, catching software anomalies that traditional testing mechanisms, such as memory leaks, could not detect.

Future work will apply this architecture by building a proof-of-concept of the Tricorder tool. To implement the reliability tool, one should specify the technical aspects even further, reaching concepts such as the list of necessary classes and objects to represent each concept and implement each functionality.

Acknowledgment. The authors would like to thank CAPES (PROEX), CNPq (308445/2021-0), and FAPESP (2013/07375-0, 2019/06937-0, 2019/26702-8) for their financial support.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Braga, D.: (Brazilian Portuguese) Assessing the Use of Performance Data Clustering for Supporting Software Testing in the Machine Learning Domain. Master's thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos (2022). https://doi.org/10.11606/D.55.2022.tde-09112022-160513
2. Chen, S., Huang, J., Gong, Y.: Static testing as a service on cloud. In: 27th International Conference on Advance Information Networking and Applications Workshops, pp. 638–642 (2013). https://doi.org/10.1109/WAINA.2013.257
3. Delamaro, M., Jino, M., Maldonado, J.: (Brazilian Portuguese) Introdução ao teste de software. Elsevier Brasil, Rio de Janeiro, RJ (2013)

4. Durães, T.d.J.O.: (Brazilian Portuguese) Evaluating the use of performance profiles applied in software testing in different application domains. Master's thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos (2022).https://doi.org/10.11606/D.55.2022.tde-05042023-144747
5. Dyck, A., Penners, R., Lichter, H.: Towards definitions for release engineering and DevOps. In: IEEE/ACM 3rd International Workshop on Release Engineering, pp. 3–3 (2015). https://doi.org/10.1109/RELENG.2015.10
6. Ferreira, V.G., Herrera, C.G.a., Souza, S., Santos, R.R.d., Souza, P.S.L.d.: Software testing applied to the development of IoT systems: preliminary results. In: Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing, p. 113–122. SAST '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624032.3624049
7. Gillani, M., Niaz, H.A., Ullah, A.: Integration of software architecture in requirements elicitation for rapid software development. IEEE Access **10**, 56158–56178 (2022). https://doi.org/10.1109/ACCESS.2022.3177659
8. Herrera, C.G., Ferreira, V.G., Souza, S.R.S., Santos, R., de Souza, P.S.L.: A comprehensive exploration of the use of software testing tools for IoT systems. In: Anais do XXV Workshop de Testes e Tolerância a Falhas. SBC (2024)
9. Joy Goodman-Deane, P.L., Clarkson, J.: Key influences on the user-centred design process. J. Eng. Design **21**(2-3), 345–373 (2010). https://doi.org/10.1080/09544820903364912
10. Lyu, M.R.: Software reliability engineering: a roadmap. In: Future of Software Engineering (FOSE '07), pp. 153–170 (2007). https://doi.org/10.1109/FOSE.2007.24
11. Microsoft: What is DevOps? (2024). https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-devops/. Accessed 19 Jan 2025
12. Montes, V.S.: (Brazilian Portuguese) Software fault detection using clustering of performance profiles. Master's thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos (2019). https://doi.org/10.11606/D.55.2020.tde-04022020-093604
13. Nakagawa, E., Simão, A., Ferrari, F., Maldonado, J.: Towards a reference architecture for software testing tools. In: 19th International Conference on Software Engineering and Knowledge Engineering, SEKE 2007, pp. 157–162 (2007)
14. Parizi, R., abdul ghani, A.a., Lee, S., Khan, S.U.R.: RAMBUTANS: automatic AOP-specific test generation tool. Int. J. Softw. Tools Technol. Transfer **19** (2017). https://doi.org/10.1007/s10009-016-0432-3
15. Sadowski, C., Van Gogh, J., Jaspan, C., Soderberg, E., Winter, C.: Tricorder: building a program analysis ecosystem. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, pp. 598–608 (2015). https://doi.org/10.1109/ICSE.2015.76
16. Sanches, A., Cardoso, J.M., Delbem, A.C.: Identifying merge-beneficial software kernels for hardware implementation. In: International Conference on Reconfigurable Computing and FPGAs, pp. 74–79 (2011). https://doi.org/10.1109/ReConFig.2011.51
17. Shieh, S.W., et al.: Reliability engineering in a time of rapidly converging technologies. IEEE Trans. Reliab. **73**(1), 73–82 (2024). https://doi.org/10.1109/TR.2024.3355905
18. Silva, R., Souza, S.S., Souza, P.L.: Using a reference architecture to support the development of a testing tool for concurrent software. In: International Conference on Information Systems and Technology Management (2013). https://doi.org/10.5748/9788599693094-10CONTECSI/PS-419

19. Vianna, A., Kamei, F.K., Gama, K., Zimmerle, C., Neto, J.A.: A grey literature review on data stream processing applications testing. J. Syst. Softw. **203**, 111744 (2023). https://doi.org/10.1016/j.jss.2023.111744, https://www.sciencedirect.com/science/article/pii/S0164121223001395
20. Voas, J., Miller, K.: Software testability: the new verification. IEEE Softw. **12**(3), 17–28 (1995). https://doi.org/10.1109/52.382180
21. Wu, C.F., Ma, S.P., Shau, A.C., Yeh, H.W.: Testing for event-driven microservices based on consumer-driven contracts and state models. In: 2022 29th Asia-Pacific Software Engineering Conference (APSEC), pp. 467–471 (2022). https://doi.org/10.1109/APSEC57359.2022.00064
22. Zhang, Y., Zhu, H.: Ontology for service oriented testing of web services. In: 2008 IEEE International Symposium on Service-Oriented System Engineering, pp. 129–134 (2008). https://doi.org/10.1109/SOSE.2008.35
23. Zhu, H.: A framework for service-oriented testing of web services. In: 30th International Computer Software and Applications Conference. vol. 2, pp. 145–150 (2006). https://doi.org/10.1109/COMPSAC.2006.95