Automated Testing for Constrained Embedded Systems: A Maturity-Oriented Approach

Alan Gramacho dos Santos[1], Andressa Oliveira Silva[2*], Breno Prazeres Barbosa[3], Frederico Garcia de Oliveira[4], Tassio Ferreira Vale[5], Valter Estevão Beal[6]

[1] *Senai Cimatec, Department of Embedded Electronics, Salvador, Bahia, Brazil - alan.gramacho@fieb.org.br*
[2]*Senai Cimatec, Department of Embedded Electronics, Salvador, Bahia, Brazil - andressa_o_silva@hotmail.com*
[3]*Senai Cimatec, Department of Embedded Electronics, Salvador, Bahia, Brazil - breno.barbosa@fieb.org.br*
[4]*Senai Cimatec, Department of Mechanical Projects, Salvador, Bahia, Brazil - frederico.oliveira@fieb.org.br*
[5]*Federal University of Recôncavo da Bahia, Department of Computer Engineering, Cruz das Almas, Bahia, Brazil - tassio.vale@ufrb.edu.br*
[6]*Senai Cimatec, Department of Mechanical Projects, Salvador, Bahia, Brazil - valtereb@fieb.org.br*
**Corresponding author: SENAI CIMATEC, Av. Orlando Gomes. 1845. Piatã, Salvador Ba.*

Abstract: The increasing complexity of embedded systems, especially in critical environments, demands reliable and scalable validation methodologies. This work presents an automated testing framework based on Python to validate embedded systems through acoustic and serial communication. The tool executes protocol-defined commands, logs responses, and generates reports with quantifiable metrics. Through this structured approach, the system enables proactive identification of edge-case failures and aligns with Capability Maturity Model Integration (CMMI) and Technology Readiness Level (TRL) frameworks. Results indicate improved test coverage, reduced post-deployment bugs, and process standardization, even in resource-constrained environments.

Keywords: Automated Testing · System Maturity · Embedded Systems · Protocol Validation · CMMI
Abbreviations: CMMI, Capability Maturity Model Integration. TRL, Technology Readiness Level. ISTQB, International Software Testing Qualifications Board. UART, Universal Asynchronous Receiver-Transmitter. TTL, Transistor-Transistor Logic.

1. Introduction

The increasing complexity of embedded systems demands structured, automated validation techniques that contribute directly to the system's maturity and reliability. As noted by Humble and Farley [1], continuous, automated testing plays an important role in enhancing system resilience and reducing time-to-deployment. This paper examines how automated testing frameworks, such as the Python-based tool developed here, enhance the maturity of complex systems by standardizing validation processes, reducing human error, and generating structured reports with detailed outcome metrics.

The proposed solution leverages acoustic and serial communication to interface with a controller board, executing all protocol-defined commands while generating reports with success/failure rates and operational responses. By supporting programmable test scenarios, the tool validates functionality and identifies edge-case vulnerabilities. This aligns with Capability Maturity Model Integration (CMMI) principles, where automated testing elevates maturity from ad hoc routines to quantitatively managed processes.

Prior studies highlight that automation accelerates development and strengthens resilience [1], but gaps remain in adapting these methods to constrained embedded environments. This work demonstrates how a lightweight, protocol-aware tool enforces rigorous standards while contributing to maturity metrics such as process predictability and defect density [2].

2. Theoretical Framework

To assess the impact of automated testing on embedded system development, this section introduces two evaluation frameworks: the CMMI and the Technology Readiness Levels (TRL). CMMI provides a model for assessing process maturity across five levels, from initial (Level 1) to optimizing (Level 5), emphasizing process standardization, measurement, and continuous improvement [2]. TRL, on the other hand, evaluates technological development stages, from conceptual research (TRL 1) to full operational deployment (TRL 7). This study focuses on how the proposed testing framework supports progression across these levels, particularly through repeatability, traceability, and data-driven feedback.

Given the remote and unassisted nature of the Device Under Test (DUT), which operates in environments such as sealed enclosures or underwater conditions, the approach follows black-box testing principles. The DUT cannot be instrumented internally during operation and therefore must be evaluated based solely on its external responses to predefined commands [6]. This justifies the choice of protocol-centric and interface-aware testing, especially considering the constraints in accessibility, power cycling, and communication.

2.1 Automated Testing as a Maturity Catalyst

Automated testing plays an important role in promoting system maturity, as defined by the **CMMI** framework [2]. As systems evolve, their validation processes should transition from ad hoc routines to structured, repeatable, and quantitatively managed workflows. Crucially, CMMI is not limited to testing but integrates all organizational processes, from requirements through deployment. Automated validation contributes to maturity when embedded in this wider cycle, where results inform requirements, design corrections, and deployment readiness.

The DUT includes a microcontroller-based system with sensors, control, and communication functions that must respond deterministically under diverse conditions. Manual testing is inefficient, error-prone, and inadequate for ensuring long-term reliability. Automated testing simulates operational conditions, validates protocol coverage, and generates structured reports that support both system validation and organization-wide process improvement.

This scenario justifies the development of an automated testing framework that can simulate real operational conditions, validate complete protocol coverage, and generate structured performance reports. Automated build-test-deploy pipelines accelerate feedback loops [1], a principle applied here through acoustic/serial communication with embedded systems.

Compared to existing tools such as Robot Framework and Latch [6], which are typically tailored to either high-level system APIs or desktop-class embedded platforms, the proposed framework addresses low-level, protocol-specific testing in highly constrained environments. Unlike Robot Framework, which requires OS-

level support, or Latch, which focuses on constrained systems without domain-specific interfaces, the proposed framework combines lightweight execution with protocol-centric testing for remote and inaccessible environments. Its integration of acoustic communication and maturity-oriented reporting distinguishes it from existing tools, which typically validate functionality without linking outcomes to organizational improvement.

2.2 Protocol-Centric Testing in Constrained Environments

Embedded systems face validation challenges due to resource limitations. As prior research shows [7], lightweight, protocol-aware approaches are essential. The paradigm of protocol-centric validation enables testing in inaccessible conditions, such as underwater deployments, where traditional wired methods fail. Configurable test logic reflects agile testing principles, supporting scenario simulation. Compliance with international guidelines ensures behavioral consistency and deterministic validation.

2.3 Structured Reporting and Maturity Benchmarking

Structured reporting transforms raw test data into maturity indicators. Standardized metrics such as success/failure rates, response times, and defect density underpin statistical process control, supporting CMMI progression from Level 3 to Level 4 [2]. These measurements also benchmark TRL advancement, from laboratory validation (TRL 4–5) to operational readiness (TRL 6–7) [4].

Research [7,8] shows structured reporting generates transparency, root-cause analysis capability, and historical data for risk modeling. As formalized by CMMI [2] and ISTQB [3], automated reporting functions as a maturity accelerator by converting reactive testing into proactive, data-driven practice.

Transparency through auditable command-response records eliminates informational gaps in validation processes [4]. Temporal and decoded logs enable root-cause failure analysis rather than symptomatic troubleshooting [8]. Historical data standardization supports probabilistic risk modeling essential for predictability [1].

Theoretical frameworks synthesize structured reporting's role as a maturity catalyst. As formalized by CMMI [2] and ISTQB standards [3]: *"Automated reporting functions as a maturity accelerator, converting reactive practices into data-driven proactivity"*. This evolution materializes when quantitative indicators support CMMI Level 4 decision-making, structured logs enable continuous optimization (Level 5), and traceability validates higher TRLs (≥6) for operational deployment [4].

3. Methodology

This study adopts an applied engineering methodology grounded in experimental observation and measurement. The hypothesis is that a lightweight, protocol-aware framework improves maturity, reliability, and efficiency in

constrained embedded systems. The methodological path followed five stages: (1) requirements and constraints definition, (2) framework design, (3) implementation on DUT hardware, (4) systematic test execution via UART and acoustic channels, and (5) analysis through predefined metrics.

The proposed framework integrates hardware and software elements to establish a fully automated testing pipeline.

Materials and parameters: The DUT was an ARM Cortex-M4 controller with FreeRTOS, acoustic modem, UART interface, and peripheral modules. The host machine ran Python 3.12 with PySerial and ReportLab. Evaluation metrics included protocol coverage, success/failure ratio, execution time, repeatability, and defect detection. These parameters ensure reproducibility and transparency.

3.1 Application Layer

Developed using the pyserial library [10], the application layer is responsible for the automated execution of all commands defined in the embedded system's communication protocol. Such as power on and off modules, measure and acquire sensors data.

The application is modular and configurable, allowing users to define test sequences, edge-case scenarios, and expected responses for each command.

3.2 DUT Embedded System Layer

The device under test (DUT) is based on a real-time operating system (FreeRTOS), executed on an ARM Cortex-M4 microcontroller. It supports multiple communication interfaces and peripherals such as an acoustic transmission module, MODBUS-compliant communication, and various sensors.

The firmware is structured around a deterministic command-response protocol, ensuring traceability across the testing process.

By supporting both acoustic and TTL-level serial communication interfaces, the framework overcomes the inherent limitations of inaccessible or remote systems, such as underwater or sealed devices, where traditional wired validation is not feasible [7].

3.3 Reporting Engine

A dedicated module within the Python application generates structured reports using the ReportLab library [10]. These reports include a complete command execution log, a decoded FreeRTOS protocol response with human-readable descriptions, a success/failure status for each test step, and temporal data to correlate commands and environmental variables.

The system's ability to generate detailed reports, including success/failure metrics and decoded response logs, directly contributes to CMMI Level 4 ("Quantitatively Managed") by enabling performance measurement, repeatability, traceability, auditability of all test campaigns and statistical control [6].

3.4 Dual-Channel Communication Strategy

Given the constraints of embedded environments, such as limited physical access and low-level hardware interfaces. The framework implements two types of communication channels. The serial for bit level in-lab validation and acoustic for underwater environments. This feature extends test coverage into operationally relevant environments, enhancing realism and robustness.

3.5 Protocol Execution and Response Decoding

Each protocol command is executed in isolation and sequence. The system records raw responses from the controller, decoded interpretations of each response (e.g., translating 0x0B to "Measure Clock Error"), and execution timestamps to enable performance profiling.

This level of detail supports the identification of behavioral inconsistencies and the early detection of defects in both firmware and hardware components.

The inclusion of user-configurable testing logic enables simulation of a wide range of operational conditions, including edge cases. This approach follows best practices in agile software testing, which emphasize adaptability and iterative feedback; this supports auditability and facilitates root-cause analysis [8,9].

3.6 Tool Limitations

Although designed to maximize coverage and repeatability, the tool has certain operational constraints. Its operation depends on strict adherence of the DUT to the defined protocol sequence and states. Test scenarios must be preconfigured manually, as there is no dynamic case generation yet. In addition, the tool runs externally to the DUT, requiring intermediate hardware for interfacing and power. These constraints do not compromise their practical use but influence the interpretation of metrics and the generalization of results.

4. Test Bench and Execution Workflow

Figure 1 shows the block diagram of the dedicated test bench assembled to validate the effectiveness of the proposed automated testing framework, simulating real-world operational conditions for the embedded system.

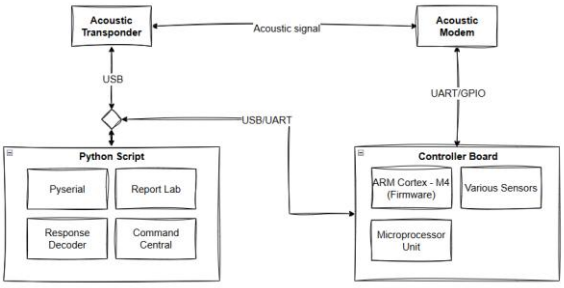

Figure 1 - System Comms

4.1 Test Environment Setup

The test bench consisted of a controller board featuring an ARM Cortex-M4 microcontroller running FreeRTOS, connected to the test system via UART and acoustic interfaces. Peripheral Modules, including various sensors, an acoustic transducer, and MODBUS-based communication modules.

The Host Machine is a computer running the Python-based automation tool, responsible for managing all test procedures, capturing responses, and generating reports.

Power supply and interface adapters are used to ensure the reliable operation of modules under test. Verifying the integrity of command-response behavior across different operational states and environments, the tool has compliance with the ISTQB guidelines [2], ensuring that protocol validation adheres to internationally recognized standards.

4.2 Execution Procedure

The test execution followed a systematic approach to ensure full protocol coverage and reproducibility of results. First, all modules were powered on and initialized through predefined commands issued via the application layer. Initial status checks were performed to validate firmware readiness and confirm proper interface activation. Then the entire set of protocol-defined commands was executed in sequence, covering operations such as power cycling modules, triggering measurements, reading memory blocks, and querying sensor data. Testing all operational cases of the DUT 50 times to evaluate the expected response and the consistency of the firmware protocol and the DUT functionalities.

Each command was sent through both the UART and acoustic interfaces to compare results across communication channels. The system responses were recorded in real time, and the raw outputs were logged alongside decoded messages, timestamps, and status classifications (e.g., success, failure, timeout).

The framework validated that command responses were identical across both modes, confirming functional equivalence and interface robustness. However, it is important to note that acoustic communication takes approximately four times longer than serial communication to receive a response from the module.

5. Empirical Validation

To assess the effectiveness of the proposed automated testing framework, a series of in-lab tests was executed using the complete set of protocol-defined commands across multiple operational scenarios. The goal was to evaluate improvements in test coverage, system reliability, and post-deployment defect rates.

At each new firmware release the tests were re-executed to validate the quality and reliability of the development. At each error reported a new firmware version was released and evaluated on the same premises. Executing all firmware commands systematically aligns with CMMI's *Defined* (Level 3) stage, where processes are documented, standardized, and integrated across the organization [2].

5.1 Coverage and Automation Metrics

The tool successfully achieved 100% coverage of the system's communication protocol, validating every command in both acoustic and UART modes. This comprehensive coverage enabled a full protocol compliance verification across interfaces, a detection of undocumented or unexpected behaviors, and consistent execution of edge-case scenarios.

The automation capabilities allowed repeated execution of test suites with no manual

intervention, reducing human error and accelerating validation cycles. The integration of *programmable testing*, *hardware-aware communication*, and *structured reporting* creates a feedback loop that elevates system maturity from *reactive* to *proactive* — a critical advancement for safety-critical embedded applications [2,3].

5.2 Defect Reduction and Reliability Gains

Comparative analysis of firmware versions before and after integrating the automated testing framework demonstrated that numerous errors were identified and resolved pre-deployment. This proactive correction substantially reduced field failure rates. The increased frequency of firmware update versions following test implementation provides empirical evidence of these corrective actions. Key observations include the early detection of issues that caused power-saving failures, the elimination of misconfigured responses related to sensor modules and the identification of firmware inconsistencies in the module's integration.

5.3 Process Efficiency and Repeatability

The adoption of the framework also improved overall test efficiency and traceability. Test execution time per module decreased by ~90%, thanks to batch automation and pre-defined test profiles, as the average test cycle reduced from 180 minutes (manual) to 15 minutes (automated), with minimal deviation across repeated sessions. The reports enabled traceability across validation sessions, with version control and serial number signatures. Repeatability was validated by executing the same test suite under different environmental and hardware configurations with consistent results.

5.4 Maturity Impact Assessment

The integration of automated testing and structured reporting contributed to measurable improvements in software process maturity, aligned with CMMI and TRL frameworks.

This alignment positions the tool as a maturity accelerator, transforming embedded system validation from isolated checks into a lifecycle-oriented process.

The automated tests enable the system's evolution by enforcing consistency, accelerating feedback cycles, and reducing the incidence of human-induced errors. The tool records success rates, failure diagnostics, and timing data, aligning with the TRL framework [4]. These outcomes confirm that the framework is not only a test tool but also a maturity accelerator, capable of advancing embedded systems validation into quantifiable, repeatable, and improvement-oriented processes.

6. Conclusion

This study presented the design, implementation, and empirical validation of a protocol-aware automated testing framework aimed at increasing the maturity of embedded systems, particularly those operating in constrained environments. By integrating acoustic and serial communication, structured reporting, and programmable test

scenarios, the framework effectively bridges the gap between informal validation routines and quantitatively managed test processes.

The results demonstrate that automated testing not only enhances protocol compliance and reduces human error but also significantly improves defect detection, test coverage, and execution efficiency. The use of structured reports and decoded protocol responses provides transparency and auditability, enabling traceable and reproducible validation cycles aligned with CMMI and TRL frameworks.

The role of the CMMI here is to coordinate the organizational process model of testing as an integrated element of the software development lifecycle. The framework's outputs feed back into requirements refinement, design validation, and deployment assurance, ensuring that automated testing accelerates maturity across all process domains. Furthermore, the framework's lightweight design ensures compatibility with resource-constrained embedded systems, extending its applicability to real-world, mission-critical deployments. As a result, it serves not only as a testing utility but also as a strategic asset for advancing the reliability, maintainability, and process maturity of complex embedded solutions, as shown in Table 1.

Table 1 - Bridging Theory and Practice

Concept	Implementation in This Work
CMMI Maturity [2]	Integrated into the organizational development lifecycle
Agile-Testing Integration [7]	Programmable test scenarios for diverse conditions
TRL Frameworks [4]	Quantitative benchmarks including success/failure rates, timing, and coverage metrics
ISTQB Compliance [3]	Structured black-box command validation via acoustic and serial communication interfaces

Future work may include integration with continuous integration (CI) pipelines, graphical dashboards for monitoring test metrics, and expansion to support additional communication protocols and hardware platforms.

References

[1] Humble J, Farley D. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Boston: Pearson Education; 2010. p. 283.

[2] CMMI Product Team. *Capability maturity model® integration (CMMI SM), version 1.1*. Pittsburgh: Software Engineering Institute, Carnegie Mellon University; 2002. Report No.: CMU/SEI-2002-TR-029.

[3] ISTQB. *Standard glossary of terms used in software testing*. International Software Testing Qualifications Board; 2016 [cited 2024 Jul 10].

[4] Gil L, Andrade MH, Costa MC. *Os TRL (Technology Readiness Levels) como ferramenta na avaliação tecnológica*. Ingenium. 2014;139:94–6.

[5] Schach, Stephen R. "Testing: principles and practice." *ACM Computing Surveys (CSUR)* 28.1 (1996): 277-279.

[6] Lauwaerts T, Marr S, Scholliers C. Latch: Enabling large-scale automated testing on constrained systems. *Sci Comput Program*. 2024;238:103157. doi: 10.1016/j.scico.2024.103157.

[7] Liu Z, Mei P. Automated testing for large-scale critical software systems. In: *2014 IEEE 5th International Conference on Software Engineering and Service Science*; 2014 Jun 27–29; Beijing, China. IEEE; 2014. p. 200–3. doi: 10.1109/ICSESS.2014.6933544.

[8] Wang Y, Mäntylä MV, Demeyer S, Wiklund K, Eldh S, Kairi T. *Software test automation maturity - a survey of the state of the practice* [preprint]. *arXiv*:2004.09210v1 [cs.SE]. 2020 Apr 20 [cited 2024 Jul 10]. Available from: https://arxiv.org/abs/2004.09210.

[9] Tyagi S, Sibal R, Suri B. Adopting test automation on agile development projects: a grounded theory study of Indian software organizations. In: *Product-Focused Software Process Improvement. PROFES 2017. Lecture Notes in Computer Science*, vol 10611. Cham: Springer; 2017. p. 184–98. DOI: 10.1007/978-3-319-57633-6_12.

[10] Python Software Foundation. *Python Language Reference*, version 3.12. Available at: https://docs.python.org/3/. Accessed: Aug. 2025.