Network Management Research Group
Internet-Draft
Intended status: Informational
Expires: 5 January 2026

Y. Cui
Y. Wei
K. Chi
X. Xie
Tsinghua University
4 July 2025

Framework and Automation Levels for AI-Assisted Network Protocol Testing
draft-cui-nmrg-auto-test-00

Abstract

This document presents an AI-assisted framework for automating the testing of network protocol implementations. The proposed framework encompasses essential components such as protocol comprehension, test case generation, automated script and configuration synthesis, and iterative refinement through feedback mechanisms. In addition, the document defines a multi-level model of test automation maturity, ranging from fully manual procedures (Level 0) to fully autonomous and adaptive systems (Level 5), providing a structured approach to evaluating and advancing automation capabilities. Leveraging recent advancements in artificial intelligence, particularly large language models (LLMs), the framework illustrates how AI technologies can be applied to enhance the efficiency, scalability, and consistency of protocol testing. This document serves both as a reference architecture and as a roadmap to guide the evolution of protocol testing practices in light of emerging AI capabilities.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-cui-nmrg-auto-test/.

Discussion of this document takes place on the NMRG Research Group mailing list (mailto:nmrg@irtf.org), which is archived at https://datatracker.ietf.org/rg/nmrg/. Subscribe at https://www.ietf.org/mailman/listinfo/nmrg/.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Copyright Notice

Table of Contents

1. Introduction

As protocol specifications evolve rapidly, traditional testing methods that rely heavily on manual effort or static models struggle to keep pace. Testing involves validating that a device's behavior complies with the protocol's defined semantics, often documented in RFCs. In recent years, emerging application domains such as the industrial internet, low-altitude economy and satellite internet have further accelerated the proliferation of proprietary or rapidly changing protocols, making comprehensive and timely testing even more challenging.

This document proposes an automated network protocol testing framework that reduces manual effort, enhances test quality, and adapts to new specifications efficiently. The framework consists of four key modules: protocol understanding, test case generation, test script conversion, and feedback-based refinement. It emphasizes modular design, reuse of existing knowledge, and AI-assisted processes to facilitate accurate and scalable testing.

In addition to the proposed framework, this document also defines a six-level classification system (Levels 0 to 5) to characterize the evolution of automation maturity in network protocol testing. These levels serve as a technology roadmap, helping researchers evaluate the current state of their systems and set future goals. Each level captures increasing capabilities in protocol understanding, orchestration, analysis, and human independence.

2. Definition and Acronyms

DUT: Device Under Test

Tester: A network device for protocol conformance and performance testing. It can generate specific network traffic or emulate particular network devices to facilitate the execution of test cases.

LLM: Large Language Model

FSM: Finite State Machine

API: Application Programming Interface

CLI: Command Line Interface

Test Case: A specification of conditions and inputs to evaluate a protocol behavior.

Test Script: An executable program or sequence that carries out a test case on a device.

3. Network Protocol Testing Scenarios

Network protocol testing is required in many scenarios. This document outlines two common phases where protocol testing plays a critical role:

1. Device Development Phase: During the development of network equipment, vendors must ensure that their devices conform to protocol specifications. This requires the construction of a large number of test cases. Testing during this phase may involve both protocol testers and the DUT, or it may be performed solely through interconnection among DUTs.

2. Procurement Evaluation Phase: In the context of equipment acquisition by network operators or enterprises, candidate equipment suppliers need to demonstrate compliance with specified requirements. In this phase, third-party organizations typically perform the testing to ensure neutrality. This type of testing is usually conducted as black-box testing, requiring the use of protocol testers interconnected with the DUT. The test cases are executed while observing whether the DUT behaves in accordance with expected protocol specifications.

4. Key Elements of Network Protocol Testing

Network protocol testing is a complex and comprehensive process that typically involves multiple parties and various necessary components. The following entities are generally involved in protocol testing:

1. DUT: The DUT can be a physical network device (such as switches, routers, firewalls, etc.) or a virtual network device (such as FRRouting (FRR) routers and others).

2. Protocol Tester: A protocol tester is a specialized network device that usually implements a standard and comprehensive protocol stack. It can generate test traffic, collect and analyze incoming traffic, and produce test results. Protocol testers can typically be controlled via scripts, allowing automated interaction with the DUT to carry out protocol tests.

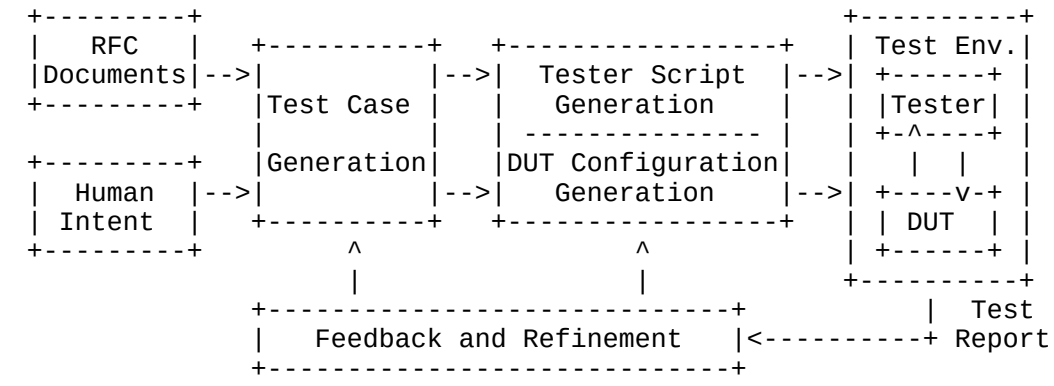
3. Test Cases: Protocol test cases may cover various categories, including protocol conformance tests, functional tests, and performance tests, etc. Each test case typically includes essential elements such as test topology, step-by-step procedures, and expected results. A well-defined test case also includes detailed configuration parameters.

4. DUT Configuration: Before executing a test case, the DUT must be initialized with specific configurations according to the test case requirements (setup). Throughout the test, the DUT configuration may undergo multiple modifications as dictated by the test scenario. Upon test completion, appropriate configurations are usually applied to restore the DUT to its initial state (teardown).

5. Tester Configuration and Execution Scripts: In test scenarios involving protocol testers, the tester often plays the active role by generating test traffic and orchestrating the test process. This requires the preparation of both tester-specific configurations and execution scripts. Tester scripts are typically designed in coordination with the DUT configurations to ensure proper interaction during the test.

5. Automated Network Protocol Test Framework

A typical network protocol test automation framework is as follows.



5.1. Protocol Understanding

Protocol understanding forms the foundation for automated test case generation. Since protocol specifications are typically written in natural language, it is necessary to model the core functionalities of the protocol and extract a machine-readable representation. This process involves identifying key behavioral semantics and operational logic from the specification text.

In addition to high-level functional modeling, structured data extraction of protocol details, such as packet field definitions, state machines, and message sequences, is also an essential component of protocol understanding. These structured representations serve as a blueprint for downstream tasks, enabling accurate and comprehensive test case synthesis based on the intended protocol behavior.

5.2. Test Case Generation

Once a machine-readable protocol specification is available, the next step is to identify test points based on human intent and extend them into concrete test cases. Test points are typically derived from constraints described in the protocol specification, such as the correctness of packet processing logic or the validity of protocol state transitions. Each test case elaborates on a specific test point and includes detailed test procedures and expected outcomes. It may also include a representative set of test parameters (e.g., frame lengths) to ensure coverage of edge conditions. Conformance test cases are generally categorized into positive and negative types. Positive test cases verify that the protocol implementation correctly handles valid inputs, while negative test cases examine how the system responds to malformed or unexpected inputs.

The quality of generated test cases is typically evaluated along two primary dimensions: correctness and coverage. Correctness assesses whether a test case accurately reflects the intended semantics of the protocol. Coverage evaluates whether the test suite exercises all protocol definitions and constraints, including performance-related behaviors and robustness requirements. However, as test cases are often represented using a mix of natural language, topology diagrams, and configuration snippets, their inherent ambiguity makes systematic quality evaluation difficult. Effective metrics for test case quality assessment are still lacking, which remains an open research challenge.

5.3. Test Script and DUT Configuration Generation

Test cases are often translated into executable scripts using available API documentation and runtime environments. This process requires mapping natural language described test steps to specific function calls and configurations of test equipment and DUTs.

Since tester scripts and DUT configuration files are typically used together, they must be generated in a coordinated manner rather than in isolation. The generated configurations must ensure mutual interoperability within the test topology and align with the step-by-step actions defined in the test case. This includes setting compatible protocol parameters, interface bindings, and execution triggers to facilitate correct protocol interactions and achieve the intended test objectives.

Before deploying the test scripts and corresponding configurations, it is essential to validate both their syntactic and semantic correctness. Although the protocol testing environment is isolated from production networks and thus inherently more tolerant to failure, invalid scripts or misconfigured devices can still render test executions ineffective or misleading. Therefore, a verification step is necessary to ensure that the generated artifacts conform to the expected syntax of the execution environment and accurately implement the intended test logic as defined by the test case specification.

5.4. Test Case Execution

The execution of test cases involves the automated deployment of configurations to the DUT as well as the automated execution of test scripts on the tester. This process is typically carried out in batches and requires a test case management system to coordinate the workflow. Additionally, intermediate configuration updates during the execution phase may be necessary and should be handled accordingly.

5.5. Report Analysis, Feedback and Refinement

Test reports represent the most critical output of a network protocol testing workflow. They typically indicate whether each test case has passed or failed and, in the event of failure, include detailed error information specifying which expected behaviors were not satisfied. These reports serve as an essential reference for device improvement, standard compliance assessment, or procurement decision-making.

However, due to the potential inaccuracies in test case descriptions, generated scripts or device configurations, a test failure does not always indicate a protocol implementation defect. Therefore, failed test cases require further inspection using execution logs, diagnostic outputs, and relevant runtime context. This motivates the integration of a feedback and refinement mechanism into the framework. The feedback loop analyzes runtime behaviors to detect discrepancies that are difficult to identify through static inspection alone. This iterative refinement process is necessary to improve the reliability of the automated testing system.

6. Automation Maturity Levels in Network Protocol Testing

To describe the varying degrees of automation adopted in protocol testing practices, we define a six-level maturity model. These levels reflect technical progress from fully manual testing to self-optimizing, autonomous systems. The classification is intended as a reference model, not as a fixed pipeline structure.

Level	RFC Interpretation	Test Asset Generation & Execution	Result Analysis & Feedback	Human Involvement
L0	Manual reading	Fully manual scripting and CLI-based execution	Manual observation and logging	Full-time intervention
L1	Human-guided parsing tools	Script templates with tool-assisted execution	Manual review with basic tools	High (per test run)
L2	Template-based extraction	Basic autogen of config & scripts for standard cases	Rule-based validation with human triage	Moderate (Manual correction and tuning)
L3	Rule-based semantic parsing	Parameterized generation and batch orchestration	ML-assisted anomaly detection	Supervisory confirmation
L4	Structured model interpretation	Objective-driven synthesis with end-to-end automation	Correlated failure analysis and report generation	Minimal (strategic input)
L5	Adaptive protocol modeling	Self-adaptive generation and self-optimizing execution	Predictive diagnostics and remediation proposals	None (optional audit)

Table 1: Automation Maturity Matrix for Network Protocol Testing

As shown in Table 1, the automation progression can be characterized along four dimensions: specification interpretation, test orchestration, result analysis, and human oversight. Each level reflects an increasing degree of system autonomy and decreasing human involvement.

6.1. L0: Manual Testing

Description: All testing tasks are performed manually by test engineers.

Key Characteristics:

* Protocol understanding, test case design, topology setup, scripting, execution, and result analysis all rely on manual work.

* Tools are only used for basic assistance (e.g., packet capture via Wireshark).

6.2. L1: Tool-Assisted Testing

Description: Tools are used to assist in some testing steps, but the core logic is still human-driven.

Key Characteristics:

* Automation includes test script execution and automated result comparison.

* Manual effort is still required for test case design, topology setup, and exception analysis.

6.3. L2: Partial Automation

Description: Basic test case generation and execution are automated, but critical decisions still require human input.

Key Characteristics:

* Automation includes:

  - A framework that generates basic test cases (e.g., covering mandatory fields and FSMs defined in RFCs) and corresponding tester scripts and DUT configurations.

  - Topology generation for a single test case.

* Manual effort includes:

  - Designing complex or edge case scenarios.

  - Root cause analysis when tests fail.

6.4. L3: Conditional Automation

Description: The system can autonomously complete the test loop, but relies on human-defined rules and constraints.

Key Characteristics:

* Automation includes:
  - Complex test cases generation based on semantic understanding of RFCs (e.g., core function modeling from RFCs).
  - Minimal common topology synthesis for a set of test cases.
  - Automated result analysis with anomaly detection.
* Manual effort includes:
  - Reviewing the test plan and confirming whether flagged anomalies represent real protocol violations.

6.5. L4: High Automation

Description: Full automation of the testing pipeline, with minimal human involvement limited to high-level adjustments.

Key Characteristics:

* Automation includes:
  - End-to-end automation from RFC parsing to test report generation.
  - Automated result analysis with root cause analysis.
  - Automated recovery from environment issues.
* Manual effort includes:
  - Defining high-level test objectives, with the system decomposing tasks accordingly.

6.6. L5: Full Automation

Description: Adaptive testing, where the system independently determines testing strategies and continuously optimizes coverage.

Key Characteristics:

* Automation includes:

  - Learning protocol implementation specifics (e.g., proprietary extensions) and generating targeted test cases.

  - Leveraging historical data to predict potential defects.

  - Iterative self-optimization to improve efficiency.

* Manual effort: None. The system autonomously outputs a final compliance report along with remediation suggestions.

7. An Example of LLM-based Automated Network Protocol Test Framework (From L2 to L3)

The emergence of LLMs has significantly advanced the degree of automation achievable in network protocol testing. Within the proposed framework, LLMs serve as central agents in multiple stages of the pipeline, enabling a transition from Level 2 (Partial Automation) to Level 3 (Conditional Automation). By leveraging LLMs with domain-specific prompting strategies, the system is able to extract, synthesize, and refine testing artifacts with minimal human intervention.

At the protocol understanding stage, LLMs can ingest RFC documents and identify structured components such as protocol field definitions, message formats, and finite state machines. These outputs, often difficult to parse using traditional rule-based methods due to inconsistencies in document structure, are extracted by prompting the LLM with carefully designed templates. The resulting structured data serves as the semantic backbone for subsequent test case generation.

Based on the extracted protocol semantics, LLMs are also capable of generating test scenarios that aim to exercise key protocol behaviors. These scenarios are proposed in natural language and then instantiated into formal test cases through alignment with predefined schemas. The use of LLMs allows for dynamic adaptation, enabling the test generation process to generalize across different protocol families while maintaining context awareness. Although the coverage achieved is not exhaustive, the LLM is often effective at identifying common corner cases and expected failure modes, especially when guided by selected examples or constraint-based instructions.

For the script generation phase, LLMs assist in mapping abstract test cases to executable configurations/scripts for both the DUT/tester. Given appropriate configuration/API documentation and environment context, the model synthesizes CLI commands, configuration snippets,

and tester control scripts (e.g., Python-based test harnesses or TCL scripts for commercial testbeds). This process is enhanced by retrieval-augmented generation, where relevant code examples or past case patterns are integrated into the model's prompt to increase the reliability and correctness of the output.

Finally, during the feedback and refinement loop, LLMs analyze execution logs and output discrepancies to identify failure points. While current models may not fully replace human diagnosis, they can summarize error types, hypothesize root causes, and suggest concrete revisions to test cases or script logic. These suggestions are then validated by experts before being applied, ensuring both efficiency and correctness.

Despite these capabilities, it is important to note that LLMs are fundamentally probabilistic and lack strong guarantees of determinism or correctness. Therefore, while the LLM-enabled system can execute a full test pipeline with reduced human effort, human oversight remains essential, particularly for verifying generated artifacts and handling ambiguous or novel scenarios. Nonetheless, the integration of LLMs marks a clear advance beyond traditional script-driven automation tools, and provides a practical path for elevating protocol testing practices toward Level 3 maturity.

8. Security Considerations

1. Execution of Unverified Generated Code: Automatically generated test scripts or configurations (e.g., CLI commands, tester control scripts) may include incorrect or harmful instructions that misconfigure devices or disrupt test environments. Mitigation: All generated artifacts should undergo validation, including syntax checking, semantic verification against protocol constraints, and dry-run execution in sandboxed environments.

2. AI-Assisted Component Risks: LLMs may produce incorrect or insecure outputs due to their probabilistic nature or prompt manipulation. Mitigation: Apply input sanitization, prompt hardening, and human-in-the-loop validation for critical operations.

9. IANA Considerations

This document has no IANA actions.

Acknowledgments

This work is supported by the National Key R&D Program of China.

Contributors

Zhen Li
Beijing Xinertel Technology Co., Ltd.
Email: lizhen_fz@xinertel.com

Zhanyou Li
Beijing Xinertel Technology Co., Ltd.
Email: lizy@xinertel.com

Authors' Addresses

Yong Cui
Tsinghua University
Email: cuiyong@tsinghua.edu.cn

Yunze Wei
Tsinghua University
Email: wyz23@mails.tsinghua.edu.cn

Kaiwen Chi
Tsinghua University
Email: ckw24@mails.tsinghua.edu.cn

Xiaohui Xie
Tsinghua University
Email: xiexiaohui@tsinghua.edu.cn