Network Management Research Group
Internet-Draft
Intended status: Informational
Expires: 26 August 2026

Y. Cui
Y. Wei
K. Chi
X. Xie
Tsinghua University
22 February 2026

Framework and Automation Levels for AI-Assisted Network Protocol Testing
draft-cui-nmrg-auto-test-01

Abstract

This document presents an AI-assisted framework for automating the testing of network protocol implementations. The proposed framework consists of components such as protocol formalization, test case generation, test script and configuration generation, and iterative refinement through feedback mechanisms. In addition, the document defines a set of Automation Maturity Levels for network protocol testing, 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.

The latest revision of this draft can be found at https://example.com/LATEST. 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@ietf.org), which is archived at https://datatracker.ietf.org/rg/nmrg/. Subscribe at https://www.ietf.org/mailman/listinfo/nmrg/.

Source for this draft and an issue tracker can be found at https://github.com/USER/REPO.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

As protocol specifications evolve at an increasing pace, traditional testing approaches that rely heavily on manual effort or protocol-specific models struggle to keep up. Protocol testing aims to validate whether a device's behavior conforms to the semantics defined by the protocol, which are typically specified in RFC documents. In recent years, emerging application domains, including the industrial Internet, low-altitude economy, modern datacenter networks, and satellite Internet, have further accelerated the emergence of proprietary or rapidly evolving protocols. This trend significantly exacerbates the difficulty of achieving comprehensive and timely protocol testing.

This document proposes an automated network protocol testing framework designed to reduce manual effort, improve test coverage, and adapt efficiently to evolving specifications. The framework consists of four key modules: protocol formalization, test case generation, test script and configuration generation, and feedback-based refinement. It emphasizes modularity, reuse of existing protocol knowledge, and AI-assisted processes to enable accurate, scalable, and maintainable protocol testing.

In addition, this document introduces six Automation Maturity Levels (Levels 0-5) to characterize the maturity of automation in network protocol testing. These levels serve as a technology roadmap that helps researchers and practitioners assess the current capabilities of their testing systems and identify directions for future improvement. Each level captures progressively stronger capabilities in protocol formalization, orchestration, analysis, and independence from human intervention.

2. Definition and Acronyms

DUT: Device Under Test

Tester: A network device implementing multiple network protocols to support protocol conformance and performance testing. It generates test-specific packets or traffic, emulates target network behaviors, and analyzes received packets to evaluate protocol compliance and performance.

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.

Tester Script: An executable program or sequence of instructions that controls a protocol tester to generate test traffic, interact with the DUT according to a specified test case, and collect relevant observations for result evaluation.

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. 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.

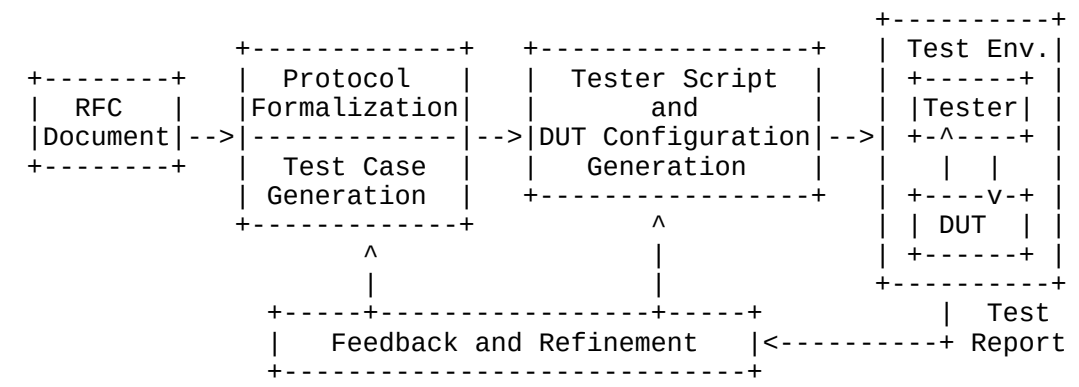
4. Test Topology: Each test case must specify the network topology it requires. Before executing a test case, the corresponding topology must be established accordingly. In a batch testing scenario, frequent changes in topology can be time-consuming and inefficient. To mitigate this overhead, it is common practice to construct a minimal common topology that satisfies the requirements of all test cases in a given batch. This minimizes the number of devices and links needed while ensuring that each test case can be executed within the shared topology.

5. 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).

6. Tester Configuration and 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 illustrated as follows.



5.1. Protocol Formalization

Protocol formalization forms the foundation for automated test case generation. Since protocol specifications are typically written in natural language, this step transforms unstructured text into a structured, machine-interpretable representation that can be traversed, queried, and validated by downstream tasks.

To enable a multi-dimensional characterization of protocol semantics, we formalize protocol content along two complementary dimensions: _Basic_ and _Logic_. Basic formalization captures the static structure and execution context of a protocol, including message formats (e.g., fields and constraints), local data structures (e.g., timers and variables), and state machines that define legal state spaces and transitions. Logic formalization captures operational semantics and behavioral constraints, including event-action rules, protocol algorithms, and error handling behaviors. In practice, effective formalization also needs to explicitly encode relationships across these basic and logical elements, such as which messages trigger particular state transitions or processing rules, so that test generation can reason across modules consistently.

5.2. Test Case Generation

Once a machine-readable protocol representation is available, the next step is to identify test points from protocol requirements and behavioral constraints, and extend them into concrete test cases. Test points can be derived from normative statements, message format constraints, packet processing logic, and valid/invalid 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., boundary values and invalid values) to improve 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 protocol definitions and constraints across multiple testing dimensions (e.g., conformance, robustness, performance, and security) and explores representative parameter spaces. 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. Tester 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 tester scripts and corresponding DUT 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 set of Automation Maturity Levels. These levels reflect technical progress from fully manual testing to self-optimizing, autonomous systems. These Automation Maturity Levels are intended as a reference model, not as a fixed pipeline structure.

Level	RFC Interpretation	Test Asset Generation & Execution	Result Analysis & Feedback	Human Involvement
0	Manual reading	Fully manual scripting and CLI-based execution	Manual observation and logging	Full-time intervention
1	Human-guided parsing tools	Script templates with tool-assisted execution	Manual review with basic tools	High (per test run)
2	Template-based extraction	Basic autogen of config & scripts for standard cases	Rule-based validation with human triage	Moderate (Manual correction and tuning)
3	Rule-based semantic parsing	Parameterized generation and batch orchestration	ML-assisted anomaly detection	Supervisory confirmation
4	Structured model interpretation	Objective-driven synthesis with end-to-end automation	Correlated failure analysis and report generation	Minimal (strategic input)
5	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 Maturity Levels are characterized along four dimensions: RFC interpretation, test asset generation and execution, result analysis, and human involvement. Each level reflects an increasing degree of system autonomy and decreasing human involvement.

6.1. Level 0: 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. Level 1: 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. Level 2: 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 performs basic protocol formalization (e.g., extracting fields, message formats, and FSM fragments) and generates baseline test cases and corresponding tester scripts and DUT configurations for standard cases.

  - Topology generation for a single test case.

* Manual effort includes:

  - Designing complex or edge case scenarios.

  - Root cause analysis when tests fail.

6.4. Level 3: 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 case and parameter generation based on semantic understanding and formalization of RFCs (e.g., structured protocol modules and behavioral constraints).

  - Minimal common topology synthesis for a set of test cases.

  - Automated result analysis with anomaly detection and iterative refinement driven by execution feedback.

* Manual effort includes:

  - Reviewing the test plan and confirming whether flagged anomalies represent real protocol violations.

6.5. Level 4: 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. Level 5: 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 Level 2 to Level 3)

The emergence of LLMs has significantly advanced the degree of automation achievable in network protocol testing. Within the proposed framework, LLMs can serve as core components in multiple stages of the testing pipeline, enabling a transition from Level 2 (Partial Automation) to Level 3 (Conditional Automation). A key enabler is to introduce an explicit protocol formalization step that transforms unstructured RFC text into a structured, machine-interpretable intermediate representation (e.g., a protocol description spanning message formats, state machines, and normative behavioral constraints). With such a representation, downstream generation becomes more systematic and less dependent on ad-hoc, protocol-specific parsers.

At the protocol formalization stage, LLMs can enrich RFCs with structured signals, such as section-level summaries, cross-references across documents, and normative requirement statements (e.g., "must" and "SHOULD"). The agent can further induce protocol modules (e.g., message formats, state machines, event-action rules, and algorithms) and formalize them into a unified representation that supports traversal and query. This representation serves as the semantic backbone for test generation, and it also helps in update scenarios by localizing changes and propagating them to the corresponding formal modules.

Based on the formalized protocol representation, LLMs can generate test cases in a more structured manner by decoupling test case templates from test parameters. The template generation step expands extracted test points into parameterized templates that define test objectives, topology, execution steps, oracles, and static testbed configurations. The parameter instantiation step then populates template placeholders with concrete values, including representative boundary values and invalid values for robustness testing. When oracle values require computation, the system can synthesize small helper programs (e.g., Python scripts) to compute expected outcomes, and can apply equivalence partitioning to reduce redundant parameter combinations without sacrificing meaningful coverage.

For test execution, LLMs can assist in translating abstract test procedures into executable artifacts for both the tester and the DUT. In practice, this translation is often a multi-step workflow that benefits from a structured agent architecture. For example, a core agent can orchestrate the artifact generation process, while specialized sub-agents handle documentation summarization, intent rewriting (turning high-level test objectives into API-aligned actions), and recurring fault fixing based on an experience pool. During execution, feedback from logs and device outputs can be used to iteratively refine generated artifacts, and an adaptive pruning mechanism can decide whether to stop exploring additional parameter instances for a given template when definitive failures are found or sufficient coverage has been achieved.

Despite these capabilities, it is important to note that LLMs are fundamentally probabilistic and cannot guarantee determinism or correctness. Therefore, even when the framework can complete an automated loop with reduced human effort, human oversight remains valuable for validating critical intermediate artifacts (e.g., formalized protocol modules, test oracles, and high-impact configuration changes) and for handling ambiguous or novel protocol behaviors. Nevertheless, integrating LLMs with explicit protocol formalization, systematic template/parameter generation, and execution-time feedback 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