RFC2TLA+: Extracting and Verifying Formal Models from RFC Documents using Continuous LLM Feedback

Guozhen Ding
University of Toronto
Toronto, Ontario, Canada
gzh.ding@mail.utoronto.ca

Kexin Li
University of Toronto
Toronto, Ontario, Canada
cassiekx.li@mail.utoronto.ca

Ilya Grishchenko
University of Toronto
Toronto, Ontario, Canada
ilya.grishchenko@utoronto.ca

David Lie
University of Toronto
Toronto, Ontario, Canada
david.lie@utoronto.ca

Abstract

Extracting natural-language protocol specifications such as Requests for Comments (RFCs) into formally verifiable models is a challenging and labor-intensive process that requires significant expertise. Recent advances in large language models (LLMs) and agents show promise in automating this task. However, existing automated approaches suffer from insufficient structural consistency, hallucinations, and limited scalability. Moreover, the ability to pass such automatically extracted models into formal verification tools such as model checkers has not been demonstrated.

We present RFC2TLA+, a three-stage pipeline for automatically extracting RFCs into automatically verifiable models. RFC2TLA+ achieves this by providing fine-grain and continuous feedback to the LLM throughout the extraction process. It decomposes the extraction process into three passes—states, transitions, and invariants—while constraining LLM generation through fine-grained tool calls, lightweight validation, and iterative feedback. This design enables early error detection and correction, mitigating hallucinations and preventing syntactic and semantic inconsistencies. The resulting communicating finite-state machines (CFSMs) are deterministically translated into TLA + and verified using the TLC model checker.

We evaluate RFC2TLA+ on a benchmark of 14 RFCs and their accompanying human-crafted FSMs and show that it generates more complete models than the baseline. In addition, our tool extracts invariants, enabling a model checker to detect injected faults. To minimize the effects of LLM memorization, we additionally generate synthetic RFCs and evaluate the tool on them, demonstrating its effectiveness in both model extraction and error detection.

CCS Concepts

• **Software and its engineering** → *Formal software verification*; • **Computing methodologies** → **Machine learning**.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837453

Keywords

Artificial intelligence and machine learning for software engineering, Model Checking

ACM Reference Format:
Guozhen Ding, Kexin Li, Ilya Grishchenko, and David Lie. 2026. RFC2TLA+: Extracting and Verifying Formal Models from RFC Documents using Continuous LLM Feedback. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3832783.3837453

1 Introduction

Modern computing systems depend on a diverse ecosystem of network protocols, each governed by a specification document maintained by the Internet Engineering Task Force. These specifications, namely *Requests for Comments (RFCs)*, define the behaviors that implementations must follow. However, RFCs, like any other specification, can contain defects [26, 27, 34].

Adopting formal methods to identify problems in protocol specifications is hindered by *specification lifting*—the extraction of natural-language specifications into formal representations such as Finite State Machine (FSM) models. This process is difficult: it requires expertise in both the source specification and the formal language and is highly labor-intensive, given the length and complexity of documents like RFCs.

Large Language Models (LLMs) have demonstrated promising potential in bridging this gap, as they show impressive capability in generating structured representations such as code from natural language instructions. However, even with LLM assistance, producing high-quality formal models remains challenging. Specifically, recent benchmarks [31] demonstrate that LLM-generated FSMs lack the quality of manually constructed models. Several reasons can lead to LLM's poor performance. For instance, LLMs are prone to hallucinations and semantic shifts. Existing LLM-based approaches try to mitigate these limitations. However, these approaches only handle short specifications and lack comprehensive evaluation on larger RFCs [30]. Importantly, they also do not support automation, as they neither produce encodings compatible with modern model checkers [5, 16, 35] nor extract protocol-specific invariants required by these checkers to define properties for verification. Further, many of them focus on extracting models from code rather than natural language, which is an easier task [20, 25].

To address the limitations of LLMs and to bridge the gaps in existing LLM-based extraction methods, we propose RFC2TLA+, a tool-enabled pipeline for automatically lifting RFCs into formally verifiable FSM specifications in TLA + format supported by a modern model checker TLC [35]. While mechanisms such as text segmentation and retrieval-augmented generation (RAG) can help LLMs handle very large RFCs, we find that they are insufficient, as these mechanisms are unaware of the formal specification requirements. In particular, FSMs are composed of states, transitions, and invariants, while RFCs tend to specify behavior by function. As a result, to enable effective lifting, RFC2TLA+ decomposes generation into three passes—state, transition, and invariant extraction—each supported by dedicated tools exposed to an LLM. The intuition behind this design is that formal model extraction should respect key dependencies: transition extraction relies on a well-defined state space, while invariant extraction requires an accurate FSM to be established. Another key insight of our design is that to mitigate hallucinations and errors of LLM outputs and extract high-quality formal models, *frequent verification and immediate correction* of intermediate LLM outputs is crucial. Rather than treating model extraction as a one-shot process followed by validation, RFC2TLA+ continuously checks partial outputs (e.g., states, transitions, and invariants) using lightweight syntactic and semantic validation, as well as execution through the underlying model checker. By incorporating early, iterative feedback, our pipeline can detect errors as they arise and guide the LLM towards valid constructions.

Invariant identification is explicitly incorporated in our design, enabling automated model property checking and protocol behavior validation of the extracted models. Specifically, invariant extraction enables us to verify the correctness of the extracted FSM and invariants by checking them with a model checker (e.g., TLC). In contrast, prior work [31] typically rely on structural or semantic matching against reference models and do not provide an integrated workflow that jointly extracts protocol-specific invariants and performs checker-driven validation of the extracted behavior model.

To this end, we make the following contributions:

- We present RFC2TLA+, the first tool-augmented pipeline for extracting natural-language RFCs into formally verifiable TLA + specifications. By combining segmentation, RAG, structured three-stage tool interfaces, and iterative feedback, RFC2TLA+ constrains LLM outputs while preserving traceability to the source text.
- We evaluate model completeness on PSMBench [31], where our approach outperforms the prior method.
- We study 14 protocols ranging from 47K to 746K characters, successfully generating models in 36 out of 42 runs. By injecting deadlocks and protocol-specific invariant violations, we assess invariant coverage, detecting 69 out of 84 injected flaws (33 deadlocks + 36 invariant violations).
- To mitigate LLM's training data leakage, we construct three synthetic RFCs with injected bugs. RFC2TLA+ detects 16 out of 18 of these bugs.
- We also evaluate RFC2TLA+'s sensitivity to the choice of LLM and conduct an ablation study to assess each component's contribution.

2 Challenges

Formal verification of network protocols requires translating natural-language specifications into model-checkable representations. This *specification extracting* process involves (1) identifying all protocol states across communicating entities, (2) extracting every message exchange as transitions, and (3) deriving invariants from normative language such as "must" and "shall". The process is tedious even for short specifications and prohibitively expensive for large ones: TCP spans 85 pages, BGP exceeds 100 pages, and some RFCs surpass 1.5 million characters [21].

Even carefully constructed manual models are not immune to these difficulties. PSMBench [31], the most recent benchmark for protocol FSM extraction, provides hand-crafted Protocol State Machines (PSMs) of 14 protocols, which are intended by the authors to be used as a benchmark to measure the accuracy of formal specifications extracted by LLMs. However, our analysis of several of these models has found that some of them are incomplete or structurally misaligned with the RFCs. For instance, in TCP (RFC 9293 [13]), the ground truth captures 11 states and 20 transitions but misses the transition `FIN_WAIT_1` $\xrightarrow{\texttt{rcv FIN+ACK}}$ `TIME_WAIT`, which handles simultaneous connection teardown. The RFC text for this case reads: *"If our FIN has been ACKed (perhaps in this segment), then enter TIME-WAIT … otherwise enter the CLOSING state."* In another example, the hand-crafted models for NNTP (RFC 3977) [14], specify the protocol as a sequence of dependent operations (`Group_Selected` → `Article_Selected` → `Reading_Headers`). However, the RFC states not only that the messages can be sent in any order, but that other messages, missing from this sequence, can also be sent. Moreover, the RFC also states instances where the server should reject incorrect sequences with an error message that are not in the human-crafted model. These examples show that manual specification lifting is not only labor-intensive but also error-prone, even when performed with extra care.

Existing automated approaches attempt to overcome manual extraction difficulties but fall short in quality and capability. RFC-NLP [24] struggles with structural complexity, producing incomplete models. LLM-based PROSPER [30] handles only small specifications and requires human-in-the-loop guidance. Li et al. [19] propose a two-stage annotation-then-conversion method (also LLM-based) for extracting temporal logic specifications from software documents, achieving 71.6% accuracy on short documents (average 2,966 words); however, their approach targets individual temporal logic formulas rather than multi-entity protocols, operates on documents orders of magnitude smaller than typical RFCs, and does not support model checking or invariant verification. Critically, none of these approaches extracts invariants from RFC text and therefore do not support end-to-end translation from natural-language documents to formally verifiable models. In summary, we identified three key challenges in automated LLM-based specification lifting:

(C1) Hallucinations and structural dependencies. Current approaches are prone to generating plausible but incorrect formal constructs: hallucinated states, inconsistent naming, and fabricated transitions that do not correspond to the specification text. These errors compound across long documents. Moreover, formal model generation must respect structural dependencies, where transitions must depend on a well-defined state space, and invariants require

an established behavioral model. But unconstrained generation by current methods often ignores these ordering requirements.

(C2) No checkable invariant extraction. Without invariants extracted from the RFC text, model checking is limited to generic properties such as deadlock freedom. However, protocol-specific requirements, such as the completion of the TCP handshake preceding data transmission typically requires invariants extracted from the RFC. Prior model-extraction approaches do not extract such properties with the behavioral model jointly.

(C3) Document scale. RFC documents range from a few hundred to over 1.5 million characters. Recent approaches based on LLMs cannot reliably process specifications at the upper end of this range within a single context window, and accuracy degrades well before the hard token limit is reached.

3 Design

To address the challenges presented in §2, RFC2TLA+ introduces a structured, tool-augmented pipeline. The LLM is constrained to issue structured tool calls that incrementally build an FSM—specifically, a communicating finite-state machine (CFSM), which is used to model concurrently communicating FSMs [8]. RFC2TLA+ includes a validation layer and model checker feedback to mitigate hallucinations (**C1**). Protocol invariants are explicitly extracted and verified by the TLC model checker (**C2**). Standard operations such as segmentation and retrieval-augmented generation (RAG) are applied to handle document scale (**C3**).

These principles are realized in a three-stage pipeline (Figure 1). Given an RFC as input, RFC2TLA+ incrementally constructs a CFSM and a set of invariants, then translates them into a TLA + specification for verification. Stage I (§3.1) preprocesses the RFC through segmentation and optional anonymization and RAG. Stage II (§3.2) extracts states, transitions, and invariants, with an LLM constrained to a dedicated set of tools, 22 in total. Each tool invocation is validated before model modification. Stage III (§3.3) deterministically renders the CFSM into TLA + and runs the TLC model checker.

A crucial aspect of the design of RFC2TLA+ is to provide frequent and timely feedback to the LLM to steer it away from hallucinations and errors (**C1**). RFC2TLA+ does this with 4 distinct mechanisms (Figure 1). First, RFC2TLA+ only allows the LLM to output FSM specifications using a set of tool calls we provide, thus constraining the output space of the LLM. Second, the use of the tool calls enables us to perform inline validation of every FSM-lifting action of the LLM, ensuring that the lifted FSM is well-formed (e.g., transitions are between defined states). Third, we provide an *iteration* mechanism used during Stage II that improves extraction completeness. Finally, a final check in Stage III against the model checker identifies global inconsistencies that the LLM can address before the final formal model is produced by RFC2TLA+.

3.1 Stage I: RFC Preprocessing

Document segmentation. To handle RFCs that exceed the LLM's effective context window and encourage the LLM to build the model step by step, RFC2TLA+ splits every document into segments using sentence-level tokenization [6]. Each segment contains 30 sentences by default and is wrapped with metadata (section identifier, paragraph index) to preserve document structure during processing.

Retrieval-Augmented Generation (RAG). Protocol specifications frequently contain forward and backward references: a transition described in one section may depend on state definitions introduced pages earlier. To provide this cross-document context without requiring the full RFC in the context, RFC2TLA+ uses RAG. The RFC is uploaded to an embedding-based vector store [22], which automatically chunks and embeds the document. For each segment processed by the Transition Extraction and Invariant Extraction components, the system queries the vector store with the segment content and retrieves the top-3 semantically relevant chunks. To avoid retrieving the segment itself, a Jaccard word-overlap filter discards chunks with $\geq$50% overlap with the query. Retrieved chunks are appended to the LLM prompt as additional context.

3.2 Stage II: Tool-Augmented CFSM Synthesis

3.2.1 CFSM Construction Tools. To address (**C1**), direct free-form formal verification language generation is avoided. RFC2TLA+ treats model generation as a constrained tool-use loop. The generation is done in three passes, adding different CFSM elements each time (states, edges, and invariants). For each RFC segment and its RAG-related content, RFC2TLA+ sends to the LLM: (i) the relevant RFC text, (ii) a summary of the current protocol model, and (iii) a set of allowed actions specific to the pass.

For each pass, only the subset of actions relevant to the pass is made available to the LLM, making it focus on the corresponding construction task. We use fixed system- and user-prompt templates specific to each pass [12]. The templates specify pass objective, permitted modeling operations, naming and expression rules, and completion condition. RFC text, retrieved context, and the current CFSM summary are instantiated at runtime. After each round, the next prompt includes the updated CFSM summary and exact validation errors or a request to extract remaining elements; the LLM responds with additional actions or `DONE`. TLC-guided refinement (§3.3) uses a separate prompt containing the TLC error and counterexample, current CFSM and invariants, and relevant RFC text.

The LLM can modify the model only by invoking a small, well-defined set of CFSM construction actions implemented by RFC2TLA+ as an in-process Python library. For each action we design a custom tool defined by a name, a description, and a JSON schema. The complete set of 22 CFSM construction tools, organized into five categories, is listed in Table 1. These definitions provide a self-describing interface that the LLM accesses through function calling. To execute the action, RFC2TLA+ uses the provider's tool call ability to provide the LLM with access to a set of construction tools. Specifically, the LLM executes actions by generating *tool call requests*, each of which consists of a tool call token, followed by a JSON structure that contains the tool name and parameters for the tool call. RFC2TLA+ parses the arguments and dispatches the call to the corresponding Python routine. After execution, RFC2TLA+ returns the result or validation error to the LLM as a tool-response message associated with the original call identifier.

Each tool performs a single operation on the CFSM model, and then the *validation layer* checks every invocation before committing the changes to the CFSM. For example, to add the *Transaction* state to the Server entity, the LLM invokes `add_state` with the argument object `{"entity_name": "Server", "state_name":`

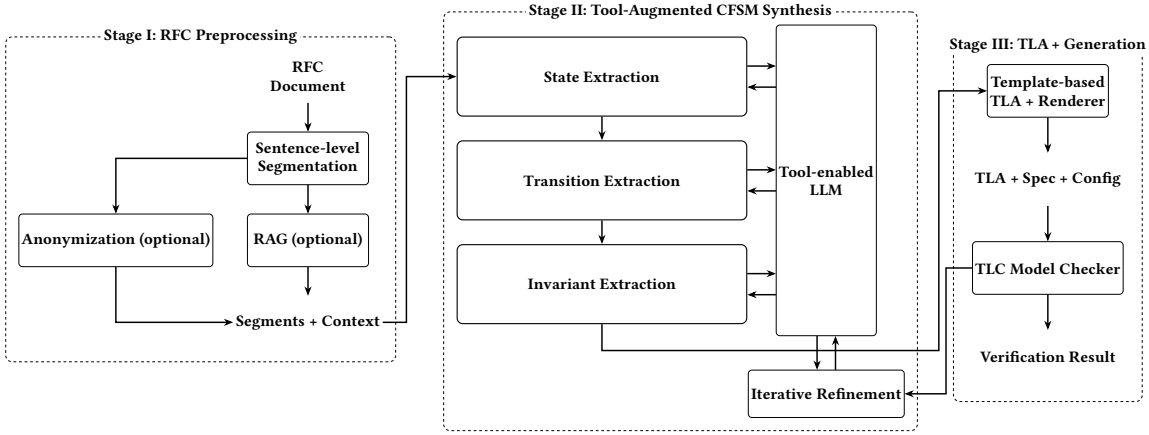

Figure 1: Overview of the RFC2TLA+ pipeline.

Table 1: Complete set of 22 CFSM construction tools exposed to the LLM via the provided function-calling interface.

Category	Tool	Description
Protocol & Entity Configuration (3)	`create_protocol`	Initialize a named protocol.
	`create_entity`	Create an FSM entity (e.g., client, server).
	`add_channel`	Establish a communication channel between two entities.
State & Variable Management (5)	`add_state`	Add a state to an entity, optionally marking it as initial or final.
	`remove_state`	Remove a state and all associated transitions and invariants.
	`add_variable`	Declare a typed variable (NATURAL, INTEGER, BOOLEAN, STRING).
	`remove_variable`	Remove a variable and all transitions that reference it.
	`change_variable_initial_value`	Modify the initial value of an existing variable.
Transition Configuration (7)	`add_internal_transition`	Add an asynchronous internal transition (e.g., timeout, local processing).
	`add_send_transition`	Add a transition that sends a message on a channel.
	`add_receive_transition`	Add a transition triggered by receiving a message from a channel.
	`replace_internal_transition`	Replace an existing internal transition (for correction).
	`replace_send_transition`	Replace an existing send transition (for correction).
	`replace_receive_transition`	Replace an existing receive transition (for correction).
	`remove_transition`	Remove a transition identified by trigger, source, and destination.
Invariant Configuration (5)	`add_state_invariant`	Define a property that must hold in a specific state.
	`add_entity_invariant`	Define a property that must hold across all states of an entity.
	`replace_state_invariant`	Replace an existing state-specific invariant (for correction).
	`replace_entity_invariant`	Replace an existing entity-wide invariant (for correction).
	`remove_invariant`	Remove a state-specific or entity-wide invariant.
Diagnostic & Bug Reporting (2)	`get_protocol_status_summary`	Return a summary of the current CFSM.
	`report_rfc_bug`	Flag a suspected specification error in the RFC text.

`"Transaction", "initial": false, "final": false}`. Our tool parses this object, validates that the referenced entity exists and that the state declaration is well-formed, and then invokes the corresponding Python routine to update the CFSM.

3.2.2 Validation Layer. Every tool invocation in Stage II is processed through a conservative set of checks that seek to prevent the LLM from introducing structural errors into the FSM lifting that can propagate to later stages: (1) Symbol declaration, ensuring that transitions only refer to previously defined states and rejecting any undeclared references; (2) Expression parsing, which validates guard conditions and variable updates using a restricted, Python-like syntax and disallows unsupported constructs; (3) Entity locality, restricting conditions to variables within the same entity and preventing cross-entity references; and (4) Variable coverage, requiring that all entity variables are accounted for in each transition, either explicitly updated or marked as unchanged, with any omissions automatically completed. Invalid tool outputs are rejected with structured feedback, prompting revision and preventing errors from propagating through the model. These checks are designed to reject only the most obvious and egregious errors. More subtle semantic or logic errors are meant to be identified during iterative refinement described in §3.3.

3.2.3 Iteration. For each state, transition, and invariant extraction pass, RFC2TLA+ uses an iteration mechanism to ensure the relevant elements (i.e., states, transitions, etc.) are extracted from the RFC segment. The LLM first extracts a set of elements from each segment, after which RFC2TLA+ iteratively feeds the cumulative set back to the LLM and asks it to identify any missing elements. It also gives the LLM an option to end the iteration by asserting that it has not missed any elements. Otherwise, if the LLM extracts more elements, RFC2TLA+ will keep iterating until a preset upper bound that depends on the element type being extracted.

3.2.4 Three-Pass Extraction Architecture. To mitigate LLM hallucination and preserve CFSM's structural dependencies, a crucial design choice in RFC2TLA+ is to decompose CFSM construction into three sequential passes during the generation stage, each supported by specialized tools. This decomposition aligns with the inherent dependencies in formal modeling: transitions depend on a well-defined state space, and invariants depend on a consistent model of system behavior.

State extraction. The system first identifies protocol entities and states from the RFC. The LLM uses Protocol & Entity Configuration operations and State & Variable Management operations (Table 1) to construct the structural backbone of the CFSM, including entities, local state spaces, communication channels, and typed variables. By separating this stage, RFC2TLA+ ensures subsequent reasoning operates over an explicit and validated state space.

For example, when processing POP3 (RFC 1939 [28]), the POP3 RFC states: *"When a client host wishes to make use of the service, it establishes a TCP connection with the server host."* From this, the LLM calls `create_protocol` and `create_entity` to initialize the protocol with Client and Server entities, `add_channel` to establish bidirectional communication channels between them. A later segment states: *"A POP3 session progresses through a number of states during its lifetime. Once the TCP connection has been opened and the POP3 server has sent the greeting, the session enters the AUTHORIZATION state.* [...] *the session enters the TRANSACTION state."* The LLM then calls `add_state` to register *Authorization* (marked as the initial state) and *Transaction* state.

Transition extraction. Variables required by transition guards and updates are declared during this pass using the variable-management tools. Given the extracted states, the LLM incrementally generates transitions that capture protocol behavior using Transition Configuration operations (Table 1). These include internal transitions as well as message-passing interactions between entities. For each segment, the LLM receives the current protocol status summary (all entities, states, transitions, and variables accumulated so far) along with the segment text and optional RAG context. Each transition is extracted through tool calls that specify guards, updates, and message structures. This stage enforces consistency with the previously defined states. Each message exchange in the RFC is modelled as a pair of transitions: one entity sends a message on a channel, and the other receives it. Internal state changes that do not involve communication are captured as internal transitions. All transitions must reference declared states, and the validation layer (§3.2.2) rejects any undeclared references. The LLM processes up to 20 iterations per segment, and the cumulative context is maintained across segments until we exhaust the context window, so the LLM can reference earlier RFC processing.

Continuing the POP3 example, the RFC describes the authentication flow: *"[...]the client must first issue the USER command. "* The LLM models the "USER command" transition with several tool calls, including Client sending a `USER` message via `add_send_transition` call, and Server receiving the message is modelled via `add_receive_transition` tool call. These transitions do not alter Client and Server states, but they are part of the sequence that ultimately allows them to switch from Authorization to the Transaction state.

Invariant extraction. The system extracts invariants specific to the protocol. These include both global invariants and state-specific constraints. The available tools are restricted to Invariant Configuration operations (Table 1). By deferring invariant generation until after the CFSM is constructed, RFC2TLA+ enables the LLM to ground properties in a concrete behavioral model, improving both relevance and correctness. This pass runs for up to 10 iterations. For POP3, the RFC states that for the client to have transactions with the server: *"The client must now identify and authenticate itself to the POP3 server."* To extract the invariant, the LLM calls `add_state_invariant` tool. This call associates the condition `authenticated == True` with the *Transaction* state to indicate that whenever the Server is in Transaction, the `authenticated` variable must be true.

Together, these three stages constrain the generation process, reduce ambiguity, and improve the completeness of the resulting formal specification. Note that the tool providing a summary of the current CFSM can be used by any stage (see Table 1, Diagnostic & Bug Reporting, first operation).

3.3 Stage III: TLA + Generation, Model Checking and Iterative Refinement Loop

After Stage II produces a CFSM with invariants, RFC2TLA+ deterministically translates it into a TLA + specification using a fixed Jinja2 [32] template. The template maps each CFSM construct to its TLA + counterpart: entity states become a state function over machines, channels become sequences of message records, send transitions append to the appropriate channel, receive transitions consume from the head, and guards/updates are translated from the validated expression internal representation into TLA + operators. Because the CFSM formalism is constrained—entities have finite local states, communicate via typed FIFO channels, and use a fixed set of variable types—a single template suffices to cover all protocols without requiring protocol-specific TLA + generation logic. Furthermore, our deterministic translation reduces syntactic errors caused by LLM hallucination, such as operator misuse, parenthesization errors, and formatting inconsistencies. As a part of deterministic translation, we also encode the extracted invariants.

The rendered specification and a TLC configuration file (including invariants to check and the state space bounds) are passed to the TLC model checker, which exhaustively explores the reachable state space and checks for deadlocks and invariant violations. If TLC reports an error, the counterexample trace is fed back to the LLM along with the relevant CFSM context and source text. The LLM then refines the model using the same tool interface, for up to 3 iterations. In the case that LLM determines that the counterexample is triggered by the RFC itself instead of a modelling error, it

calls to `report_rfc_bug` tool (Table 1, last row), to terminate the refinement loop early and report an error.

4 Evaluation

We evaluate RFC2TLA+ along four research questions:

- **RQ1** How does RFC2TLA+ compare to existing LLM-based FSM extraction methods?
- **RQ2** Can RFC2TLA+ generate verifiable TLA + specifications, and can model checking of the generated specifications detect injected protocol defects?
- **RQ3** How sensitive is RFC2TLA+ to the choice of LLM?
- **RQ4** How do RAG, iterative refinement, constrained tool use, and deterministic tool-call validation contribute to RFC2TLA+'s effectiveness?

Dataset. We evaluate on PSMBench [31], which contains human-crafted FSMs for 14 RFCs spanning a wide range of sizes, domains, and complexities. The specifications range from 47K characters (POP3, RFC 1939) to 746K characters (RTSP, RFC 7826). For bug detection experiment (RQ2), we generate two buggy variants for each RFC using automated fault injection (§4.2.1), yielding 28 RFCs.

Setup. We use GPT-5.1 as the underlying LLM. All runs use the full three-stage pipeline with segmentation and RAG enabled. Each RFC is anonymized (§3.1) to discourage the LLM from relying on memorized protocol knowledge. Segments are configured with a default size of 30 sentences. We run TLC 2.20 for model checking.

As an additional precaution, we note that well-known protocols (e.g., POP3, FTP, TCP) appear extensively in LLM training data, raising the concern that the model may recall memorized protocol behavior rather than faithfully extracting from the input text. To mitigate this, we strip identifying metadata from the RFC: the RFC number is replaced with "RFC XXXX," centered title lines are replaced with a generic placeholder, and protocol abbreviations in page headers are masked. The body text is preserved intact so that the LLM can still extract protocol semantics.

4.1 RQ1: Model Extraction Quality

For RQ1, we adopt the PSMBench evaluation protocol [31], which computes precision (P), recall (R), and F1-score (F1) for both states and transitions against manually curated ground-truth FSMs. For baseline, we use RFC2PSM [31], the FSM generation mechanism introduced together with the benchmark, which directly produces single-entity FSMs (the authors adopted the term Protocol State Machine or PSM). To compare the extracted FSM with the ground truth, we use PSMBench's standard semantic fuzzy matching with a cosine similarity threshold of 0.5.

Since RFC2TLA+ produces multi-entity CFSMs (e.g., client and server) and contains states and transitions that use substantially different names from the single-entity ground truth FSMs in PSMBench (e.g., entity prefixes, granularity, abbreviations, or synonym actions), standard string-based or embedding-based matching is insufficient; we perform the following automatic conversion for each entity. We use a *pairwise bidirectional LLM matching* procedure that leverages a less capable LLM (GPT-4.1-mini) as a semantic judge, allowing one-to-many and many-to-one mappings between extracted elements and the ground-truth. The LLM is provided with an explanation that entity-prefixed states might map to bare ground-truth names (e.g., `Server_Established` corresponds to `Established`), and that synthetic states such as `Done` or `Start` may correspond to `Closed` or `Idle` in the ground truth. For each extracted element (state or transition), the *forward pass* asks the LLM which ground-truth element(s) it matches, if any. Symmetrically, the *reverse pass* asks, for each ground-truth element, which extracted element(s) match it. Precision is computed from the forward pass as the fraction of extracted elements that matched at least one ground-truth element. Recall is computed from the reverse pass as the fraction of ground-truth elements that matched at least one extracted element. The bidirectional design prevents directional bias: the forward pass alone would miss recall issues, and the reverse pass alone would miss precision issues. Results are averaged over three independent evaluation runs to account for LLM matching variability.

State extraction. RFC2TLA+ achieves an average state F1 of 0.97, compared to 0.49 for RFC2PSM, matching or exceeding RFC2PSM on all 14 protocols. RFC2TLA+ achieves near-perfect recall (0.98 average) and high precision (0.97 average). For six protocols RFC2TLA+ achieves a perfect F1 score for state extraction. RFC2PSM's approach tends to over-generate states, resulting in comparable recall (0.83 average) but much lower precision (0.38 average).

Transition extraction. RFC2TLA+ achieves an average transition extraction F1 score of 0.61, compared to 0.21 for RFC2PSM. RFC2TLA+ outperforms RFC2PSM on all 14 protocols for transition F1 score. The highest gains are on BGP (+0.45 F1), PPTP (+0.67 F1), and MQTT (+0.50 F1). Transition extraction is inherently harder than state extraction—it requires reasoning about guard conditions, message types, and variable updates—and both approaches show lower scores for transition extraction than for state extraction. RFC2TLA+ achieves substantially higher transition extraction recall than RFC2PSM (0.84 average vs. 0.58), exceeding 0.80 recall for seven protocols. Notably, RFC2TLA+ also achieves higher transition precision (0.49 average vs. 0.13 for RFC2PSM), indicating that RFC2PSM generates many spurious transitions. We note that the precision metric is likely an underestimate as the FSMs that RFC2TLA+ extracts tend to be more fine-grained and complete than the human-extracted ground-truth ones. Upon inspection, we find that many transitions in the RFC2TLA+-extracted FSM are in fact correct but have no corresponding transition in the ground truth FSM, either because several states and transitions that RFC2TLA+ extracted have been collapsed into a single state in the ground truth FSM, or the ground truth FSM has errors as we describe below.

Errors in ground truth models. Our evaluation reveals that PSMBench ground truths are incomplete or misaligned with the RFC for several protocols, which deflates the reported scores of both approaches. The ground truth models NNTP as a linear pipeline with sequential states (e.g., `Group_Selected` → `Article_Selected` → `Reading_Headers`), whereas RFC 3977 defines a fundamentally stateless command model where these commands are self-loops within a reader mode. The ground truth also omits transit-mode behavior (`IHAVE`) entirely. Consequently, our extraction contains 18 states and 201 transitions, compared with 10 states and 16 transitions in the ground truth. Although the bidirectional matching successfully aligns the different state abstractions, yielding a state F1 of 0.99, the transition F1 remains 0.34 because many RFC-grounded transitions have no corresponding transition in the ground truth.

For POP3, the ground truth omits valid RFC commands (`TOP`, `UIDL`), penalizing our higher-fidelity extraction. For DHCP, the ground truth adopts the client perspective; our pipeline extracts both client and server entities. We report the best-matching entity (client, F1=1.00 for states). These observations suggest that the PSMBench ground truths, while useful as a standard benchmark, may undercount extractions that produce more complete models like ours.

4.2 RQ2: Verification Results

A key contribution of RFC2TLA+ over prior work is its ability to generate complete, verifiable TLA + specifications from large RFCs and use model checking to detect protocol defects.

Scalability. RFC2TLA+ successfully generates TLA + specifications for all 14 protocols in PSMBench, including large and complex specifications such as SIP (648K characters) and RTSP (746K characters). The segmentation and RAG strategies (§3.1) enable document processing far beyond the LLM's effective context window. The generated specifications range from 262 lines (DHCP) to 1944 lines (NNTP), with an average of 710 lines of TLA + per protocol.

Model complexity. The extracted models capture substantial protocol complexity, with an average of 17.07 states and 87.35 transitions across all protocols. NNTP produces the largest model (201 transitions), reflecting its rich command set, while IMAP produces the most compact model relative to its RFC size (28 transitions from a 367K-character document), as its specification focuses on high-level state descriptions. The number of tool calls correlates with model complexity rather than document size. For example, NNTP requires 219 tool calls despite being smaller than SIP, which requires 126 calls. This difference arises because NNTP's detailed command-response structure demands more fine-grained transition modeling.

4.2.1 TLC Verification and Fault Injection. Within 3 runs, the generated TLA + specifications for all 14 protocols are syntactically valid and accepted by TLC for model checking. To assess whether TLC can detect protocol defects on the generated models, we conduct repeated verification runs on all protocols using both the original (correct) RFC and two types of fault-injected variants:

Deadlock injection removes sentences describing server responses to client commands, creating states from which no further progress is possible.

Invariant bypass injection introduces an alternative protocol path that bypasses a required step (e.g., authentication), which leads to an invariant violation (e.g., "the session must be authenticated before accessing data").

We ensure that RFC2TLA+ has no prior knowledge of how we inject the bugs, e.g., which invariants are affected. Table 3 summarizes the results across three independent runs per variant.

Correct specifications. The unmodified RFC specifications pass TLC without errors in 36 out of 42 runs (85.7%). Eight of the 14 protocols pass in all three runs. We analyze those occasional failures, and some of them are caused by vagueness in the natural language. For instance, one FTP run produces a deadlock because the RFC does not fully specify client behavior when the server closes a data connection prematurely, and one PPP run encounters a deadlock due to underspecified LCP negotiation recovery. Thus, these results might even lead to suggestions for RFC improvements.

Deadlock detection. TLC detects injected deadlocks in 33 out of 42 runs (78.6%). Seven protocols achieve 100% detection. Protocols with request-response patterns (e.g., PPTP, SMTP) yield models where missing response reliably creates an observable deadlock.

Invariant bypass detection. The invariant bypass bug is detected in 36 out of 42 runs (85.7%). On eight protocols RFC2TLA+ achieves 100% detection. For POP3, the LLM consistently extracts an invariant requiring authentication before entering the transaction state, and TLC identifies a trace that bypasses this check via the injected path. This result demonstrates that the three-pass architecture (§3.2), which explicitly extracts invariants, is critical for detecting semantic errors in protocol specifications.

Analysis of non-detections. Bug detection is non-deterministic because an LLM may or may not extract the relevant transitions or invariants from the buggy text. A bug goes undetected in two scenarios: (1) the LLM does not extract the buggy transition at all, effectively ignoring the injected fault; or (2) the LLM does not extract a sufficiently strong invariant to flag the violation. Fault injection into the PPP protocol illustrates case (1): its complex multi-phase structure leads to LLM's inability to model specific interactions affected by the faults.

A key concern is that LLMs may fall back on memorized protocol knowledge—reconstructing the correct protocol behavior from their training data rather than faithfully extracting from the (fault-injected) input text. Although we anonymize all RFCs by stripping protocol names and metadata (§3.1), the LLM can still recognize well-known protocols from structural cues. Notably, the fact that RFC2TLA+ *does* detect bugs in a majority of runs despite this adversarial tendency shows that RFC2TLA+'s tool-constrained extraction is effective at grounding the LLM in the input. The synthetic RFC experiments below further isolate this memorization effect.

Table 4: TLC model checking results on synthetic RFCs.

Protocol	Size	Correct	Deadlock	Inv. Bypass
Synth-1	10K	3/3	3/3	3/3
Synth-2	45K	3/3	2/3	3/3
Synth-3	86K	2/3	2/3	3/3
Overall		8/9	7/9	9/9

4.2.2 Synthetic RFC Experiments. The non-detection analysis above raises the question that when the LLM fails to detect an injected bug, is it because our pipeline is insufficient, or because the LLM "recognizes" the protocol from its training data and silently reconstructs the correct behavior, ignoring the input with injected faults?

To mitigate this memorization effect, we construct 3 synthetic RFCs using a different LLM (Claude Opus 4.6) to write original protocol specifications that follow the structure and conventions of real RFCs but describe entirely fictitious protocols that we assumed do not exist in any model's training data. Each synthetic RFC defines 2 entities with 5–10 states, and we scale the document size from 10K to 86K characters to test across different complexities. For each, we create a correct version, a deadlock-injected version, and an invariant-violation-injected version, and run RFC2TLA+ three times on each variant (27 runs total).

Table 2: RFC2TLA+ vs. RFC2PSM (GPT-5.1) [31] on 14 RFC protocols. State columns depict P, R, and F1 of state extraction for RFC2TLA+ and RFC2PSM. Transition columns depict the same metrics for transition extraction. Best values per metric in bold.

	State						Transition					
	RFC2TLA+			RFC2PSM			RFC2TLA+			RFC2PSM		
Protocol	P	R	F1	P	R	F1	P	R	F1	P	R	F1
BGP	**1.00**	1.00	**1.00**	0.50	1.00	0.67	**0.61**	**0.95**	**0.75**	0.19	0.77	0.30
DCCP	**1.00**	1.00	**1.00**	0.64	1.00	0.78	**0.49**	0.77	**0.59**	0.24	**0.84**	0.38
DHCP	**1.00**	1.00	**1.00**	0.73	1.00	0.84	**0.52**	**0.92**	**0.66**	0.41	0.90	0.57
FTP	**1.00**	**0.87**	**0.93**	0.24	0.60	0.34	**0.20**	**0.71**	**0.31**	0.03	0.33	0.06
IMAP	**0.96**	**1.00**	**0.98**	0.33	0.80	0.47	**0.45**	**0.76**	**0.56**	0.05	0.64	0.09
MQTT	**1.00**	**1.00**	**1.00**	0.32	0.92	0.48	**0.46**	**0.88**	**0.61**	0.06	0.35	0.11
NNTP	**0.98**	**1.00**	**0.99**	0.23	0.40	0.30	**0.20**	**0.94**	**0.34**	0.01	0.06	0.01
POP3	**0.88**	1.00	**0.93**	0.43	1.00	0.60	**0.58**	**0.91**	**0.70**	0.21	0.72	0.32
PPP	**0.93**	1.00	**0.97**	0.59	1.00	0.74	**0.64**	**0.80**	**0.71**	0.10	0.59	0.18
PPTP	**1.00**	**0.89**	**0.94**	0.06	0.22	0.10	**0.60**	**0.77**	**0.67**	0.00	0.00	0.00
RTSP	**1.00**	1.00	**1.00**	0.20	1.00	0.33	**0.62**	0.80	**0.70**	0.31	**0.94**	0.46
SIP	**0.97**	1.00	**0.99**	0.06	1.00	0.11	**0.52**	0.87	**0.64**	0.08	**1.00**	0.14
SMTP	**0.86**	1.00	**0.92**	0.30	1.00	0.47	**0.55**	**0.94**	**0.70**	0.10	0.50	0.16
TCP	**1.00**	**1.00**	**1.00**	0.67	0.73	0.70	**0.46**	**0.72**	**0.55**	0.06	0.45	0.10
Average	**0.97**	**0.98**	**0.97**	0.38	0.83	0.49	**0.49**	**0.84**	**0.61**	0.13	0.58	0.21

Table 3: TLC model checking results on correct and fault-injected RFCs. *Correct* is the fraction of runs that pass TLC without errors. *Deadlock* and *Inv. Bypass* is the fraction of runs where TLC detected the injected bug.

Protocol	Size	Correct	Deadlock	Inv. Bypass
POP3	47K	3/3 (100%)	1/3 (33%)	2/3 (67%)
MQTT	89K	3/3 (100%)	2/3 (67%)	3/3 (100%)
PPP	103K	2/3 (67%)	3/3 (100%)	3/3 (100%)
DHCP	114K	2/3 (67%)	2/3 (67%)	2/3 (67%)
PPTP	133K	2/3 (67%)	3/3 (100%)	2/3 (67%)
FTP	147K	2/3 (67%)	2/3 (67%)	3/3 (100%)
BGP	223K	3/3 (100%)	1/3 (33%)	2/3 (67%)
SMTP	226K	3/3 (100%)	3/3 (100%)	2/3 (67%)
NNTP	247K	2/3 (67%)	3/3 (100%)	3/3 (100%)
TCP	264K	3/3 (100%)	2/3 (67%)	3/3 (100%)
DCCP	319K	3/3 (100%)	3/3 (100%)	2/3 (67%)
IMAP	367K	3/3 (100%)	2/3 (67%)	3/3 (100%)
SIP	648K	2/3 (67%)	3/3 (100%)	3/3 (100%)
RTSP	746K	3/3 (100%)	3/3 (100%)	3/3 (100%)
Overall		36/42 (85.7%)	33/42 (78.6%)	36/42 (85.7%)
Best-of-3		14/14 (100%)	14/14 (100%)	14/14 (100%)

Table 4 shows comparable validation rates on correct synthetic and real RFCs (8/9, 89%) and higher fault detection on synthetic RFCs (16/18, 89%) than on real RFCs (69/84, 82.1%), suggesting that RFC2TLA+ generalizes for extracting formal models for verification, and suggest that the fault-detection performance on real RFCs may underestimate RFC2TLA+'s capabilities.

4.3 RQ3: Sensitivity to the Underlying LLM

The quality of the extracted CFSM and its verification outcome may depend on the underlying LLM's ability to follow construction-tool schemas, maintain a consistent model in a long interaction, and respond to deterministic validation and TLC feedback. We therefore evaluate the sensitivity of RFC2TLA+ to the choice of LLM.

Setup. We compare GPT-5.1 with two open-weight models, namely, DeepSeek-V4-Pro and Qwen3-235B-A22B-Instruct-2507. All three models support tool calls. We follow the exact experimental setup as RQ2 in §4.2. RFC2TLA+ is evaluated with three runs on all three models for each RFC variant, producing 126 runs per model.

Results. As shown in Table 5, GPT-5.1 achieves the highest specification validity and overall accuracy. DeepSeek-V4-Pro also follows the construction-tool schemas reliably, producing valid specifications in 118 of 126 runs (93.7%), but achieves a lower accuracy of 61.9%. The eight invalid DeepSeek specifications resulted primarily from inconsistencies across tool calls, including variables removed while still referenced by transitions, incompatible representations of message fields, and invalid invariant expressions.

DeepSeek's main source of error is semantic misinterpretation. For example, the deadlock-injected BGP document requires both peers to receive a `KEEPALIVE` before sending one. DeepSeek bypassed the injected circular dependency, causing TLC to report no deadlock. This indicates that DeepSeek sometimes normalizes explicit protocol deviations instead of preserving them faithfully.

Qwen shows a distinct weakness in long tool-calling interactions: it often fails to maintain schema-compliant structured outputs. For instance, a malformed JSON tool call caused the provider to reject the subsequent requests due to invalid JSON, halting the construction. This suggests that Qwen's tool-calling reliability degrades over extended interactions, particularly under long-context tasks.

Overall, both open-weight models demonstrate the basic capabilities required by RFC2TLA+, but remain less reliable than GPT-5.1. DeepSeek achieves high specification validity but low accuracy

Table 5: RFC2TLA+'s sensitivity to the LLM. *Valid Spec.* is the fraction of runs that produce a valid specification. *Accuracy* is the fraction of runs whose verification outcome matches expectation: successful verification for a correct RFC, proper error detected for a bug-injected RFC. *No Spec.* is the fraction of runs that terminate before producing a specification.

Model	Valid Spec.	Accuracy	No Spec.
GPT-5.1	126/126 (100.0%)	105/126 (83.3%)	0/126 (0.0%)
DeepSeek	118/126 (93.7%)	78/126(61.9%)	2/126(1.6%)
Qwen	15/126(11.9%)	9/126(7.1%)	39/126(31.0%)

Table 6: Ablation results of the four RFC2TLA+ components.

Configuration	Valid Spec.	Accuracy	No Spec.
Full pipeline	126/126 (100.0%)	105/126 (83.3%)	0/126 (0.0%)
w/o RAG	41/42 (97.6%)	28/42 (66.7%)	0/42 (0.0%)
w/o iterative refinement	39/42 (92.9%)	13/42 (31.0%)	0/42 (0.0%)
w/o tool use	20/42 (47.6%)	5/42 (11.9%)	0/42 (0.0%)
w/o deterministic validation	1/42 (2.4%)	0/42 (0.0%)	0/42 (0.0%)

because it sometimes fails to preserve protocol-specific semantics across the full extraction. Qwen performs substantially worse, with 11.9% specification validity and 7.1% accuracy, largely due to malformed tool calls and context limitations in long interactions.

4.4 RQ4: Ablation Study

To determine the individual contribution of each design component to RFC2TLA+, we ablate the four mechanisms that determine context selection (RAG), error correction (iterative refinement), model construction (tool use), and error prevention (deterministic tool-call validation). Each ablation only disables one component, producing four configurations: *w/o-RAG*, *w/o iterative refinement*, *w/o tool use*, and *w/o deterministic validation*.

In *w/o-RAG*, we disable the retrieval of additional RFC chunks. In *w/o iterative refinement*, deterministic validation remains enabled, but we disable the post-pass connectivity and transition-table corrections, CFSM validation repairs, and TLC-guided correction passes. In *w/o tool use*, we remove the CFSM construction interface and ask the LLM to generate TLA + module and TLC configuration. In *w/o deterministic validation*, the same construction tools are available, but tool calls are executed without checking symbol declarations, state and channel references, expression syntax, message schema, variable-update coverage, or duplicate model elements.

Setup. We use the same underlying model and experimental setup as RQ2 in §4.2. Due to the high cost of additional experiments, each ablation is evaluated only once on each RFC-variant, totaling 42 runs. Similar to §4.3 (Table 5), we measure how removing each component affects specification validity, result accuracy, and the fraction of RFCs for which no spec was produced.

Results. As shown in Table 6, the full pipeline achieves 100% specification validity and 83.3% accuracy. Removing RAG has little impact on specification validity, which drops from 100% to 97.6%, but reduces accuracy to 66.7%. This indicates that RAG primarily improves semantic grounding rather than syntactic validity. Tool use and deterministic validation can preserve the formal structure, but may omit or misrepresent protocol-specific transitions and invariants.

Disabling iterative refinement has a smaller effect on specification validity but a substantial effect on verification accuracy: only 13/42 (31%) runs produce the expected verification outcome. Thus, even though the initial extraction generates syntactically correct models, TLC-guided refinement is essential to correct semantic inconsistencies that deterministic validation cannot detect.

Removing tool use has a larger effect, reducing specification validity to 47.6% and accuracy to 11.9%. Generation without construction tools requires the LLM alone to maintain consistency across declarations, transitions, invariants, and TLC configuration. Common failures include malformed expressions, undefined operators, unassigned constants, and specification–configuration mismatches. Without tool use, a syntactically plausible TLA + artifact may be generated, but it does not reliably preserve protocol behavior.

Removing deterministic validation alone while keeping the others has the largest impact on specification validity, because the LLM can neither inspect the generated TLA + nor get feedback from the tools. As a result, invalid references, duplicate declarations, and malformed expressions are silently accepted. Only 1/42 runs (2.4%) produce a valid specification, and none yield the expected verification result. Construction tool use and deterministic validation are interdependent: tools abstract TLA + generation, while validation enforces constraints hidden by that abstraction.

Overall, construction-tool constraints and deterministic validation primarily enhance the structural integrity of the generated specification, whereas RAG and iterative refinement primarily improve its fidelity to the RFC. The full pipeline is required to achieve both high specification validity and high verification accuracy.

4.5 Rejected Tool Calls

Across all runs we observed, the pipeline issues a total of 44,519 tool calls, where 531 (1.2%) calls are rejected by the validation layer (§3.2). Critically, all rejections are caught *before* the CFSM is modified: the validation layer returns a structured error to the LLM, which retries with a corrected tool call. Without the validation layer, these errors would silently propagate into the generated TLA + specifications. For example, we observe the following four types of violations:

Nonexistent element references. the LLM attempts to remove or modify a non-existing transition or invariant in the current CFSM.

Undeclared state references. the LLM creates transitions reference states that were never introduced. Each introducing a dangling FSM state, breaking reachability analysis and TLC verification.

Duplicate declarations. the LLM re-declares a variable or entity that already exists, sometimes with a conflicting type. Without rejection, these would cause repeated definition in the final generated model, preventing it from being verified.

Malformed expressions. the LLM introduces guard conditions or invariant expressions that contain syntactic errors (e.g., = instead of ==) or logically vacuous expressions (e.g., tautologies such as `x != "A" ∨ x == "A"`). These would produce TLA + specifications that either fail to parse or encode meaningless conditions.

4.6 Case Study: Iterative Refinement on SMTP

We illustrate the effect of iterative refinement with a representative case from SMTP (RFC 5321 [18]), the Simple Mail Transfer Protocol that governs email transmission between mail servers across the Internet. At 226K characters and 116 segments, SMTP is a medium-sized but structurally rich protocol, defining a multi-phase session lifecycle spanning connection setup, mail transaction, data transfer, and session teardown.

CFSM construction. RFC2TLA+ processes the 116 segments across three passes to construct the CFSM. State Extraction identifies 2 entities (Client and Server), 2 channels (client-to-server and server-to-client), and 12 states (6 per entity): the Client progresses through *Start*, *SessionInitialized*, *MailTransactionStarted*, *RecipientsSpecified*, *DataTransfer*, and *Done*; the Server mirrors this with *Initial*, *SessionInitialized*, *MailTransactionStarted*, *RecipientsSpecified*, *DataTransfer*, and *SessionTerminated*. RFC2TLA+ extracts 101 transitions modeling the SMTP command–response protocol (`EHLO`, `MAIL FROM`, `RCPT TO`, `DATA`, `QUIT`, etc.) and other cross-entity state consistency invariants. The resulting TLA + specification is 864 lines.

Iterative refinement. Among the initially generated invariants requires the Client and Server to enter their termination states simultaneously. This is overly strong for an asynchronous CFSM: under RFC 5321, the Client sends `QUIT`, the Server receives it, returns a `221` reply, and then closes the connection. The entities may therefore occupy different states while a command or reply is in transit.

In Stage III, TLC produces a counterexample in which one entity has entered its termination state while the other remains in its previous state and a termination message is buffered. Given the counterexample, violated invariant, CFSM, and relevant RFC text, the LLM removes the unsupported invariants while preserving the RFC-permitted termination behavior. TLC then explores 801 states (194 distinct) and reports no errors.

This case shows how three pipeline components work together: the three-pass design independently extracts the transition and invariants, TLC exposes their inconsistency with a concrete counterexample, and iterative refinement removes the unsupported invariant without regenerating the full model. Without this combination, the modeling error would remain undetected.

4.7 Token Consumption

Table 7 reports the average token consumption according to OpenAI's calculation suggestion [23] per original protocol and its fault-injected variants, providing a measure of LLM costs, the consumption is averaged over three runs.

Longer protocols tend to consume more tokens, but the relationship is driven by protocol complexity, such as the number of states, transitions, and interaction patterns, instead of document size alone. Among larger RFCs, unmodified NNTP (247K characters) consumes up to 46.8M tokens due to its rich command set and 201 transitions, whereas DCCP (319K characters) requires only 6.7M tokens despite being a larger document, because its state machine is more compact. Similarly, the correct SMTP variant (226K characters) consumes 55.3M tokens which is more than what is needed for the larger TCP (264K characters, 4.3M tokens). The cause of this might be SMTP's multi-phase session lifecycle, which demands more fine-grained transition modeling. Buggy variants can consume more tokens than correct variants when injected faults trigger additional iterative refinement cycles. For example, SIP deadlock injection increases token consumption from 29.4M to 75.6M as the pipeline iterates to "fix" the model with the faulty specification. Synthetic RFCs follow the same complexity-driven trend: Synth-1 (10K characters) uses 658K tokens, scaling to 4.3M for Synth-3 (86K characters).

Table 7: Average estimated token consumption per protocol and variant, averaged over 3 independent runs. Token consumption computation follows OpenAI's method.

Protocol	Size	Correct	Deadlock	Inv. Bypass
POP3	47K	2,085K	1,627K	3,353K
MQTT	89K	326K	380K	372K
PPP	103K	2,834K	7,221K	7,873K
DHCP	114K	8,258K	7,718K	1,915K
PPTP	133K	20,439K	5,991K	3,592K
FTP	147K	9,996K	15,069K	19,039K
BGP	223K	9,045K	2,808K	2,375K
SMTP	226K	55,252K	8,494K	9,925K
NNTP	247K	46,788K	80,589K	28,887K
TCP	264K	4,347K	4,211K	28,061K
DCCP	319K	6,725K	6,478K	6,787K
IMAP	367K	12,483K	12,502K	28,564K
SIP	648K	29,416K	75,595K	31,378K
RTSP	746K	23,297K	14,096K	15,476K
Synth-1	10K	658K	262K	308K
Synth-2	45K	2,668K	1,144K	1,211K
Synth-3	86K	4,310K	4,052K	3,943K

5 Threat to Validity

External threats. External threats to validity arise from the scope and generalizability of our evaluation. We are limited by the 14 RFC protocols from PSMBench [31], which, while diverse, remain limited in scale, with only three runs per protocol and a fixed model checker. This may limit the applicability of our findings to other specification documents or checkers. Furthermore, while injected deadlocks and invariant violations will be distributed for reproducible assessments of bug detection, they may not fully reflect the diversity and subtlety of real-world errors in protocol specifications. As a result, our conclusions focus on consistent trends across protocols rather than statistical significance, and broader evaluation across models and specifications is left for future work.

Internal threats. Internal validity is affected by challenges in assessing the quality of the extracted models. This remains an open problem, particularly for large RFCs, where specifications introduce complexity due to distributed behaviors and implicit assumptions. As a result, evaluation requires metrics that more accurately capture behavioral coverage and alignment with the source specification.

To mitigate this, we adopt the most recent evaluation protocol introduced in the literature [31], although it is largely based on syntactic similarity. To this, we also show that RFC2TLA+'s extracted formal specifications can be verified by a model checker, and that faults injected into the documents result in counterexamples when

the resulting formal specifications are checked. Developing scalable and semantically meaningful evaluation methods remains an important direction for future work.

Other threats. Other threats are primarily related to LLM behavior. These include nondeterminism and the potential reliance on memorized protocol knowledge. Even after anonymization, well-known protocols may still be identifiable by LLMs through structural similarity, which can help the model to "adversarially" reconstruct expected behavior rather than faithfully extract it from the provided protocol specification. This phenomenon is an instance of the broader *data contamination* problem [10], where unintended overlap between training and evaluation data undermines the validity of benchmark results. In our setting, contamination manifests not as inflated test scores but as reduced bug sensitivity: the LLM reconstructs the correct protocol from memory, masking the injected defect. Our anonymization step (§3.1) partially mitigates surface-level recognition, but structural cues (e.g., the three-state POP3 pattern) can still trigger memorized behavior.

To address this, we anonymize RFCs, repeat experiments across multiple runs, and incorporate synthetic RFCs that are not present in the training data, thereby helping to isolate memorization effects. Notably, one practical use case of RFC2TLA+ is assisting RFC developers in drafting new RFCs, which are absent from training data and thus less prone to memorization artifacts.

6 Related Work

Model checking. Classical model checking enables formal analysis of distributed systems and protocols, with mature tools such as SPIN [17], nuXmv [9], Tamarin [2], ProVerif [7], and UPPAAL [3], each targeting domains including infinite-state systems, cryptographic protocols, and real-time verification. Recent tools such as Apalache for TLA + and Quint [1] further improve accessibility for exhaustive state-space exploration. These model checkers have shown practical uses. For instance, ProVerif has been used to analyze QUIC's handshake [36], and Google uses TLA + to formalize BGP [29]. Further, Ginesin et al. [15] provide a comprehensive analysis of SCTP, synthesizing attack traces and verifying fixes, which demonstrates model checking's dual role in vulnerability discovery and validation. However, constructing formal specifications remains labor-intensive and requires substantial expertise. Beyond hand-crafted specifications, active automata learning approaches such as AALpy [20] and LearnLib [25] can infer protocol models from implementations, enabling applications like TLS state machine learning [4] and automated conformance checking [33]. In contrast, RFC2TLA+ lifts formal specifications directly from the RFC text. By combining segmentation, tool-enabled three-stage extraction, and feedback, it automates model construction and invariant extraction, capturing protocol semantics efficiently.

Automated natural language document analyses with LLMs. Recent advances in LLMs have enabled automated extraction of formal models from natural language. RFCNLP [24] introduces the first large-scale dataset and infrastructure for FSM extraction from RFCs, but struggles with structural complexity, resulting in incomplete models. PROSPER [30] demonstrates the feasibility of LLM-enabled extraction of protocol specifications, but its scope is limited to small specifications. Further, its generalizability is limited due to the human-in-the-loop framework design. Li et al. [19] systematically evaluate RFC specification extraction to formal model extraction and conclude that LLMs can effectively extract formal specifications from natural language documents to facilitate autotesting. However, the approach's false positive rate is alarmingly high, exceeding 90%, and even when equipped with the best-performing LLM, it has a relatively low accuracy of 71%. Cosler et al. [11] propose nl2spec, which translates natural language into temporal logic via decomposed subformulas and user feedback. While focused on property specification rather than full model extraction, it highlights the effectiveness of structured intermediate representations. Prior work, therefore, addresses two related but distinct tasks. RFCNLP [24] and RFC2PSM [31] extract behavioral models without properties, whereas Li et al. [19] and nl2spec [11] extract standalone properties without governing the behavioral model. In contrast, RFC2TLA+ jointly extracts a multi-entity CFSM and invariants grounded in its declared entities, states, and variables. It deterministically translates the CFSM into a TLA + behavioral specification and each extracted invariant into a named TLA + state predicate. The generated TLC configuration registers these predicates as verification properties, which TLC checks over all reachable states of the translated CFSM.

7 Conclusion

We presented RFC2TLA+, a tool-enabled pipeline for automatically lifting RFC specifications into formally verifiable TLA + models. By decomposing the task into state, transition, and invariant extraction, and combining it with segmentation, RAG, and iterative feedback from a model checker, RFC2TLA+ enables end-to-end automated verification. Notably, RFC2TLA+'s invariant extraction enables validation against generated models with the TLC model checker.

Our evaluation demonstrated improved completeness over prior approaches and shows that RFC2TLA+ can effectively handle larger and more complex specifications. We also demonstrated that extracted models and invariants are sufficient to capture deadlocks and errors in protocol operation with high recall. Experiments on synthetic RFCs further suggested that the pipeline can detect injected defects while mitigating the impact of training data leakage. Overall, our results imply that tool-augmented LLM pipelines with immediate feedback are a promising direction for bridging natural-language specifications and formal verification, and play a valuable role in supporting the development of future protocol designs.

Acknowledgments

We thank the anonymous reviewers and shepherd for their insightful feedback. Funding for this work was provided in part by NSERC Discovery Grant RGPIN-2026-07548, a contract with Telus, NSERC Alliance Grant ALLRP 586310-23, and NSERC-CSE Research Communities Grant ALLRP 588144-23. David Lie is supported by Tier 1 Canada Research Chair CRC-2019-00242. Researchers funded through the NSERC-CSE Research Communities Grants do not represent the Communications Security Establishment Canada or the Government of Canada. Any research, opinions or positions they produce as part of this initiative do not represent the official views of the Government of Canada.

Data Availability Statement

All experiment source code, synthetic RFC documents, and fault-injected RFC variants generated for evaluation have been made available in Zenodo [12].

References

[1] Apalache Development Team. 2025. Apalache Model Checker. https://apalache-mc.org/.

[2] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. 2017. Symbolically analyzing security protocols using tamarin. *ACM SIGLOG News* 4, 4 (Nov. 2017), 19–30. doi:10.1145/3157831.3157835

[3] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control* (New Brunswick, New Jersey, USA). Springer-Verlag, Berlin, Heidelberg, 232–243. doi:10.1007/BFb0020949

[4] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2017. A messy state of the union: taming the composite state machines of TLS. *Commun. ACM* 60, 2 (Jan. 2017), 99–107. doi:10.1145/3023357

[5] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification*. Springer, 184–190. doi:10.1007/978-3-642-22110-1_16

[6] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly Media.

[7] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Found. Trends Priv. Secur.* 1, 1–2 (Oct. 2016), 1–135. doi:10.1561/3300000004

[8] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (1983), 323–342. doi:10.1145/322374.322380

[9] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 334–342. doi:10.1007/978-3-319-08867-9_22

[10] Yuxing Cheng, Yi Chang, and Yuan Wu. 2025. A Survey on Data Contamination for Large Language Models. doi:10.48550/arXiv.2502.14425

[11] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In *Proceedings of the 35th International Conference on Computer Aided Verification (CAV)*. Springer, 383–396. doi:10.1007/978-3-031-37703-7_18

[12] Guozhen Ding, Kexin Li, Ilya Grishchenko, and David Lie. 2026. *Reproduction Package for RFC2TLA+*. doi:10.5281/zenodo.21788017

[13] Wesley M. Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. doi:10.17487/RFC9293

[14] Clive Feather. 2006. Network News Transfer Protocol (NNTP). RFC 3977. doi:10.17487/RFC3977

[15] Jacob Ginesin, Max von Hippel, Evan Defloor, Cristina Nita-Rotaru, and Michael Tüxen. 2024. A Formal Analysis of SCTP: Attack Synthesis and Patch Verification. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3099–3116. https://www.usenix.org/conference/usenixsecurity24/presentation/ginesin

[16] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381. doi:10.1007/s100090050043

[17] G.J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295. doi:10.1109/32.588521

[18] John C. Klensin. 2008. Simple Mail Transfer Protocol. doi:10.17487/RFC5321

[19] Hui Li, Zhen Dong, Siao Wang, Hui Zhang, Liwei Shen, Xin Peng, and Dongdong She. 2025. Extracting Formal Specifications From Documents Using LLMS for Test Automation . In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. doi:10.1109/ICPC66645.2025.00039

[20] Edi Muškardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. 2022. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering* 18, 3 (03 2022), 417–426. doi:10.1007/s11334-022-00449-3

[21] David Noveck and Chuck Lever. 2020. Network File System (NFS) Version 4 Minor Version 1 Protocol. RFC 8881. doi:10.17487/RFC8881

[22] OpenAI. 2025. File Search and Vector Stores. https://developers.openai.com/api/docs/assistants/tools/file-search/. Accessed: 2026-03-24.

[23] OpenAI. 2026. *What are tokens and how to count them?* https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them Accessed: 2026-03-26.

[24] Maria Leonor Pacheco, Max von Hippel, Ben Weintraub, Dan Goldwasser, and Cristina Nita-Rotaru. 2022. Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 51–68. doi:10.1109/SP46214.2022.9833673

[25] Harald Raffelt, Bernhard Steffen, and Therese Berg. 2005. LearnLib: a library for automata learning and experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems* (Lisbon, Portugal) *(FMICS '05)*. Association for Computing Machinery, New York, NY, USA, 62–71. doi:10.1145/1081180.1081189

[26] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. https://datatracker.ietf.org/doc/html/rfc8446. doi:10.17487/RFC8446 Accessed: 2025-09-10.

[27] Ivan Ristić. [n. d.]. *OpenSSL Cookbook: A Practical Guide to TLS and SSL* (3rd ed.). Feisty Duck. https://www.feistyduck.com/library/openssl-cookbook/online/testing-with-openssl/testing-renegotiation.html

[28] Dr. Marshall T. Rose and John G. Myers. 1996. Post Office Protocol - Version 3. RFC 1939. doi:10.17487/RFC1939

[29] Aman Shaikh. 2024. Specifying BGP using TLA+. Presentation. https://research.google/pubs/specifying-bgp-using-tla/ Accessed: 2026-08-03.

[30] Prakhar Sharma and Vinod Yegneswaran. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (Cambridge, MA, USA) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 41–47. doi:10.1145/3626111.3628205

[31] Zilin Shen, Xinyu Luo, Imtiaz Karim, and Elisa Bertino. 2025. PSMBench: A Benchmark and Dataset for Evaluating LLMs Extraction of Protocol State Machines from RFC Specifications. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*. https://neurips.cc/virtual/2025/poster/121835 Poster.

[32] The Pallets. 2026. Jinja2. https://github.com/noirbizarre/jinja2.

[33] Arthur Tran Van, Olivier Levillain, and Herve Debar. 2024. Mealy Verifier: An Automated, Exhaustive, and Explainable Methodology for Analyzing State Machines in Protocol Implementations. In *Proceedings of the 19th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '24)*. Association for Computing Machinery, New York, NY, USA, Article 16, 10 pages. doi:10.1145/3664476.3664506

[34] U.S. Cybersecurity & Infrastructure Security Agency (CISA). 2014. *OpenSSL "Heartbleed" vulnerability (CVE-2014-0160)*. Alert TA14-098A. CISA, United States. https://www.cisa.gov/news-events/alerts/2014/04/08/openssl-heartbleed-vulnerability-cve-2014-0160

[35] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*. Springer-Verlag, Berlin, Heidelberg, 54–66. doi:10.1007/3-540-48153-2_6

[36] Jingjing Zhang, Lin Yang, Xianming Gao, and Qiang Wang. 2020. Formal analysis of QUIC handshake protocol using ProVerif . In *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. IEEE Computer Society, Los Alamitos, CA, USA, 132–138. doi:10.1109/CSCloud-EdgeCom49738.2020.00030

Received 2026-03-26; accepted 2026-06-18