

# RFC-013: TLA+ Formal Specification

**Status:** Proposed **Date:** January 2026 **Author:** Derrell Piper  
ddp@eludom.net

---

## Abstract

This RFC specifies the use of TLA+ (Temporal Logic of Actions) for formal specification and model checking of Cyberspace protocols, ensuring correctness before implementation.

---

## Motivation

Running code is necessary but not sufficient:

- **Tests check examples:** Not all possible executions
- **Reviews check logic:** Not all interleavings
- **Bugs hide in corners:** Race conditions, edge cases

TLA+ provides:

1. **Precise specification:** Mathematical description of behavior
2. **Model checking:** Exhaustive state space exploration
3. **Proof capability:** Formal verification of properties
4. **Design tool:** Find bugs before writing code

From Lamport:

*If you're thinking without writing, you only think you're thinking.*

---

## Specification

### TLA+ Overview

TLA+ describes systems as state machines:

**VARIABLES** state, messages, decisions

```
Init ==  
/\ state = [n \in Nodes |-> "idle"]
```

```

/\ messages = {}
/\ decisions = {}

Next ==
  \/\ Propose(...)
  \/\ Prepare(...)
  \/\ Commit(...)
  \/\ Decide(...)

Spec == Init /\ [] [Next]_<<state, messages, decisions>>

```

## Safety Properties

**Invariants** that must always hold:

```

TypeOK ==
  /\ state \in [Nodes -> {"idle", "prepared", "committed"}]
  /\ messages \subseq Message
  /\ decisions \subseq Value

Agreement ==
  \A n1, n2 \in Nodes:
    (decisions[n1] # {} /\ decisions[n2] # {}) =>
      decisions[n1] = decisions[n2]

```

## Liveness Properties

**Temporal** properties about progress:

```

Termination ==
  <>(\A n \in Nodes: decisions[n] # {})

EventualConsistency ==
  []<>(\A n1, n2 \in Nodes: state[n1] = state[n2])

```

---

## Cyberspace Protocol Specifications

### Threshold Signatures (RFC-007)

```

----- MODULE ThresholdSig -----
EXTENDS Integers, FiniteSets

CONSTANTS Signers, Threshold, Script

VARIABLES signatures, verified

```

```

Init ==
  /\ signatures = {}
  /\ verified = FALSE

Sign(s) ==
  /\ s \in Signers
  /\ s \notin {sig.signer : sig \in signatures}
  /\ signatures' = signatures \union
    {[signer |-> s, script |-> Script, valid |-> TRUE]}
  /\ verified' = verified

Verify ==
  /\ Cardinality({sig \in signatures : sig.valid}) >= Threshold
  /\ verified' = TRUE
  /\ UNCHANGED signatures

Next ==
  \/ \E s \in Signers: Sign(s)
  \/ Verify

/* Safety: Never verify with insufficient signatures */
Safety ==
  verified => Cardinality({sig \in signatures : sig.valid}) >= Threshold

/* Liveness: If enough sign, eventually verify */
Liveness ==
  (Cardinality(Signers) >= Threshold) => <>(verified)

=====

```

## Audit Trail (RFC-003)

```

----- MODULE AuditTrail -----
EXTENDS Integers, Sequences

CONSTANTS Actors, Actions

VARIABLES log, sequence

Init ==
  /\ log = <<>>
  /\ sequence = 0

Append(actor, action) ==

```

```

/\ actor \in Actors
/\ action \in Actions
/\ sequence' = sequence + 1
/\ log' = Append(log, [
    seq |-> sequence',
    actor |-> actor,
    action |-> action,
    parent |-> IF sequence = 0 THEN "genesis" ELSE log[sequence].hash
])

/* Invariant: Chain integrity */
ChainIntegrity ==
  \A i \in 1..Len(log)-1:
    log[i+1].parent = log[i].hash

/* Invariant: Monotonic sequence */
MonotonicSequence ==
  \A i \in 1..Len(log)-1:
    log[i+1].seq = log[i].seq + 1

```

=====

## Byzantine Consensus (RFC-011)

```

----- MODULE PBFT -----
EXTENDS Integers, FiniteSets

CONSTANTS Nodes, f, Values

ASSUME Cardinality(Nodes) >= 3*f + 1

VARIABLES
  view,
  prepares,
  commits,
  decisions

Init ==
  /\ view = 0
  /\ prepares = [n \in Nodes |-> {}]
  /\ commits = [n \in Nodes |-> {}]
  /\ decisions = [n \in Nodes |-> {}]

PrePrepare(primary, v) ==
  /\ primary = Leader(view)

```

```

/\ v \in Values
/\ \A n \in Nodes:
    prepares' = [prepares EXCEPT ![n] = @ \union {[view |-> view, value |-> v]}]
/\ UNCHANGED <<view, commits, decisions>>

Prepare(n, v) ==
/\ [view |-> view, value |-> v] \in prepares[n]
/\ Cardinality({m \in Nodes : [view |-> view, value |-> v] \in prepares[m]}) >= 2*f + 1
/\ commits' = [commits EXCEPT ![n] = @ \union {[view |-> view, value |-> v]}]
/\ UNCHANGED <<view, prepares, decisions>>

Commit(n, v) ==
/\ [view |-> view, value |-> v] \in commits[n]
/\ Cardinality({m \in Nodes : [view |-> view, value |-> v] \in commits[m]}) >= 2*f + 1
/\ decisions' = [decisions EXCEPT ![n] = {v}]
/\ UNCHANGED <<view, prepares, commits>>

/* Safety: Agreement
Agreement ==
  \A n1, n2 \in Nodes:
    (decisions[n1] # {} /\ decisions[n2] # {}) =>
      decisions[n1] = decisions[n2]

```

=====

---

## Model Checking Process

### 1. Write Specification

Define state machine and properties.

### 2. Configure Model

```

CONSTANTS
  Nodes = {n1, n2, n3, n4}
  f = 1
  Values = {v1, v2}

```

### 3. Run TLC Model Checker

```

$ tlc PBFT.tla
TLC2 Version 2.18
...
Model checking completed. No errors found.

```

```
States explored: 847293
Distinct states: 12847
```

#### 4. Analyze Counterexamples

If property violated, TLC shows trace:

Error: Invariant Agreement is violated.

Trace:

```
State 1: <Initial>
State 2: PrePrepare(n1, v1)
State 3: Prepare(n2, v1)
...
State 12: decisions = [n1 |-> {v1}, n2 |-> {v2}] << VIOLATION
```

---

### Integration with Implementation

#### Specification → Implementation

TLA+ Spec	Scheme Implementation
-----	-----
VARIABLES state	(define-record-type <state> ...)
Init ==	(define (init) ...)
Action(x) ==	(define (action x) ...)
Invariant	(assert (invariant? state))

#### Runtime Assertions

```
(define (append-audit! entry)
  ;; TLA+ invariant: MonotonicSequence
  (assert (> (entry-sequence entry)
            (entry-sequence (last-entry))))
  ;; TLA+ invariant: ChainIntegrity
  (assert (equal? (entry-parent entry)
                  (entry-hash (last-entry))))
  ;; Proceed with append
  ...)
```

---

### PlusCal (Algorithmic TLA+)

Higher-level syntax that compiles to TLA+:

```

--algorithm ThresholdSign {
  variables signatures = {}, verified = FALSE;

  process (signer \in Signers)
  {
    sign:
      signatures := signatures \union {self};
  }

  process (verifier = "v")
  {
    verify:
      await Cardinality(signatures) >= Threshold;
      verified := TRUE;
  }
}

```

---

## Benefits

Aspect	Without TLA+	With TLA+
Design	Informal, ambiguous	Precise, mathematical
Bugs	Found in testing/production	Found before coding
Confidence	"Seems to work"	"Proven correct"
Documentation	Natural language	Executable specification
Maintenance	Risky changes	Verify changes

---

## Limitations

- **State explosion:** Large state spaces take time
- **Learning curve:** TLA+ is different
- **Abstraction gap:** Spec  $\neq$  implementation
- **Finite models:** Cannot check infinite systems directly

Mitigations: – Symmetry reduction – Abstraction – Proof for infinite cases

---

## References

1. Lamport, L. (2002). Specifying Systems: The TLA+ Language.
  2. Lamport, L. (2009). The PlusCal Algorithm Language.
  3. Newcombe, C., et al. (2015). How Amazon Web Services Uses Formal Methods.
  4. TLA+ Tools: <https://lamport.azurewebsites.net/tla/tools.html>
- 

## Changelog

- **2026-01-06** – Initial specification
- 

**Implementation Status:** Proposed **Tool:** TLA+ / TLC Model Checker **Target Protocols:** RFC-003, RFC-007, RFC-011