0.1 Mathematica as a Specification Language

From ./AE-Specifications-ETH/standalone/Mathematica-Spec-Language.tex

Exploring **formally executable specifications** in **datacenter architecture** touches the core of verifiability, reproducibility, and automation in modern systems.

A *formally executable specification* is:
- **Precise and unambiguous:** defined mathematically or via formal syntax.
- **Executable:** interpretable or simulatable.
- **Deterministically testable:** consistent output for consistent input.

Definition: Formally Executable Specification

In datacenter contexts, this implies that hardware, networking, storage, and compute orchestration policies are:
- Executable in simulation or emulation environments,
- Amenable to formal verification for correctness, safety, and performance.

Why It Matters in Datacenters

- **Correctness:** validate failover, routing, and policy enforcement.
- **Optimization:** evaluate configurations automatically.
- **Security:** prove isolation and policy compliance.
- **Confidence:** ensure safe deployment at scale.

Relevant Tools and Technologies

Domain	Tools
Network Architecture	P4, TLA+, NetKAT, Batfish
Storage Systems	TLA+, Ivy, Alloy, Z3 SMT
Orchestration	Kubernetes CRDs, Pulumi, OPA, Nomad
Formal Languages	TLA+, Coq, Lean, Dafny, Alloy
Execution	Mininet, NS-3, OMNeT++, QEMU, Verilator

Figure 1: Selected tools for formally modeling datacenter systems

Example: Rack-Aware Topology Specification

Imagine a model with:
- Compute nodes linked via ToR (Top-of-Rack) switches,
- Spine switches in a leaf-spine topology,
- Multi-path routing and QoS,
- VM placement and replication constraints.

The spec could:
- Simulate failures and load distribution,
- Detect routing loops or black holes,
- Evaluate bandwidth and latency guarantees,
- Prove placement constraints meet SLAs.

Vision: “Datacenter-as-Code” Verified

- High-level specs compile into deployable artifacts,
- Every change is property-checked and testable,
- Infrastructure becomes version-controlled logic, replacing spreadsheets and tribal lore.

Evaluating Mathematica for Executable Specification

Mathematica is a powerful computational platform. Its value depends on whether expressiveness or formal rigor is the priority.

0.1.1 Strengths of Mathematica

Limitations Compared to Formal Languages

- **Formal Semantics:** lacks type theory foundations (Coq, Lean).
- **Verification:** no native model checking or invariant proofs.
- **Concurrency:** no Lamport clocks or message-passing models.
- **Determinism:** pattern matching may be nondeterministic.
- **Refinement:** lacks formal spec-to-implementation pathways.

Suitable Use Cases

- Modeling tradeoffs in resource allocation,
- Simulating flows using graph theory,
- Prototyping performance constraints,
- Symbolic scheduling and placement logic,
- Writing executable whitepapers with computation and code.

Where It Falls Short

- Verifying safety and liveness across all states,
- Proving conformance or refinement,
- Modeling concurrency and faults rigorously,
- Integrating with RTL verification pipelines,
- Participating in formal proof communities.

Category	**Capability**
Symbolic Computation	Excellent for pipelines, graphs, latency models
Executability	Immediate execution and visualization
Expressiveness	Supports discrete, continuous, algebraic models
Rapid Prototyping	Rich in units, semantics, interactivity
Logic Tools	First-order logic, SAT solving, quantifiers
Documentation	Notebooks are self-contained and reproducible

Figure 2: Strengths of Mathematica in system modeling

0.1.2 Summary Judgment

Mathematica is:

- **Excellent** for exploratory, high-level modeling and simulation,
- **Weak** for formal verification, proofs, and correctness guarantees,
- **Valuable** as a literate architecture spec tool, but not a full formal methods platform.

0.1.3 Appendix A: TLA+ Model – Rack-Aware Topology

RackAwareSpec.tla

```
----------------------------- MODULE RackAwareSpec -----------------------------
EXTENDS Naturals, Sequences

CONSTANTS Racks, Nodes, Links

VARIABLES rackStatus, linkStatus, trafficMap

(*--algorithm RackAware
variables rackStatus \in [Racks -> {"up", "down"}],
          linkStatus \in [Links -> {"up", "down"}],
          trafficMap \in [Nodes -> [Nodes -> {"ok", "blocked", "reroute"}]];

define
  IsAvailable(n) == \E r \in Racks: rackStatus[r] = "up" /\ n \in Nodes /\ TRUE
end define;

begin
  Init ==
    /\ \A r \in Racks: rackStatus[r] = "up"
    /\ \A l \in Links: linkStatus[l] = "up"
    /\ \A s, d \in Nodes: trafficMap[s][d] = "ok";

  Next ==
```

```
    \E r \in Racks:
      /\ rackStatus[r] = "up"
      /\ rackStatus' = [rackStatus EXCEPT ![r] = "down"]
      /\ UNCHANGED <<linkStatus, trafficMap>>
  \/ \E l \in Links:
      /\ linkStatus[l] = "up"
      /\ linkStatus' = [linkStatus EXCEPT ![l] = "down"]
      /\ UNCHANGED <<rackStatus, trafficMap>>;

end algorithm;
===============================================================================
```

Appendix B: Alloy Model – Storage Placement Constraints

StorageModel.als

```
module StorageModel

abstract sig Rack {}
sig Node {
  hostRack: one Rack,
  stores: set Volume
}
sig Volume {
  replicas: some Node
}

fact ReplicationFactor {
  all v: Volume | #v.replicas = 3
}

fact NoReplicaOnSameRack {
  all v: Volume |
    all disj n1, n2: v.replicas |
      n1.hostRack != n2.hostRack
}

pred ShowExample {}

run ShowExample for 3 Rack, 6 Node, 2 Volume
```