Project MidGARD
Requirements Document

Project Sponsors
Chris Ortiz and Rex Jackson (SanDisk)

Faculty Mentor
Jeevana Swaroop Kalapala

Team Odin
Lacy Hamilton
Andrew Gajewski
Myles Hill
Skyler Guard

Version 2

Last Updated April 30, 2026

Accepted as baseline requirements for the project:

For the client:_____ **For the team:**_____

Table of Contents

Introduction

Modern storage systems are a core part of today's computing environment. From personal devices like laptops and smartphones to large-scale cloud infrastructure, nearly every system depends on reliable storage technology. This includes products such as solid-state drives (SSDs), embedded flash storage, and high-performance memory solutions. Because these systems are responsible for storing and managing critical data, reliability and correctness are extremely important. Even small errors in system behavior can lead to data corruption, system failures, or costly delays in development.

The storage industry itself is large and continues to grow as demand for data increases. Companies in this space support a wide range of applications, including cloud computing, artificial intelligence, and enterprise systems. As a result, storage technologies must meet high performance and quality standards while also handling increasingly complex system requirements. This complexity makes both system design and validation more challenging over time.

SanDisk operates within this industry as a major provider of flash memory and storage solutions. The company develops products used by both consumers and enterprise customers, and its success depends heavily on delivering reliable, high-quality technology. Within SanDisk, engineering and product teams follow a structured workflow that includes system design, specification, validation, and testing before products are released. Ensuring that these systems behave correctly throughout this process is a key part of maintaining the company's reputation and meeting customer expectations.

As systems grow more complex, companies like SanDisk are increasingly exploring new approaches to improve how they design and validate their technologies. One area of interest is the use of formal methods, such as TLA+ (Temporal Logic of Actions), which allow engineers to describe system behavior mathematically and verify correctness using model checking tools. These approaches are becoming more common in industry because they can help identify issues earlier in the design process and provide stronger guarantees about system behavior.

This project is sponsored by Chris Ortiz, a Senior Technologist, and Rex Jackson, a Vice President in Tools & Infrastructure at SanDisk. Their team works on improving internal tools and processes that support engineering and validation efforts. The project fits into a broader initiative within SanDisk to better integrate formal methods into their workflow and improve how system behavior is analyzed and validated.

Overall, this project exists at the intersection of storage systems, formal verification, and software tooling. By focusing on this area, it contributes to ongoing efforts in the industry to improve system reliability and make complex technologies easier to design, understand, and validate.

Problem Statement

In SanDisk's current workflow, engineers use TLA+ to create formal specifications that describe system behavior. These specifications are then analyzed using the TLC model checker, which explores all possible valid executions of the system. After a successful model check, TLC can generate a state-space representation of the system in the form of a directed graph (DOT file), where nodes represent system states and edges represent transitions between those states .

This process provides a very detailed and complete view of how a system can behave. In theory, this information is extremely valuable, especially for validation and regression testing, since it captures all possible execution paths. A simplified version of this workflow can be described as:

While this workflow is powerful, it does not scale well as system complexity increases. For real-world systems, especially those involving protocols like NVMe and PCIe, the number of possible states and transitions can grow very quickly. As a result, the generated graphs often become extremely large, dense, and difficult to interpret in practice.

The main problem is that although the system produces a complete and correct representation of behavior, it does not provide a practical way for engineers to use that information for regression testing. Engineers are left with large, complex graphs that require significant effort to analyze, making it difficult to identify which execution paths are actually important.

This leads to several inefficiencies and limitations in the current workflow:

Inefficiencies:

- State graphs are too large and complex to interpret manually
- Engineers must spend significant time analyzing graphs to understand system behavior
- Debugging issues using these graphs can be time-consuming and error-prone
- Heavy reliance on expert knowledge to interpret model outputs
- Poor scalability as system size and state space increase

Missing Elements in the Current Workflow:

- No automated way to simplify or reduce graph complexity
- Lack of filtering to remove redundant or low-value transitions
- No clear method for identifying critical states or execution paths
- Limited support for generating or prioritizing useful test sequences
- No structured output that directly supports regression test planning

Overall, there is a gap between formal verification and practical testing. While TLA+ and TLC can confirm that a system behaves correctly, they do not directly help engineers decide *how to test that system efficiently*. This disconnect results in a workflow where valuable information is generated but not fully utilized, leading to inefficiencies and missed opportunities for improving test coverage.

Solution Vision

MidGARD is a system designed to bridge the gap between formal verification and practical testing. It takes state-space graphs generated in DOT format by the TLC model checker, processes and simplifies these graphs, and produces meaningful execution paths that can be used as regression tests. By transforming complex verification outputs into actionable testing artifacts, MidGARD reduces the need for manual graph interpretation and allows engineers to focus on validating system behavior more efficiently.

Key Features:

- Graph simplification and reduction to improve interpretability
- Identification of critical states and execution paths
- Automated generation of regression test sequences from state graphs
- Filtering of redundant or low-value transitions
- User-configurable parameters for test generation (e.g., seed, number of tests, coverage goals)

System Overview and Data Flow

Input Data:
MidGARD operates on DOT files generated by the TLC model checker after analyzing a TLA+ specification. These files represent the full state-space of the system, including all reachable states and transitions.

Generated Data:
The system produces:

- Simplified and reduced representations of the original state graph
- Ordered sets of execution paths that serve as regression test sequences
- Coverage-related information indicating how thoroughly the system behavior has been explored

Core Processes:
MidGARD transforms input graphs into useful testing outputs through several high-level processes:

- **Graph Reduction:** The system reduces the complexity of the input graph by collapsing redundant or equivalent states and minimizing unnecessary structural complexity. This allows engineers to focus on meaningful behavioral differences rather than redundant paths.
- **Path Identification:** The system analyzes the reduced graph to identify execution paths that are most valuable for testing. These paths are selected to maximize coverage of system behavior while avoiding unnecessary duplication.
- **Test Generation:** Based on the identified paths, MidGARD generates structured test sequences that can be used for regression testing. These tests are designed to provide progressively deeper coverage of the system.

Impact on Workflow

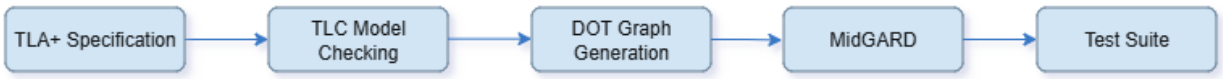
MidGARD is expected to significantly improve the efficiency of the existing SanDisk workflow. By automating the analysis of large state graphs, the system eliminates the need for engineers to manually interpret complex DOT outputs. Instead, engineers are provided with directly usable test sequences, enabling faster and more focused regression testing.

This shift reduces the time spent on graph analysis, increases overall testing efficiency, and lowers the reliance on specialized expertise. As a result, a broader range of engineers can effectively work with formal verification outputs, improving both productivity and accessibility within the workflow.

Trade-offs and Considerations

While MidGARD offers substantial benefits, there are some trade-offs to consider. The system introduces additional preprocessing time to analyze and reduce the input graphs before generating tests. Additionally, the reduction and path selection processes may deprioritize or exclude certain low-value edge cases in favor of higher-impact test scenarios.

However, these trade-offs are intentional and align with the system's goal of improving practical usability. By focusing on meaningful coverage and reducing redundancy, MidGARD provides a more efficient and scalable approach to test generation without significantly impacting overall correctness or reliability.



Functional Requirements

Domain-Level Requirements

- **DR-1:** The system shall accept a TLA+ specification and produce a directed state graph suitable for analysis.
- **DR-2:** The system shall reduce the state graph to a computationally manageable size without losing meaningful test diversity.
- **DR-3:** The system shall generate an ordered set of tests providing progressively deeper coverage of the state graph.
- **DR-4:** The system shall allow user control over generation parameters, including seed, test count, extension versus regeneration, and stopping behavior.

FR-1: TLA+ Parsing and Graph Construction (DR-1)

These requirements govern the system's intake of TLA+ specifications and production of an annotated directed state graph.

- **FR-1.1:** The system shall accept a valid, satisfiable TLA+ specification file as input.
- **FR-1.2:** The system shall invoke an external TLC model checker to validate the input specification and shall report any errors or malformed constructs to the user before proceeding.
- **FR-1.3:** The system shall generate a directed graph of states and transitions from the validated specification, where nodes represent state predicates and edges represent actions.
- **FR-1.4:** Each node in the generated graph shall carry metadata derived from the TLA+ specification, including the variable state associated with that state predicate.
- **FR-1.5:** Each edge in the generated graph shall carry metadata derived from the TLA+ specification, including the action name that labels the corresponding transition.

Planned Implementation Notes:

This functionality will likely be implemented through integration with the TLC model checker, which will validate the input TLA+ specification and produce graph output in Graphviz DOT format. The system will then parse the DOT file into an internal Rust graph representation, where each node represents a state predicate and each edge represents an action-labeled transition. Metadata from the TLA+ specification, including variable states and action labels, will be stored alongside the graph structure so later reduction, coverage, and test generation stages can operate on both graph topology and domain-specific information.

FR-2: Graph Reduction (DR-2)

These requirements specify the three-layer reduction pipeline that simplifies the state graph while preserving meaningful test diversity.

FR-2.1: Domain Reduction

Domain reduction collapses states that are considered indistinguishable, permanently removing redundant tests from the space of possibilities.

- **FR-2.1.1:** The system shall accept programmer annotations that specify metadata-driven collapse rules (e.g., a directive that a variable should be reduced modulo some value) and shall merge states accordingly.
- **FR-2.1.2:** The system shall perform strong bisimulation reduction, merging states that are indistinguishable with respect to their transition structure and action labels.
- **FR-2.1.3:** If strong bisimulation does not achieve sufficient reduction, the system shall support branching bisimulation reduction as a less strict alternative capable of further merging.
- **FR-2.1.4:** The system shall report reduction statistics to the user after domain reduction completes, including the number of states and transitions before and after reduction.

FR-2.2: Lossless Structural Decomposition

Lossless structural decomposition divides the graph into independently addressable subproblems without discarding any structural information.

- **FR-2.2.1:** The system shall identify strongly connected components (SCCs) within the state graph using Tarjan's algorithm.
- **FR-2.2.2:** The system shall condense each SCC into a single representative node, producing a directed acyclic graph (DAG) of condensed subproblems.
- **FR-2.2.3:** The system shall track boundary connections between each condensed component and its parent graph, recording which edges cross subproblem boundaries.
- **FR-2.2.4:** The system shall generate an independent sub-program representation for each subproblem, such that each subproblem can be solved in isolation.

FR-2.3: As-Needed Visibility Filtering

Visibility filtering temporarily hides graph elements that are irrelevant to the current coverage phase, reducing the working problem size without permanent information loss.

- **FR-2.3.1:** The system shall identify and temporarily remove edges and vertices that are unnecessary for the active coverage criterion (e.g., transitive reduction during node coverage).
- **FR-2.3.2:** Filtered elements shall be restorable when the system transitions to a subsequent coverage phase that requires them.

FR-2.4: As-Needed Structural Decomposition

When subproblems remain too large after initial decomposition, the system applies further structural decomposition as needed.

- **FR-2.4.1:** The system shall estimate the time and space cost of solving a subproblem directly versus decomposing it further, and shall only pursue further decomposition when it meaningfully reduces cost.
- **FR-2.4.2:** The system shall implement community detection using the Girvan-Newman algorithm to divide large subproblems into smaller communities based on connectivity.
- **FR-2.4.3:** The system shall track boundary connections introduced by community-based decomposition, recording cross-boundary edges and their associated requirements.
- **FR-2.4.4:** The system shall support both serialized subproblem solving (one subproblem at a time, to conserve memory) and interleaved subproblem solving (multiple subproblems active concurrently, to improve heuristic quality).

Planned Implementation Notes:

The graph reduction pipeline will be implemented as a sequence of Rust modules that operate on the internal graph representation. Domain reduction will apply programmer-provided annotations before structural reductions are performed. Strongly connected components will be identified using Tarjan's algorithm, and large remaining subproblems may be further divided using Girvan-Newman community detection when cost estimates indicate that additional decomposition would improve performance. The system should maintain mapping data between the original graph, reduced graph, and decomposed subproblems so that reduced or hidden elements can be restored or traced when needed.

FR-3: Test Generation (DR-3)

These requirements specify the greedy, heuristic-driven test generation engine and its phased coverage strategy.

FR-3.1: Heuristic Initialization and Management

- **FR-3.1.1:** The system shall initialize a potential field over all nodes using a centrality-based estimate of maximum node potential (e.g., Brandes' algorithm).
- **FR-3.1.2:** The system shall maintain an edge value field representing the direct contribution of each edge to unmet coverage goals.
- **FR-3.1.3:** The system shall maintain a node potential field representing the quality of each node's best outgoing path toward unmet requirements.
- **FR-3.1.4:** After each test is generated, the system shall update the heuristic fields through backwards propagation, using a max-heap to process nodes in order of descending potential.
- **FR-3.1.5:** When the system transitions between coverage phases, it shall re-initialize the heuristic using cumulative coverage counts from the prior phase to estimate how much of the new criterion is already satisfied.

FR-3.2: Greedy Test Path Construction

- **FR-3.2.1:** The system shall generate test paths one at a time, constructing each path by walking the graph guided by the current heuristic.
- **FR-3.2.2:** At each step during path construction, the system shall select the next edge based on a combination of edge value and adjacent node potential, favoring transitions that satisfy unmet requirements and lead toward high-potential regions.
- **FR-3.2.3:** Edge selection shall incorporate pseudorandom tiebreaking based on a deterministic seed, so that the same seed produces the same test suite.
- **FR-3.2.4:** The system shall support an optional local coverage heuristic that considers only the most recently generated tests, ensuring new tests are distinct from their immediate neighbors as well as the full preceding set.
- **FR-3.2.5:** When the local coverage heuristic is enabled, the system shall accept a configurable weight parameter controlling the balance between global and local diversity.

FR-3.3: Phased Coverage Sequence

The system progresses through coverage criteria in a fixed sequence, ensuring that fundamental coverage goals are met before pursuing deeper criteria.

- **FR-3.3.1:** Phase 1 — Node Coverage: The system shall generate tests until every reachable state in the (reduced) graph has been visited at least once.
- **FR-3.3.2:** Phase 2 — Edge Coverage: After node coverage is satisfied, the system shall generate tests until every transition in the (reduced) graph has been exercised at least once.
- **FR-3.3.3:** Phase 3 — Loop Coverage: After edge coverage is satisfied, the system shall generate tests systematically covering loops to approximate simple loop coverage.
- **FR-3.3.4:** Phase 4 — Further Test Generation: After loop coverage is satisfied up to a degree specified in the config file, the system shall switch to a general-purpose test generation algorithm.

Planned Implementation Notes:

The test generation engine will be implemented as a greedy, heuristic-guided traversal system over the reduced graph or active subproblem. Rust data structures such as priority queues or max-heaps may be used to manage node potential and edge value updates efficiently. Deterministic pseudorandom behavior will be controlled through a configurable seed so generated test suites can be reproduced. The system will progress through node coverage, edge coverage, loop coverage, and general-purpose generation phases while updating coverage counts and heuristic fields after each generated test.

FR-4: User Controls and Interface (DR-4)

These requirements specify the controls available to the user for configuring, running, and managing test generation.

- **FR-4.1:** The system shall read a config file with the input TLA+ specification and a user-provided seed for pseudorandom generation. If no seed is provided, the system shall generate one randomly and report it to the user.
- **FR-4.2:** The system shall accept additional commands via a command line interface.
- **FR-4.3:** The system shall allow the user to specify a fixed number of tests to generate.
- **FR-4.4:** The system shall allow the user to specify a maximum number of tests along with a target coverage criterion, stopping generation when either the criterion is met or the maximum count is reached.
- **FR-4.5:** The system shall support extending an existing test suite by appending new tests that build on the coverage already achieved by the existing suite.
- **FR-4.6:** The system shall support regenerating a new test suite from scratch using a different seed.
- **FR-4.7:** The system shall allow the user to stop generation at any point and receive the complete set of tests generated up to that point.

Planned Implementation Notes:

User controls will likely be implemented through a command-line interface and configuration file. The configuration file will define required inputs such as the TLA+ specification path, seed, test count, coverage target, and heuristic settings. Command-line flags will allow users to run common workflows such as generating a new suite, extending an existing suite, regenerating with a new seed, or stopping generation. The CLI should also provide help output and validation messages so users can understand incorrect inputs before long-running generation begins.

FR-5: Decomposition Assembly (DR-2, DR-3)

These requirements specify how the system recombines results from decomposed subproblems into a coherent test suite.

- **FR-5.1:** The system shall reassemble tests from decomposed subproblems by joining path segments at boundary edges.
- **FR-5.2:** For SCC-based decomposition, the system shall extend child-component tests directly into parent-graph paths at the corresponding boundary connections.
- **FR-5.3:** For community-based decomposition, the system shall identify and solve an additional subproblem covering requirements that span multiple community boundaries, ensuring that decomposition does not introduce coverage gaps.
- **FR-5.4:** The system shall propagate heuristic data between subproblems: solved subproblems shall report boundary heuristic data forward, and (when interleaved solving is used) active subproblems shall exchange heuristic data bidirectionally.

Planned Implementation Notes:

Decomposition assembly will be implemented by preserving boundary metadata during each decomposition step. When subproblems are solved independently, the system will use recorded boundary edges to reconnect generated path segments into coherent full test paths. For

SCC-based decomposition, child component paths can be extended into parent graph paths at known boundary connections. For community-based decomposition, additional cross-boundary requirements may need to be solved separately to ensure that decomposition does not leave gaps in coverage.

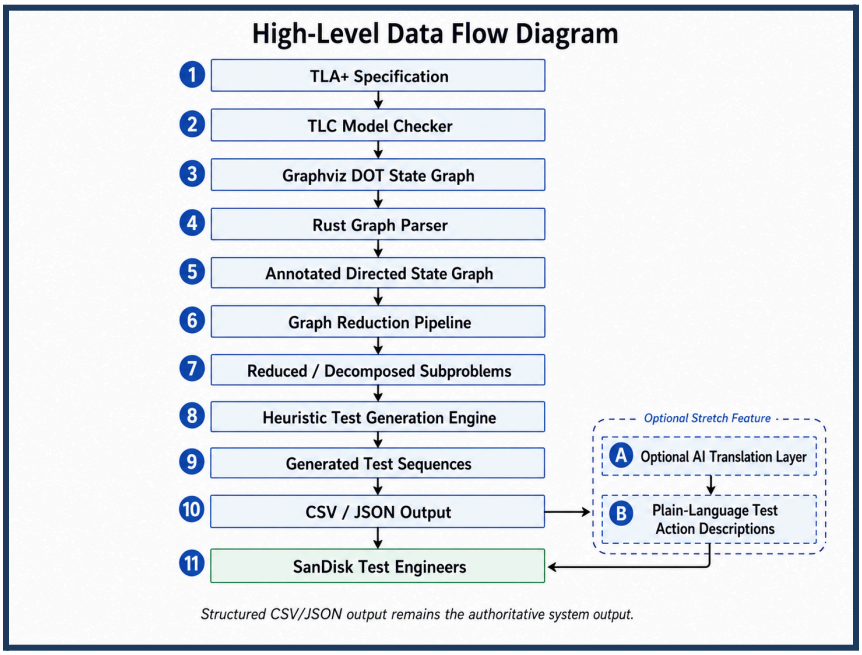
FR-6: Session Persistence (DR-4)

These requirements address saving and restoring generation state so that work can continue across sessions.

- **FR-6.1:** The system shall support saving the current generation state — including the reduced graph, heuristic fields, coverage progress, and generated tests — to persistent storage.
- **FR-6.2:** The system shall support loading a previously saved generation state and resuming test generation from where it left off, without recomputing the reduction pipeline or prior heuristic updates.
- **FR-6.3:** The system shall support loading and solving individual subproblems in isolation from a saved state, enabling serial solving workflows that conserve memory.

Planned Implementation Notes:

Session persistence will likely be implemented through serialized state files that store the reduced graph, active subproblems, heuristic fields, coverage progress, configuration values, and generated test sequences. The storage format should support partial loading so the system can reload a single subproblem without loading the entire graph into memory. To protect saved progress, the system should use atomic write behavior, such as writing to a temporary file and then renaming it after the save completes successfully.



Performance Requirements

Operating Environment

- **OE-1:** The reference hardware platform for all performance targets in this section is a modern workstation-class machine with at least 8 logical cores, 32 GB of RAM, and SSD-backed storage. Performance targets may not hold on hardware below this baseline.
- **OE-2:** The system shall run on 64-bit Linux. Other operating systems (macOS, Windows via WSL) are secondary targets and are not required to meet the performance targets stated in this section.
- **OE-3:** The system assumes the availability of OpenJDK >= 11.0.6 (non-Oracle distribution) for TLC model-checker invocation.
- **OE-4:** The system shall accept well-formed TLA+ specification files as input. Behavior on malformed input is governed by FR-1.2 (validation and error reporting); the performance targets in this section assume validated input.
- **OE-5:** The system shall accept Graphviz DOT format files as the interchange format for directed state graphs produced by TLC.

PR-1: Scalability

- **PR-1.1:** The system shall be capable of processing state graphs produced by industry-scale TLA+ specifications. As a baseline target, the system shall handle graphs on the order of 10^7 nodes and 10^8 edges without exhausting available memory on the reference hardware defined in OE-1.
- **PR-1.2:** When a full graph does not fit in available memory, the system shall automatically decompose the problem into subproblems that can each be solved within available memory, as specified by the functional decomposition requirements.

PR-2: Speed

- **PR-2.1:** With a graph on the order of 10^6 nodes and 10^7 edges, the system shall complete pre-processing within 30 minutes and edge coverage within 15 minutes after pre-processing.
- **PR-2.2:** With a graph on the order of 10^7 nodes and 10^8 edges, the system shall complete pre-processing within 24 hours and obtain edge coverage within 8 hours without sacrificing heuristic quality.

PR-3: Session Persistence Performance

- **PR-3.1:** Saving and loading generation state (FR-6.1, FR-6.2) shall not require recomputation of the graph reduction pipeline or previously completed heuristics.
- **PR-3.2:** The serialized state format shall support partial loading, so that an individual subproblem can be loaded and solved without loading the full graph into memory (FR-6.3).

PR-4: Resource Management

- **PR-4.1:** The system shall use concurrent execution, such as multithreading, to utilize available processing cores during computationally intensive operations including graph reduction, heuristic initialization, and test generation.
- **PR-4.2:** Peak memory consumption during any single operation shall not exceed the memory required to hold the active subproblem's graph representation, heuristic fields, and a bounded working set. The system shall release memory for completed subproblems before loading subsequent ones during serialized solving.
- **PR-4.3:** Permanent memory usage shall not exceed 30 GB on the reference hardware platform during normal operation.
- **PR-4.4:** The system shall document expected storage requirements for generated graphs, serialized session states, intermediate subproblem files, and final CSV or JSON test sequence outputs.

PR-5: Responsiveness

- **PR-5.1:** The system shall acknowledge user commands within 2 seconds of invocation. Long-running operations shall provide periodic progress indicators so that the user can distinguish ongoing computation from an unresponsive process.
- **PR-5.2:** When the user issues a stop command, the system shall halt generation and present the current test suite within 10 seconds.

PR-6: Reproducibility

- **PR-6.1:** Given the same input specification, seed, and configuration parameters, the system shall produce identical test suites across runs on the same platform. All pseudorandom behavior shall be fully determined by the user-provided or system-generated seed.

PR-7: Coverage Performance

- **PR-7.1:** After the node coverage phase, 100% of reachable states in the reduced graph shall appear in at least one generated test.
- **PR-7.2:** After the edge coverage phase, 100% of transitions in the reduced graph shall appear in at least one generated test.

- **PR-7.3:** Domain reduction shall reduce the graph such that no two remaining states are bisimulation-equivalent. The system shall report the reduction ratio (original states to reduced states) to allow verification.
- **PR-7.4:** The greedy farthest-point-first strategy shall produce test suites with measurably lower redundancy than random path selection. Specifically, for the same number of tests, the heuristic-guided suite shall achieve equal or greater coverage across all active criteria compared to a randomly generated baseline using the same graph and seed.

PR-8: Usability

- **PR-8.1:** The system shall provide a help command or flag that documents all available commands, options, and configuration parameters with brief descriptions and usage examples.
- **PR-8.2:** Common workflows — generate a new suite, extend an existing suite, regenerate with a different seed — shall each be achievable with a single command invocation and no more than 5 flags or arguments.
- **PR-8.3:** The system shall provide sensible defaults for all optional parameters (seed, test count, coverage criterion, heuristic weight) so that a minimal invocation requiring only the input specification path produces a useful test suite.

PR-9: Reliability

- **PR-9.1:** The system shall not crash, deadlock, or produce corrupted output when processing graphs up to the scale defined in PR-1.1 on the reference hardware. If an operation exceeds available memory, the system shall report the condition and attempt graceful degradation (e.g., falling back to serialized subproblem solving) rather than terminating abruptly.
- **PR-9.2:** The system shall handle concurrent thread execution without data races. All shared data structures accessed during multithreaded operations shall be protected by Rust's ownership and borrowing guarantees or explicit synchronization primitives.
- **PR-9.3:** If the system is interrupted during state serialization, previously saved state files shall not be corrupted. The system shall use atomic write strategies (e.g., write-to-temporary then rename) to protect persisted state.

Environmental Requirements

ER-1: Language and Dependencies

- **ER-1.1:** The system shall be implemented in Rust.
- **ER-1.2:** All third-party dependencies shall be documented with their versions, and all third-party dependencies shall use licenses compatible with proprietary distribution.

ER-2: TLA+ Toolchain Integration

- **ER-2.1:** The system shall integrate with the TLC model checker (requiring OpenJDK >= 11.0.6, non-Oracle distribution) to validate TLA+ specifications and generate Graphviz DOT output.
- **ER-2.2:** The system shall accept Graphviz DOT format files as the interchange format for directed state graphs produced by TLC.
- **ER-2.3:** The system shall handle DOT files that include self-referential edges (a consequence of TLA+ stuttering invariance) by providing an option to remove or retain such edges before analysis.

ER-3: Development Environment

- **ER-3.1:** The system shall be developed and tested using VSCode with the following extensions: Rust-analyzer, TLA+ (TLA+ Foundation), Graphviz Interactive Preview (tintinweb), and Live Share (Microsoft).
- **ER-3.2:** The complete codebase and documentation shall be maintained in a private GitHub repository accessible to the SanDisk project sponsors.

ER-4: Output Format

- **ER-4.1:** Generated test sequences shall be output in a structured, machine-readable format (e.g., JSON or CSV) so that SanDisk product teams can consume them for regression deployment planning.
- **ER-4.2:** The system shall produce documentation sufficient for SanDisk product teams to use as a reference manual, including usage instructions, input/output format descriptions, and configuration options.

Optional Features/Stretch Goals

SF-1: AI-Assisted Test Sequence Translation

If time permits, the system may provide an optional AI-assisted translation layer that converts generated CSV or JSON test sequences into plain-language action steps for test engineers. This feature would be supplemental only, and the structured CSV or JSON output would remain the authoritative system output.

Potential Risks

While MidGARD aims to improve SanDisk's validation workflow, there are several risks that could impact the effectiveness, reliability, and overall usefulness of the system. These risks are most relevant because they directly affect how engineers interpret results and make decisions about system validation and product quality.

Misleading Test Coverage

MidGARD reduces a large state-space graph into a smaller set of test sequences. While this improves usability, it introduces the risk that important behaviors are excluded.

Potential impacts include:

- Critical edge cases or rare behaviors may not be tested
- Engineers may assume full coverage when gaps still exist
- Undetected bugs could reach production systems
- Reduced reliability of storage products in real-world use

Over-Reliance on Automated Outputs

By automating test generation, MidGARD reduces manual effort, but may also lead to blind trust in the results.

Potential impacts include:

- Engineers may not fully understand system behavior
- Incorrect or incomplete test sequences may go unnoticed
- Reduced critical evaluation of testing strategies
- False confidence in validation results

Graph Processing Errors

MidGARD modifies graphs through simplification and filtering. Errors in these steps could affect correctness.

Potential impacts include:

- Important states or transitions may be removed
- Generated tests may not reflect actual system behavior
- Validation results may be misleading
- Debugging becomes more difficult due to altered data

Scalability and Performance

Large real-world graphs may still be difficult to process efficiently.

Potential impacts include:

- Long execution times or high memory usage
- Reduced usability in real engineering workflows
- Limited adoption if performance is insufficient

Integration and Usability Risks

MidGARD must fit into an existing workflow and produce outputs engineers can use.

Potential impacts include:

- Tool compatibility issues (TLA+, TLC, Rust, Graphviz)
- Increased workflow complexity instead of simplification
- Outputs that are difficult to interpret or apply
- Low adoption if engineers prefer existing methods

Technological and Competitive Risk

The field is evolving, and alternative tools may emerge.

Potential impacts include:

- MidGARD may become less relevant over time
- Competing solutions may offer better performance or usability
- Additional development may be needed to stay competitive

Risk Mitigation Strategies

To reduce these risks, the system will incorporate:

- Transparency in how test sequences are generated
- User-configurable parameters for coverage and output control
- Validation against known test cases and existing methods
- A modular design to support future improvements and tool changes
- Continuous feedback from engineers and stakeholders

Overall, while these risks could impact the system's effectiveness, they are manageable with careful design and validation. By focusing on accuracy, usability, and integration, MidGARD can still provide meaningful improvements to SanDisk's validation process while minimizing negative outcomes.

Project Plan

At this stage of the project, our team has identified a set of major milestones that will guide the development of MidGARD. These milestones are based on the core functional requirements of the system and support a step-by-step implementation of the full pipeline, from input processing to test generation. While this plan is high-level, it provides a clear roadmap for development over the coming months.

1. Environment Setup & Tool Familiarization

- Install and configure TLA+, the TLC model checker, Rust, and required libraries (e.g., petgraph)
- Review documentation for each tool
- Build small test examples to understand individual components

2. TLA+ Integration & DOT File Generation

- Run TLA+ specifications through the TLC model checker
- Generate DOT files representing system state graphs
- Ensure reliable production of input data for MidGARD

3. Graph Parsing & Representation

- Read DOT files and convert them into an in-memory graph structure
- Use Rust libraries (e.g., petgraph) for implementation
- Preserve metadata such as state labels and transition actions

4. Graph Preprocessing & Simplification

- Remove redundant edges (e.g., self-loops)
- Filter unnecessary transitions
- Reduce graph complexity to improve usability and performance

5. Graph Structural Analysis

- Identify strongly connected components (SCCs)
- Break the graph into smaller, manageable substructures
- Prepare the graph for efficient downstream processing

6. Test Path Generation

- Implement traversal algorithms to generate execution paths
- Maximize coverage of system behavior
- Minimize redundant or low-value paths

7. Test Prioritization & Output Formatting

- Organize test sequences based on coverage goals
- Output results in structured formats (e.g., JSON or CSV)
- Ensure outputs are usable for regression testing

8. User Controls & System Refinement

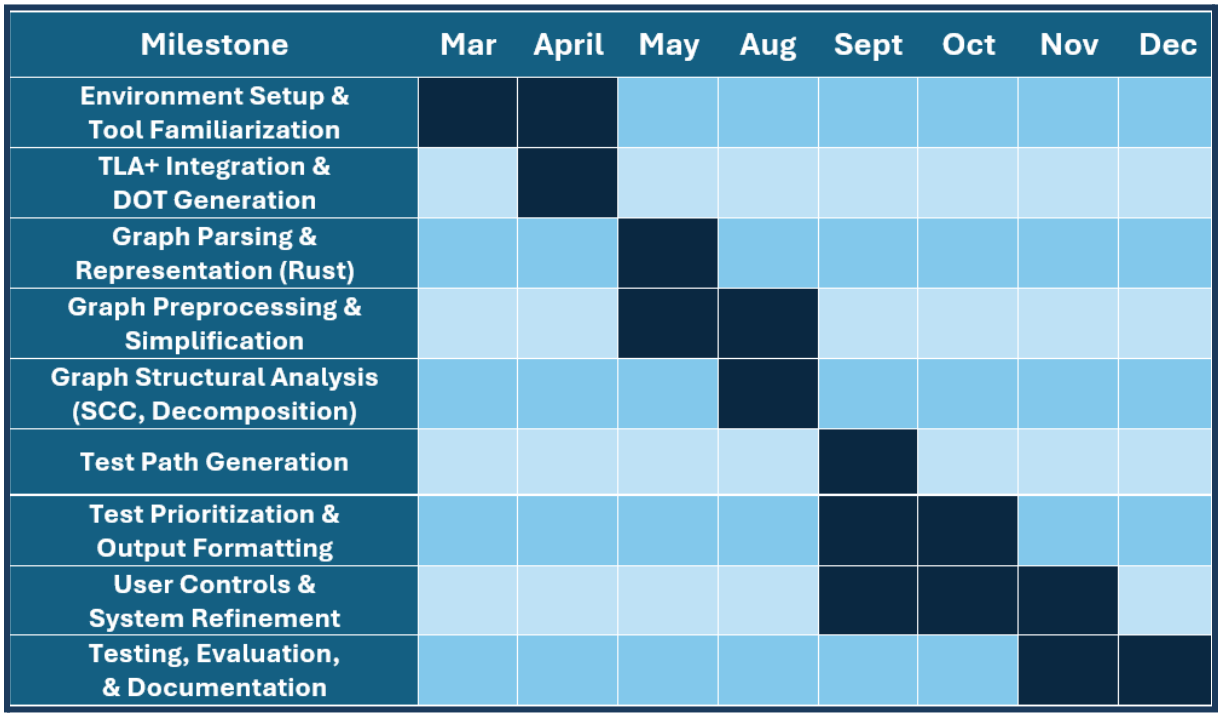
- Add configurable parameters (e.g., seed, number of tests, coverage criteria)
- Improve usability and consistency across runs
- Refine system behavior based on testing and feedback

9. Testing, Evaluation & Documentation

- Test the system on increasingly complex inputs
- Evaluate performance, scalability, and correctness
- Finalize documentation and prepare deliverables for the sponsor

These milestones are distributed across the capstone timeline, with early phases focused on setup and integration, and later phases focused on algorithm development and system refinement. A Gantt chart (included below) provides a visual representation of this schedule and highlights dependencies between phases.

Overall, this plan follows an incremental development approach, ensuring that each component is working before moving forward. As the project progresses, the plan will be refined with more detailed tasks and adjusted based on implementation results and sponsor feedback.



Conclusion

The increasing complexity of modern storage systems makes ensuring correctness and reliability more challenging than ever. As described in this document, SanDisk's current workflow already uses powerful tools like TLA+ and the TLC model checker to formally verify system behavior. However, while these tools provide a complete and accurate representation of all possible system states and transitions, there is still a gap between verification and practical testing. The large and complex state graphs produced by this process are difficult to interpret and do not directly translate into actionable regression test strategies.

This document outlined that gap in detail by first explaining the current workflow and then identifying the key inefficiencies and missing elements. These include the difficulty of working with large state graphs, the lack of automated filtering and simplification, and the absence of a structured way to generate meaningful test sequences. Together, these issues create a process where valuable information is available but not fully utilized, leading to inefficiencies and increased reliance on manual effort and expert knowledge.

To address this problem, we introduced MidGARD as a solution that connects formal verification with practical testing. The system is designed to take the output of the existing workflow and transform it into a more usable form by simplifying graphs, identifying important execution paths, and generating structured regression test sequences. By doing this, MidGARD aims to reduce the need for manual analysis while improving the overall effectiveness and scalability of testing.

Throughout this document, we defined the key features and high-level processes of the system, including how it handles input data, performs graph reduction, and generates test outputs. We also discussed how the system integrates into the existing workflow and what trade-offs come with its design. These sections help show that the proposed solution is not only useful, but also realistic and aligned with the sponsor's needs.

Overall, this project represents an opportunity to improve how formal methods are used in practice. By making the results of formal verification more accessible and actionable, MidGARD can help engineers work more efficiently and with greater confidence in their test coverage. Moving forward, the team will continue refining the system design and begin implementation with the goal of delivering a tool that is both practical and impactful for SanDisk's validation process.

Glossary

The following terms are used throughout this document and may have specific meanings within the context of this project:

TLA+ (Temporal Logic of Actions):
A formal specification language used to describe system behavior mathematically. It allows engineers to model systems and verify correctness before implementation.

TLC Model Checker:
A tool used to execute and verify TLA+ specifications by exploring all possible system states and checking for correctness.

Rust Programming Language:
A systems programming language focused on performance, memory safety, and reliability. It is used in this project to implement graph processing and analysis.

Graphviz:
A tool used to visualize graphs. In this project, it is used to represent system state graphs generated from TLA+ specifications.

Regression Testing:
A testing process used to ensure that changes to a system do not introduce new errors or break existing functionality.

MidGARD (Model-initiated di-Graph Analysis for Regression Deployment):
The system being developed in this project, which analyzes state graphs generated from TLA+ and produces optimized regression test sequences.

DOT Files:
A file format used by Graphviz to describe graphs, including nodes (states) and edges (transitions). Generated by TLC Model Checker.

SCC (Strongly Connected Components):
Groups of nodes in a directed graph where each node is reachable from every other node in the same group.

DAG (Directed Acyclic Graph):
A directed graph with no cycles. Often created after simplifying or decomposing a more complex graph.

Tarjan's Algorithm:
An algorithm used to identify strongly connected components in a graph efficiently.

Girvan-Newman Algorithm:
An algorithm used to detect communities within a graph by progressively removing edges.

Brandes' Algorithm:
An algorithm used to calculate centrality measures in a graph, helping identify important nodes.