Project MidGARD
Technology Feasibility

Project Sponsors:
Chris Ortiz and Rex Jackson (SanDisk)

Faculty Mentor:
Jeevana Swaroop Kalapala

Team Odin:
Lacy Hamilton
Andrew Gajewski
Myles Hill
Skyler Guard

Last Updated April 2, 2026

Table of Contents

1 Introduction

Modern computing systems are becoming increasingly complex, particularly in environments where reliability and correctness are critical. In industries such as data storage, networking, and distributed systems, software errors discovered late in the development lifecycle can lead to costly delays, extensive debugging efforts, and potential failures in deployed products. As systems grow in complexity, ensuring that specifications are correct and thoroughly tested before deployment becomes an essential challenge for engineering teams.

One approach that has gained traction in industry is the use of formal specification languages, such as TLA+ (Temporal Logic of Actions). TLA+ allows engineers to mathematically describe the behavior of a system and verify that the design satisfies key correctness properties using model checking tools such as the TLC model checker. Organizations including Microsoft, Amazon Web Services, and others have adopted formal verification techniques to improve system reliability and reduce the likelihood of subtle design errors. However, while formal verification can confirm that a system specification is logically correct, translating those verified specifications into practical and effective regression testing strategies remains a challenge.

Our project sponsor, SanDisk, is exploring ways to incorporate formal verification into their design and validation processes. Currently, engineers can generate directed graphs of system states and transitions from verified TLA+ specifications. These graphs represent all possible valid behaviors of the system. However, these graphs often become extremely large and complex, making it difficult to determine which execution paths should be used to guide regression testing. As a result, test planning may still rely heavily on manual reasoning and engineering intuition.

To address this challenge, our capstone team is developing MidGARD (Model-initiated di-Graph Analysis for Regression Deployment). MidGARD is designed to analyze directed graphs generated from formally verified TLA+ models and automatically identify a finite set of optimized test sequences that provide strong coverage of the system's state space. By leveraging graph analysis techniques, the system aims to assist engineers in generating regression test plans that are both efficient and mathematically grounded.

While the vision for MidGARD provides a promising approach for improving regression testing strategies, it is important to ensure that the proposed system can realistically be implemented within the technological constraints of the project. Early in the design process, identifying potential challenges and evaluating viable technological solutions helps reduce risk and ensures that development proceeds on a solid foundation.

This Technological Feasibility Analysis examines the major technical challenges associated with building the MidGARD system. In the sections that follow, we first identify the key technological hurdles and design decisions that must be addressed in order to implement the system successfully. We then analyze potential technological solutions for each challenge, evaluate the strengths and limitations of those alternatives, and provide justification for the approaches selected by the team. Together, this analysis establishes a realistic and well-supported path toward the successful implementation of the MidGARD system.

2 Technological Challenges

The creation of the MidGARD (Model-initiated di-Graph Analysis for Regression Deployment) tool brings up technological challenges that need to be addressed in order for our team to deliver a product that meets our and the client's expectations. The following high-level requirements were identified:

2.1 Managing Large State Spaces

Challenges: TLA+ Specifications can generate very large graphs, even near infinite.

Details: Complex specifications for technologies such as NVMe and PCIe may produce extremely large state graphs. And in some cases, the TLC tool may fail to generate a DOT file due to the size and complexity of the state space. MidGARD must address how to manage these large graphs, whether through abstraction or other methods. The VIEW and ALIAS features in TLC for example.

2.2 Representation and Generation of Graphs

Challenges: Graphs generated from TLA+ specifications may contain thousands or potentially millions of nodes and edges.

Details: Selecting an efficient data structure to represent the graph in memory will be critical. The system must determine how to load, store, and manipulate these large graphs while maintaining acceptable performance and memory usage. This includes choosing the right graph library or implementing our own custom graph structure.

2.3 Meaningful Test Sequences Extraction

Challenges: Directed graphs produced from TLA+ specifications can represent an extremely large number of possible execution paths. Determining the best algorithms to analyze the directed graph and produce a finite set of action sequences with strong coverage is a major challenge.

Details: The system must determine how to generate test sequences that balance completeness, efficiency, and practicality for regression testing. Potential techniques may involve graph traversal algorithms, strongly connected component detection (such as Tarjan's algorithm), Hamiltonian path approximations, or other graph coverage techniques. The challenge lies in determining which algorithms or combination of algorithms will produce sequences that maximize coverage while remaining computationally feasible.

2.4 Redundant and Irrelevant Transitions Filtering

Challenges: The DOT file generated by TLC may contain unnecessary or redundant information. TLA+ specifications are stuttering invariant, meaning the correctness of a system does not change if you repeat states. These self-referencing edges bloat generated directed graphs.

Details: MidGARD has to determine how to parse the DOT file, identify these edges, and selectively remove them without losing important structural information in the graph. Implementing reliable parsing and transformation of Graphviz DOT will be a technical challenge.

2.5 Tool Integration

Challenges: The integration of many different technologies, including TLA+, TLC Model Checker, Rust Programming Language, etc.

Details: The system must be able to interact with TLA+ specifications and trigger the TLC model checker correctly. We will have to determine how our tool will read or reference and integrate with the TLA+ toolchain. As well as how to programmatically execute TLC, capture its output, and detect failure states. Rust introduces its own challenges with a steep learning curve. We must learn how to structure the system architecture, select appropriate Rust libraries for graph processing, and implement efficient algorithms while maintaining code reliability. MidGARD aims to create an automated pipeline that moves from TLA+ specification through the steps to graph analysis. Outputs from each stage have to be correctly passed to the next stage without a hitch.

3 Technology Analysis

With the major technological challenges identified, we can turn to a detailed examination of each. The goal of this section is to evaluate potential solutions, compare alternatives against criteria specific to MidGARD's requirements, and arrive at well-supported technology decisions that will serve as the foundation for implementation.

The challenges span different areas: working within TLC's constraints to manage large state spaces, selecting the right Rust libraries for graph representation, designing the algorithms at the core of MidGARD's test generation, deciding how to filter redundant transitions, and integrating a diverse toolchain into a coherent pipeline. In the subsections that follow, we analyze each area by describing the issue, identifying desired characteristics, presenting and evaluating alternatives, and justifying the approach we selected. Where applicable, we outline demonstrations we plan to build to validate our choices before full implementation begins in the fall.

3.1 Managing Large State Spaces

Issue:

TLA+ specifications for complex protocols like NVMe and PCIe can produce state graphs with millions or billions of states, often too large for TLC to export as a DOT file. Since MidGARD's entire pipeline depends on receiving a graph as input, we need a reliable strategy for reducing the state space to a tractable size while preserving enough structural detail for meaningful test generation.

Desired Characteristics:

Produces a graph small enough for TLC to export and MidGARD to analyze; preserves essential state-transition structure; requires no modifications to TLC; gives the user control over the degree of abstraction.

Alternatives:

- **TLC VIEW and ALIAS abstractions.** TLC's built-in VIEW keyword defines an equivalence relation over states, collapsing states that differ only in excluded variables. ALIAS reformats state output. Both are configured in the TLC model file and are part of the open-source TLA+ toolchain. The SanDisk project description specifically identifies these as the intended approach.
- **TLC simulation mode.** TLC's -simulate flag generates random execution traces up to a configurable depth instead of performing exhaustive BFS. This produces a finite sample of behaviors without constructing the full state graph. Built into TLC, no additional tooling required.
- **Apalache symbolic model checker.** Apalache translates TLA+ specifications into SMT constraints solved by Z3, which can handle certain state spaces intractable for TLC. Free and open-source (Apache 2.0), developed at Informal Systems.

Analysis:

VIEW and ALIAS are the most direct solution. The user retains precise mathematical control over what is preserved, and the result is still a complete state-transition graph, so MidGARD's downstream analysis works without modification. TLC simulation mode is a viable fallback but produces sampled traces rather than a complete graph, which means full-coverage guarantees are lost and a substantially different analysis pipeline would be needed. Apalache does not generate DOT files, does not support the full TLA+ language, and is still described by its developers as experimental, introducing risk that is hard to justify when TLC's own features address the problem.

Chosen Approach:

We will use TLC's VIEW and ALIAS configuration options. MidGARD will accept VIEW and ALIAS definitions as input and pass them to TLC during model checking. For our technology demo, we will construct a TLA+ specification with a deliberately large state space, apply VIEW to reduce it, and confirm TLC generates a DOT file that MidGARD can parse.

3.2 Representation and Generation of Graphs

Issue:

MidGARD must parse TLC's DOT output and build an in-memory directed graph supporting efficient traversal and algorithmic analysis. DOT files may contain thousands to millions of nodes and edges, so memory efficiency and traversal speed matter. The representation must also preserve DOT metadata (state-predicate labels on nodes, action labels on edges) for human-readable test output.

Desired Characteristics:

O(|V| + |E|) memory usage; fast adjacency queries; support for directed graphs with arbitrary node/edge data; DOT format parsing; availability of common graph algorithms (SCC, DFS, BFS); permissive open-source license.

Alternatives:

- **petgraph.** The most widely adopted Rust graph library (9M+ monthly downloads), with adjacency-list storage, multiple graph types including StableGraph (stable indices across removals), built-in algorithms (Tarjan's SCC, DFS, BFS, Dijkstra's), and a dot_parser feature for importing DOT files. Dual-licensed MIT/Apache-2.0.
- **graphlib.** A simpler Rust graph library with an API modeled after standard Rust collections. MIT licensed. No DOT parser, no SCC implementation, and a much smaller user base.
- **Custom implementation.** Build our own adjacency-list structure and DOT parser tailored to TLC output. Maximum control, but significant development effort and high bug risk.

Analysis:

Petgraph satisfies every requirement directly. Its dot_parser handles DOT import, its StableGraph type supports node/edge removal during preprocessing without index invalidation,

and its algorithm module includes Tarjan's SCC, which is central to our analysis strategy. Graphlib lacks both a DOT parser and SCC decomposition, meaning we'd implement significant functionality ourselves. A custom implementation carries the highest risk and cost; writing correct graph algorithms is substantial engineering work, and the standard Graphviz DOT format doesn't justify a custom parser.

Criterion	petgraph	graphlib	Custom
DOT Parsing	Built-in	None	Must build
Built-in algorithms	Extensive	Minimal	Must build
Community	Very high	Low	N/A
Development Speed	High	Medium	Very low

Chosen Approach:

Project MidGARD will use petgraph with dot_parser and stable_graph features enabled, representing the graph as StableGraph<String, String, Directed>. For our technology demo, we will parse a TLC-generated DOT file into a petgraph StableGraph and output summary statistics (node count, edge count, SCC count).

3.3 Meaningful Test Sequences Extraction

Issue:

The system must analyze directed graphs generated from TLA+ specifications and extract a finite set of meaningful execution sequences. The challenge is selecting appropriate graph traversal and analysis algorithms that provide strong coverage without excessive computational cost.

Desired Characteristics:

High Coverage: Generated sequences should cover as many states and transitions as possible.

Scalability: The approach should handle reduced but still potentially large graphs.

Practicality: The number of generated sequences should remain manageable for testing.

Deterministic Output: Results should be consistent and reproducible.

Efficiency: The approach should avoid unnecessary computation when generating sequences.

Non-Redundancy (Isomorphism Awareness): The system should avoid generating test sequences that are structurally equivalent, ensuring each sequence provides unique testing value.

Alternatives:

- **Traversal-Based Approaches:** Depth-First Search (DFS), Breadth-First Search (BFS)

- **Graph Analysis Techniques:** Strongly Connected Components (e.g., Tarjan's Algorithm), Cycle detection
- **Path Coverage Strategies:** All-paths coverage (impractical for large graphs), Edge coverage, Node coverage
- **Advanced Approaches:** Hamiltonian path approximations, Heuristic-based path selection

Analysis:

Basic traversal algorithms such as DFS and BFS are simple to implement and ensure that all reachable nodes are explored. However, on their own, they may generate redundant or overly long paths.

Strongly connected component (SCC) detection helps simplify the graph by grouping cycles, reducing complexity and making traversal more efficient. This is especially useful for systems with repeated states.

Full path coverage is not feasible due to exponential growth in the number of possible paths. Instead, edge or node coverage provides a practical balance between completeness and performance.

More complex approaches like Hamiltonian paths are computationally expensive and difficult to implement, making them less suitable for this project. Heuristic-based methods can improve efficiency but may add unnecessary complexity.

Many traversal-based approaches can generate redundant paths that are structurally equivalent but differ only in ordering or minor variations. These isomorphic paths do not provide additional testing value and can unnecessarily increase the number of generated sequences. Incorporating techniques to detect or reduce such redundancy is important for keeping the output concise and meaningful.

Chosen Approach:

The system will use a combination of SCC detection and traversal-based algorithms. First, strongly connected components will be identified to simplify the graph structure. Then, a modified DFS or BFS approach will be used to generate representative paths that achieve strong edge and node coverage.

This hybrid approach balances coverage and efficiency while remaining feasible to implement within the project timeline. It also ensures that the generated test sequences are practical for regression testing without overwhelming the user.

The system will incorporate basic filtering techniques to reduce redundant or structurally equivalent paths. Rather than explicitly solving graph isomorphism (which can be computationally expensive), the approach will focus on generating representative paths that minimize duplication while maintaining strong coverage.

3.4 Redundant and Irrelevant Transitions Filtering

Issue:

It is possible (and very likely) for generated graphs to contain redundant edges, self loops, and cycles that don't contribute any meaningful information.

Desired Characteristics:

- The ability to identify and remove unnecessary edges
- Must preserve important system behavior
- Reduction of graph complexity
- Original TLA+ spec remains unchanged

Alternatives:

- **Manual Filtering Rules –** Manually inspecting the generated graph and defining rules for removing transitions. While it is very simple, this approach does not scale well. For systems with hundreds or thousands of states, it becomes impractical and error prone.
- **Graph Simplification Algorithms –** More structured and automated approach to reducing graph complexity. Some common strategies are self loop & duplicate edge removal, reachability filtering, and cycle reduction. All of these algorithms are widely used in graph theory and aim to preserve the essential behavior of the system.
- **Post Processing Scripts –** Post-processing scripts provide a practical way to apply graph simplification techniques to the output of the TLC model checker. After generating a graph (e.g., in DOT format), a script can parse the graph data and apply filtering and transformation rules programmatically. Automation and repeatable graph cleanup is possible and very desirable with this method.

Analysis:

Manual filtering is not practical for large graphs due to its lack of scalability and susceptibility to human error.

Graph simplification algorithms provide a theoretically sound approach to reducing graph complexity, but implementing them directly within the modeling or verification process can introduce unnecessary complexity and reduce flexibility.

Post-processing scripts offer a balanced solution by applying graph simplification techniques after the graph has been generated. This approach also ensures that the original TLA+ specification remains unchanged, preserving the integrity of the formal model while allowing flexibility in how the resulting graph is analyzed. By leveraging common simplification techniques such as self-loop and duplicate edge removal, post-processing can significantly reduce noise without altering meaningful system behavior.

Chosen Approach:

We will implement automated graph post-processing to remove redundant transitions and simplify the graph structure. This approach enables us to apply well-established graph

simplification techniques in a flexible and scalable manner, ensuring that the resulting graph remains both manageable and representative of the system's behavior, without modifying the original TLA+ specification.

3.5 Tool Integration

Issue:

The system requires integration of multiple tools, including TLA+ for specification, the TLC model checker for simulation, a graph visualization tool, and the Rust programming language for implementation. Ensuring these tools work together in a seamless pipeline presents a key technical challenge.

Desired Characteristics:

- Compatibility: Tools must work well together and support integration through command-line interfaces or APIs.
- Automation Support: The system should allow programmatic execution (e.g., running TLC and capturing output automatically).
- Reliability: Tools should be stable and widely used to reduce unexpected failures.
- Performance: The system should efficiently handle model checking outputs and graph generation.
- Ease of Use: Tools should have sufficient documentation and community support to reduce development difficulty.

Alternatives:

- High-Level Modeling Language: TLA+ vs. PlusCal
- Model Simulation Engine: TLC vs. Apalache
- Graph Visualization: Graphviz vs. Cytoscape
- Programming Language: Rust vs. Python

Analysis:

TLA+ is preferred over PlusCal because it is the primary language used by the client and directly supported by the TLC model checker. While PlusCal offers a more algorithmic syntax, it ultimately translates to TLA+, making TLA+ the more direct choice.

TLC is selected over Apalache because it is the standard model checker for TLA+ and is more widely documented. Although Apalache offers symbolic model checking, TLC is simpler to integrate and sufficient for this project's needs.

Graphviz is chosen over Cytoscape due to its lightweight nature and ease of generating directed graphs programmatically. Cytoscape provides more advanced visualization features but is more complex and less suited for automated pipelines.

Rust is selected over Python because it offers better performance and memory safety, which is beneficial for handling graph processing and larger datasets. Although Rust has a steeper learning curve, it provides stronger guarantees for building reliable systems.

Chosen Approach:

The system will use TLA+ as the high-level modeling language, as it aligns with the client's existing workflow. The TLC model checker will be used to execute specifications and generate output data.

Graphviz will be used to visualize directed graphs because it is lightweight, easy to integrate, and supports automated generation of visual outputs.

Rust will be used as the primary programming language due to its performance, safety features, and suitability for building efficient data-processing pipelines.

Together, these tools will support an automated workflow that transforms TLA+ specifications into graph-based representations for analysis.

4 Technology Integration

In the preceding sections, we examined each of MidGARD's core technological challenges in isolation and arrived at concrete decisions: TLA+ with the TLC model checker for formal specification, Rust with petgraph for graph representation, automated post-processing for graph cleanup, greedy coverage heuristics with SCC decomposition for test generation, and Graphviz for visualization. Each choice addresses the challenge it was selected for. The critical question is whether they compose into a single cohesive system where a user provides a TLA+ specification and receives an ordered set of test sequences without manual intervention between stages. This section describes how we intend to bring these pieces together.

At the highest level, MidGARD is a Rust command-line application organized as a five-stage pipeline. A TLA+ specification file enters the system, is verified and converted into a Graphviz DOT file, parsed into an in-memory directed graph, cleaned and analyzed to produce test paths, and finally ordered and written out as a structured test plan. Rust serves as the orchestration layer throughout. With the exception of the TLC model checker, which runs as an external Java subprocess, every stage executes natively within a single Rust binary, keeping the deployment footprint small and avoiding the complexity of coordinating multiple runtimes.

The first stage bridges the formal specification world and the graph analysis world. When a user provides a .tla file, MidGARD spawns TLC as a child process using Rust's std::process::Command, passing along any user-specified configuration, including VIEW and ALIAS options for state-space reduction. The pipeline captures TLC's output streams, inspects the exit code, and either proceeds with the generated .dot file or halts and surfaces the error. This is the only stage that depends on an external runtime (OpenJDK ≥ 11.0.6), and it represents the "model-initiated" nature of MidGARD: the TLA+ specification is what drives everything downstream.

Once TLC has produced a DOT file, the second stage parses it and constructs an in-memory directed graph. Using petgraph's dot_parser feature, the system populates a petgraph::StableGraph, mapping state predicates to nodes and actions to directed edges. Metadata from the DOT file (node labels, edge labels, and any TLC-attached attributes) is preserved as associated data, since it will be needed when presenting test sequences in human-readable form. The StableGraph type is used rather than petgraph's base Graph because the preprocessing stage that follows removes nodes and edges, and StableGraph keeps indices stable across these removals. Petgraph's adjacency-list representation gives us efficient traversal and low memory overhead, and its built-in algorithms, including Tarjan's SCC decomposition and various search strategies, save us from reimplementing well-known graph routines from scratch.

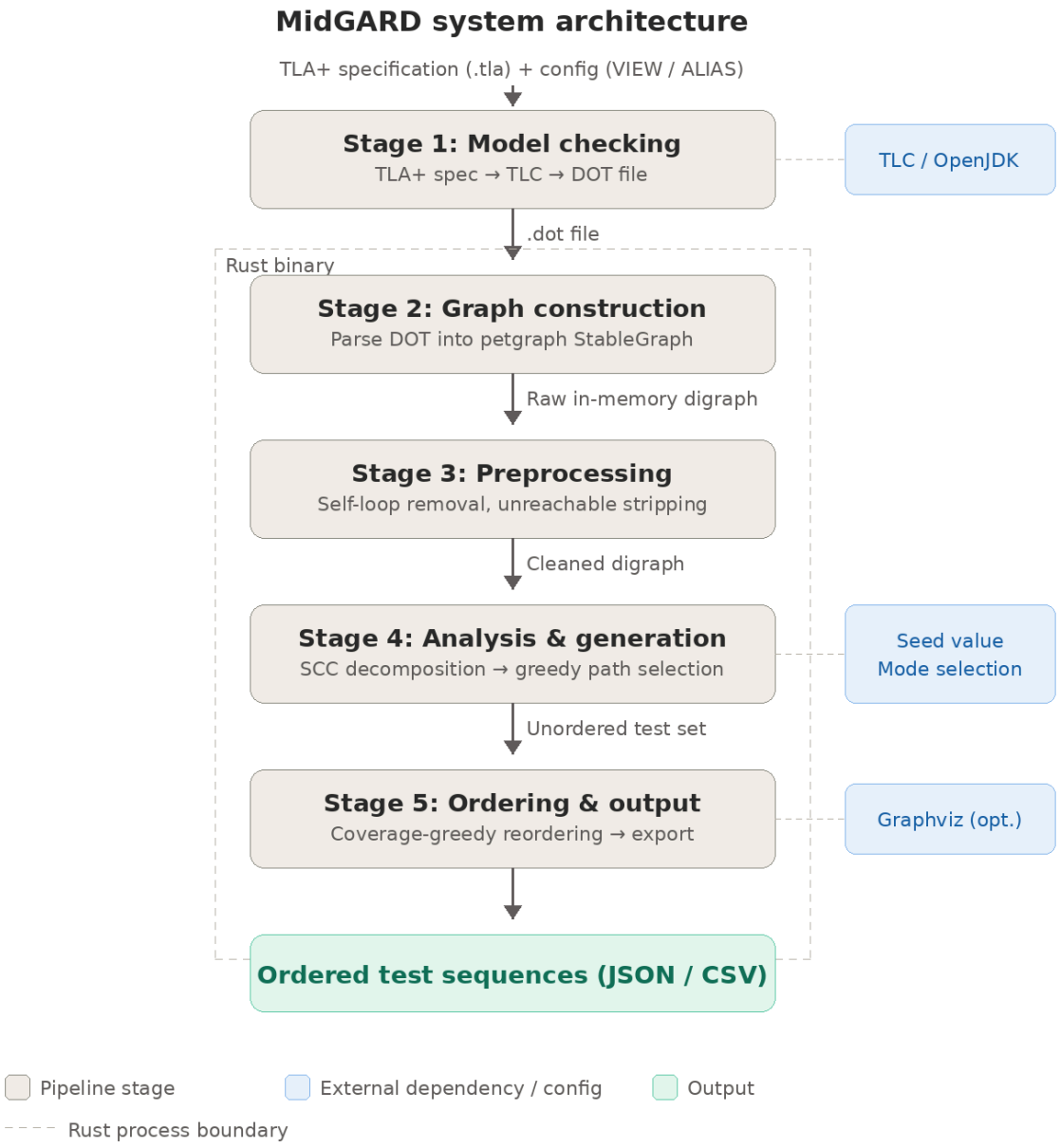
The third stage applies the preprocessing strategy from our redundant-transitions analysis. Operating directly on the in-memory graph, the system removes self-referencing edges (artifacts of TLA+ stuttering invariance), computes reachability from a configurable initial state, and strips unreachable nodes along with their incident edges. Additional cleanup such as collapsing equivalent states may be applied depending on the input specification. Each substep mutates the graph in place, while the original TLA+ specification and DOT file remain untouched on disk.

This separation is deliberate: the formal model retains its integrity as the mathematical source of truth, while the in-memory copy is shaped for practical analysis.

The fourth stage is where MidGARD's core value is realized. The cleaned graph first undergoes SCC decomposition via Tarjan's algorithm (available in petgraph) to identify loop structures and clusters of mutually reachable states. The engine then generates test paths according to the user's chosen mode: in full-coverage mode, each new path is selected to greedily maximize the number of uncovered edges it traverses; in n-test mode, each new path is selected to greedily maximize its minimum edge-set distinctness from all previously generated tests. A seed value controls every nondeterministic decision, making results fully reproducible. For graphs that remain large after preprocessing, bounded path lengths and greedy heuristics keep computation tractable; the system converges on a high-quality finite set rather than attempting exhaustive enumeration.

In the fifth stage, tests are reordered using a coverage-greedy strategy: each successive position is filled by the test covering the most edges not yet covered by earlier entries. The practical effect is that any prefix of the list still constitutes a strong test suite on its own, which matters when resource constraints dictate running only a subset. The ordered list is written out in a structured format to be determined with SanDisk (JSON and CSV are likely candidates), and Graphviz can optionally render a visualization of the cleaned graph with covered edges highlighted.

A system diagram illustrating this five-stage pipeline, the data artifacts flowing between stages, and the external dependencies (TLC/Java, Graphviz) is displayed on the following page. The diagram shows the key property of this architecture: each stage has a well-defined interface (a file on disk or an in-memory graph) and operates independently of the others' internals. This modularity is what makes integrating a diverse set of technologies feasible, and it ensures that individual components can be tested, replaced, or extended without rearchitecting the system as a whole.


MidGARD system architecture
TLA+ specification (.tla) + config (VIEW / ALIAS)
Stage 1: Model checking
TLA+ spec → TLC → DOT file
TLC / OpenJDK
.dot file
Rust binary
Stage 2: Graph construction
Parse DOT into petgraph StableGraph
Raw in-memory digraph
Stage 3: Preprocessing
Self-loop removal, unreachable stripping
Cleaned digraph
Stage 4: Analysis & generation
SCC decomposition → greedy path selection
Seed value
Mode selection
Unordered test set
Stage 5: Ordering & output
Coverage-greedy reordering → export
Graphviz (opt.)
Ordered test sequences (JSON / CSV)
Pipeline stage
External dependency / config
Output
Rust process boundary

5 Conclusion

MidGARD addresses a real and pressing challenge in SanDisk's validation process: the directed graphs generated from formally verified TLA+ specifications contain far too many possible execution paths for any engineer to evaluate manually, yet the quality of regression testing depends on selecting the right paths. Without a systematic, mathematically grounded approach, test planning relies on intuition and risks missing corner cases or cross-feature interactions buried in the state space.

In this document, we identified five core technological challenges facing the project: managing the large state spaces inherent in TLA+ specifications, representing and processing directed graphs efficiently in Rust, extracting meaningful test sequences from those graphs, filtering redundant transitions introduced by TLA+'s stuttering invariance, and integrating a diverse toolchain into a single automated pipeline. For each challenge, we evaluated multiple alternatives against criteria specific to MidGARD's requirements and selected the approach best supported by that analysis. TLC's VIEW and ALIAS abstractions give us a principled, user-controlled mechanism for state-space reduction. The petgraph library provides a mature, well-tested foundation for graph representation, DOT parsing, and algorithmic analysis. Greedy coverage heuristics with SCC decomposition offer a flexible generation strategy that supports both of MidGARD's operating modes. Automated post-processing keeps graph cleanup scalable and repeatable without modifying the original specification. And Rust ties the pipeline together as a single binary with only one external runtime dependency.

Taken together, these decisions define a realistic and well-supported path toward implementation. Every major component of the system has a viable, free, and open-source solution, and no challenge requires inventing fundamentally new technology. Our next steps are to validate these choices through targeted technology demonstrations: parsing a TLC-generated DOT file into petgraph, running SCC decomposition and greedy test generation on a sample graph, and confirming end-to-end data flow from a TLA+ specification to an ordered set of test sequences. These demonstrations will be completed before the end of the spring semester, providing a solid foundation for full implementation when the team reconvenes in the fall.