Team SSDynamics

Software Design Document

February 7, 2025

Team Members
Carter Kaess, Charles(Chas) Diaz, Connor Aiton, Charles Descamps

Mentors
Brian Donnelly, Savannah Chappus

Sponsors
Chris Ortiz, Senior Technologist
Western Digital Corp.

John Lee, Senior Director
Western Digital Corp.

Version 2.1

Table of Contents

Introduction

Solid-state drives (SSDs) are essential to modern storage, powering everything from personal devices to large-scale cloud data centers. Among them, NVMe (Non-Volatile Memory Express) drives offer high-performance storage using NAND flash memory, ensuring exceptional speed and efficiency. Their ability to handle demanding workloads makes them vital for applications requiring high throughput and responsiveness.

Ensuring NVMe drive reliability is crucial in an industry worth tens of billions of dollars. Western Digital, a leader in storage solutions, relies on rigorous validation processes to maintain competitive excellence. Traditionally, engineers manually define test sequences, but this approach has limitations and inefficiencies, which can lead to critical issues later in the product's life cycle. These limitations include:

- **Incomplete test coverage** – Manually defined test sequences may miss important edge cases, leading to undetected issues.
- **Unconscious bias in test design** – Engineers might inadvertently focus on expected scenarios while overlooking rare but critical failure cases.
- **Difficulty identifying edge cases** – Manually creating test sequences makes it challenging to capture unexpected or extreme conditions.
- **Inefficiency and high effort** – Engineers spend significant time manually designing and maintaining test sequences instead of focusing on higher-level validation tasks.

Solution Vision

To address these challenges, Western Digital's senior SSD validation technologist, Chris Ortiz, has tasked our team with developing a proof-of-concept solution that uses random model simulation to automate test generation. By eliminating the need for manual test definitions, our approach expands coverage, accelerates validation, and improves overall SSD reliability.

Our solution must meet specific requirements, which we have divided into two categories: software and hardware considerations.

Software Requirements

- Automate NVMe test sequence generation dynamically based on real-world usage patterns.
- Ensure randomness while maintaining reproducibility for debugging purposes.
- Measure improvements in test coverage and validation efficiency.

- Integrate seamlessly with Western Digital’s existing validation framework and SSD testing tools.

Hardware Requirements

- Comply with Western Digital’s hardware validation infrastructure and test environments.
- Remain adaptable to future validation frameworks with minimal modifications.
- Ensure resource efficiency to avoid introducing significant computational overhead.

By addressing inefficiencies in SSD validation through automated test generation, our solution aims to make the process more robust, scalable, and effective while reducing reliance on manual test definitions.

Implementation Overview

Our solution aims to significantly enhance NVMe validation by automating test case generation using a random model simulation framework. The key challenge we are addressing is the manual nature of current validation workflows, particularly the creation of test cases, which often leads to limited coverage and missed edge cases. The proposed system will streamline the testing process by automatically generating diverse test sequences, ultimately improving reliability, adaptability, and providing valuable data-driven insights. This will allow Western Digital to optimize its NVMe testing and better meet the demands of the industry.

General Approach

The solution centers on a **random model simulation** that autonomously generates diverse test case sequences. These test cases are based on an **NVMe specification file** written in **TLA+**, a formal specification language. This file provides a high-level design of the system, including component interactions, states, and state transitions. The simulation explores new state spaces which represent every possible state of testing sequences, which increases test case and functionality coverage while uncovering edge cases that traditional methods might miss. The generated test sequences are executed on physical NVMe drives, with results logged for later analysis.

Technologies and Frameworks

1. **TLA+**: The system specification is defined using TLA+, which abstracts the design and system behaviors. This abstraction allows engineers to focus on

higher-level design rather than implementation details, ensuring alignment with industry standards.

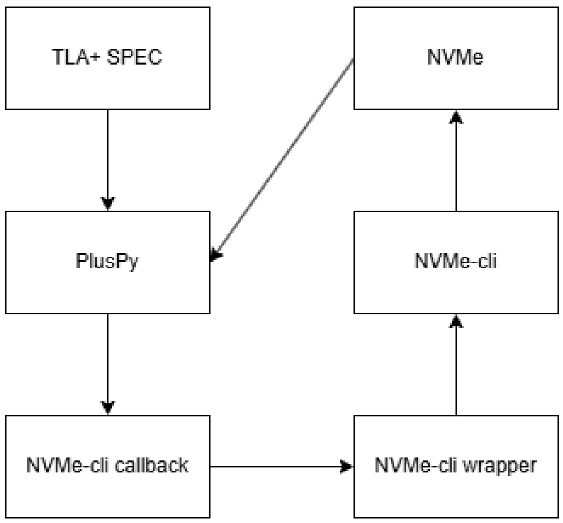
2. **Random Simulation Algorithm**: A seeded random algorithm autonomously generates test cases, ensuring broad coverage of test cases and the discovery of new edge cases. This randomness reduces human bias and helps explore scenarios that might not be immediately apparent to an engineer or developer.
3. **NVMe Command Interface**: This interface allows the model simulation to interact directly with the NVMe drives. It automates the execution of test sequences and ensures consistent, repeatable results.
4. **Logging System**: Test results are captured by a modular logging system, which can be customized to meet Western Digital's needs. This system ensures traceability and enables engineers to focus on relevant data.
5. **Seed Resampling**: To ensure repeatability, the system includes seed resampling. This allows errors to be reproducible, which is critical for debugging and fixing issues in the testing process.
6. **Python:** An easy to maintain language that will act as our main driver connecting everything.

Architectural Overview

To effectively carry out our project functions, we will need to build a clear and well modularized system.The idea for our solution is to use an API-like terminal program called NVMe-CLI which can communicate and call commands to the hardware of an SSD, then connect this to a Python script that will call those commands. The commands that the Python script calls will be determined by two things:

1. A TLA+ specification file
2. A randomly generated number or seed.

Figure 1



TLA+ is a high level design language which is focused on the design above the code level. In our case we could write a specification file that defines all tests the NVMe-CLI will execute. The seed will randomly decide which of the tests written in the TLA+ specification are used during that run.

This means that our solution can load a TLA+ specification script, then call the tests randomly to the NVME-CLI through a Python PlusPy script. These tests can be done repeatedly and can also be used during down time between other tests. This will increase the likelihood of the testing team to find errors in their SSDs.

How It Works

Figure 2

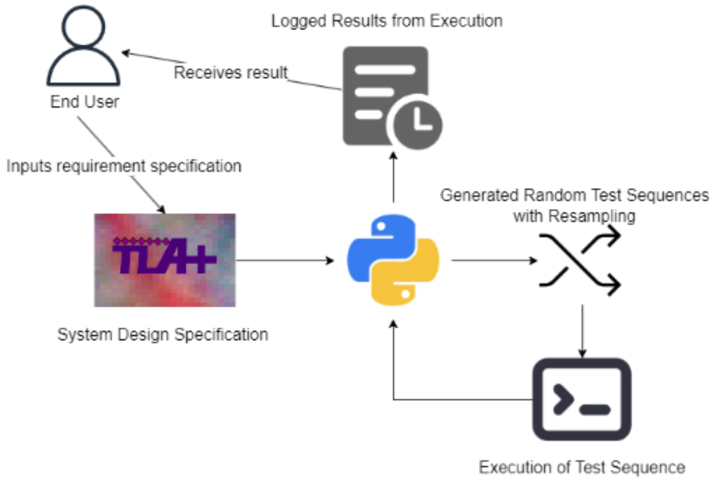

Logged Results from Execution
Receives result
End User
Inputs requirement specification
TLA+
Generated Random Test Sequences with Resampling
System Design Specification
Execution of Test Sequence

Diagram displaying system architecture

- System Design Specification: This specification file will include industry standards for driving test sequences, as well as new requirements from our client.The system begins by taking an NVMe specification file, written in TLA+, which defines how the system should work at a *design* level, defining how components interact and possible state and state transitions on a very high level.
- Generated Random Test Sequences: A core feature of our software is a seeded random simulation algorithm that autonomously creates numerous test case sequences that aim to explore previously unexplored test cases.
- Execution of Test Sequence: The generated test sequence will be executed automatically on physical NVMe drives. The program will then record the results for later analysis. Seed sampling will then be implemented to ensure repeatable results. If the algorithm detects an error, we resample the seed until it ends.
- Logged Results from Execution: Results are captured via a modular logging component that captures data on each outcome, ensuring traceability. This logger will be customizable to the client's needs, including changing the levels of failure reported or sequences recorded. Engineers will use these reports to evaluate product readiness.

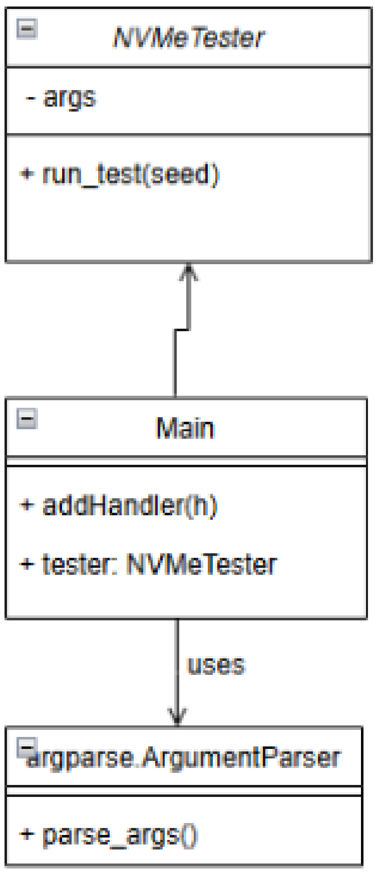
Module and Interface Descriptions

Now that we've gone over the basics we will dive deeper into the project details, including files and interactions between them which will describe the implementation of our project.

interface.py

`interface.py` helps to automate testing by allowing the user to provide arguments for things such as the number of attempts to make, retry limits, starting state, next state, and other settings through the command line. These parameters are used to guide the testing while staying within the TLA+ specification. Testing can be made more thorough by increasing the number of attempts to make, changing the starting and next states, and decreasing how many errors need to be found before testing is halted. This flexibility ensures that testing is as robust as needed and can be stopped when errors are found.

Figure 3



interface.py
classes

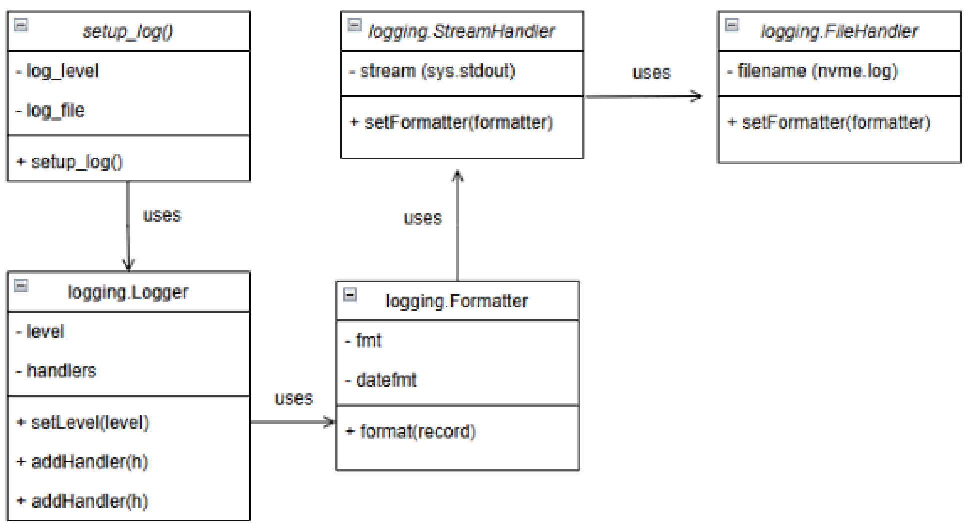
tester.py

The `tester.py` file serves as the main calling function and uses `interface.py` to set up the testing parameters such as logging detail and commands to be run. `interface.py` is accessed directly by `tester.py` which takes in the testing parameters and creates a testing object. This object serves as a basic data structure and persists throughout the test to ensure the parameters are used correctly.

logger_config.py

The `logger_config.py` file sets parameters for logging the output from the console, including formatting, the output file, and level of logging detail. The output file is specified by the tester at the same time as the testing parameters are entered. The `logger_config.py` file also prints out a success message to ensure the logging was successfully outputted to a file. Depending on the logging level, the file will also include a timestamp, the importance of the logging line, and the log message. This file ensures that the output of the program is saved to a file for later review.

Figure 4



logger_config.py
classes

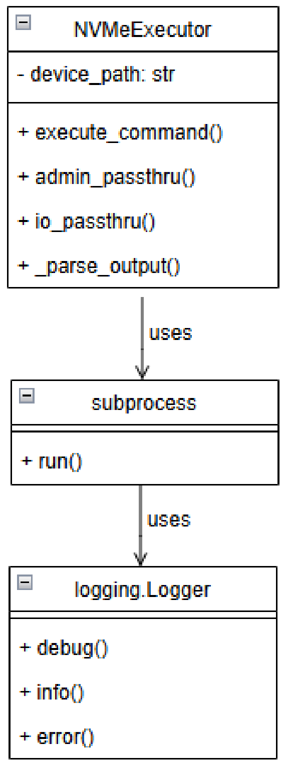
This file focuses on logging the output from the NVMe CLI. In order to maintain modularity, we are using several methods as follows: The function **`setup_log()`** allows the user to input testing parameters which specify the logging detail and output file name. The logger then takes these parameters and uses them to initialize the logging handler. The formatter then takes the information and adds metadata such as the timestamp of the output. Stream handler takes this formatted output and outputs it to standard out for the user to see in real time. Finally the file handler outputs this to a file for later review.

nvme.py

`nvme.py` provides an interface for executing NVMe CLI commands. The main use is to run commands automatically as the states inside the TLA+ file are explored. The file also implements the logging functionality created by other files ensuring the output is properly saved to a file. This file also handles error messages as they happen and sends them to be logged.

The nvme.py file focuses on executing the NVMe CLI commands using a modular design that uses different functions for each type of NVMe CLI command as they have different input parameters.

Figure 5



nvme.py classes

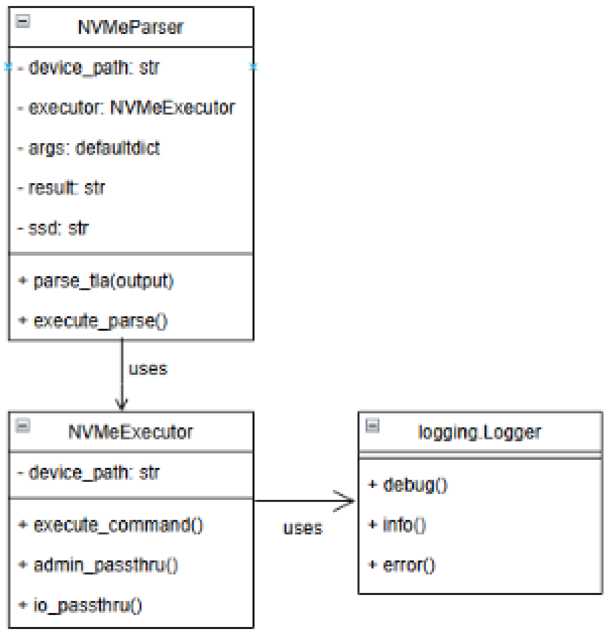
The **execute_command()** function uses the device path along with supporting functions to execute the command on the correct device. The type of NVMe CLI command determines which execution command is called: either **admin_passthru()** or **io_passthru()**. These functions take the input information from **execute_command()** and format it into a proper NVMe CLI command before the **run()** function is called which executes the command. The output is then passed to the logging file where its logic is handled separately.

parser.py

parser.py attempts to directly translate the states provided by the TLA+ specification into NVMe CLI commands before using nvme.py to execute them. It specifies which NVMe device the commands should be executed on before attempting to translate them. If a direct translation is not possible, nvme.py is sent the incomplete data and then returns an error message.

parser.py focuses on translating the transition between TLA+ states into an NVMe CLI command. This is done by taking the information from **interface.py** about the device and testing parameters. The information is then converted into a NVMe CLI command before calling the executor functionality.

Figure 6



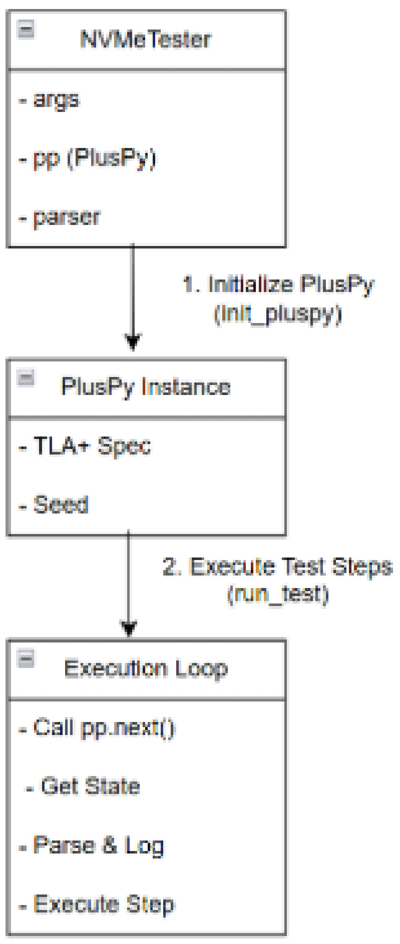
parser.py classes

Pluspy

The pluspy library is specifically designed to help interpret TLA+ models, check models, visualize, and much more, specifically in python. However we will just be using it to help interpret TLA+ models as it is assumed that the TLA+ specification will already be followed when using this program. We will modify Pluspy to help suit our needs better, only taking advantage of the functionality we need and allow for Western Digital specific states to be interpreted.

Pluspy is a library used internally and has no public facing functionality. We are using it to help translate TLA+ states and transitions into NVMe CLI commands. It's– specifically called by `nvme_tester` with the TLA+ information before passing its output to `tester.py` to execute the test.

Figure 7



tester.py classes

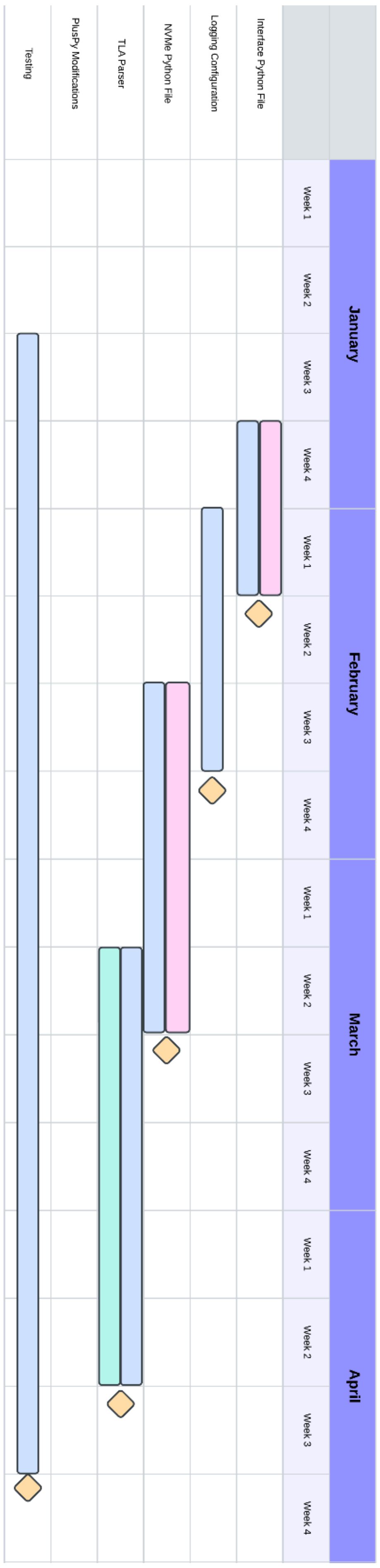
Implementation Plan

When planning out each section, we noted we took about 2-3 weeks typically to complete it, and make sure that each part interacts with the system overall. This is very dynamic, and may be resized to make sure that whichever sections are giving us trouble can be moved around or retimed to allow for the other sections to be completed properly. The testing part of the timeline is meant to endure through the entire project, as we need to make sure that the project is working as intended before handed over to the client.

When working on the project, we intend to work on it together, but also have pairs who specialize in certain areas. That's what each color bar is for, the pink for NVMe-CLI, blue for PlusPy/Python, and green for TLA+. It focuses on the kind of code we are going to write. The PlusPy modification bar is purely speculation in case the PlusPy library we are using needs to be modified to support a function that we write.

We like to do pair programming, when possible, as it helps keep each other accountable, and keeps everyone in the loop about what each function or code is doing. This does try to align with our focuses.

Generative SSD Testing Timeline - Figure 8



Each milestone is there to outline the major goals, and when we expect to have them completed by, to be able to show them off to the client during that week's meeting. We would ideally have the project complete by the end of April.

The development timeline is meant to mirror the development of each Python module for the project, with each row being a module of our code, and how long we expect it to take. Flexibility is key here though, as we may need to implement more modules to either make the project more modular, or to implement some functionality we need to add on.

Conclusion

Solid state drives are everywhere due to their small size and incredible performance compared to larger hard disks. Because of their speed, these devices need to be tested thoroughly to ensure functionality and reliability. This new technology and extra speed leads to more errors, requiring more through testing. We hope that our solution will assist human-made test cases that may not thoroughly test these devices. Our system implements several technologies to achieve this such as, TLA+, Pluspy, and NVMe CLI commands, with several python files which help translate TLA+ into NVMe CLI commands, execute those commands, and save the output to a file for review. In addition we are implementing a seeded test case generator allowing for the same test to be run multiple times to ensure stability. The modularity of these files allows for easier maintenance and compatibility. Our program will serve as a proof of concept for a more automated testing system, allowing engineers to focus on resolving bugs instead of trying to find them. Given the need for more through testing our program will assist engineers in ensuring that their devices are stable and ready for the consumer market.