System Verification and Validation Plan for TPG

Team 3, Tangle
Calvyn Siong
Cyruss Allen Amante
Edward Gao
Richard Li
Mark Angelo Cruz

April 2, 2025

Revision History

Date	Version	Notes
11/04/2024	0.0	Initial draft of VnV plan.
03/11/2025	0.1	Modified VnV Plan to include unit testing plan and changed summary and objective section based on TA feedback.
03/17/2025	0.2	Updated challenge level section to be advanced.
03/29/2025	0.3	Updated FR-SLN1 to make it more clear.
03/30/2025	0.4	Made requirement 15.3.1 less vague.
03/31/2025	0.5	Updated Visualization Module secrets.
04/02/2025	1	Revision 1

Contents

List of Tables

1 Symbols, Abbreviations, and Acronyms

symbol	description
TPG	Tangled Program Graphs
DNNs	Deep Neural Networks
RL	Reinforcement Learning
SRS	Software Requirement Specification
SLN	Solution
FR	Functional Requirement
NFR	Non-Functional Requirement
MIS	Module Interface Specification

Table 1: Symbols, Abbreviations, and Acronyms

This document provides the Software Validation and Verification plan of the TPG Capstone Project. It consists of general information such as objectives and relevant documentation, plans for the system's verification and validation, including test descriptions for both system and unit tests. This information is subject to change at any point throughout the project and will be updated accordingly with changes mentioned in the revision history table. Any complex and complicated aspects that may be required to complete an objective will be assisted by utilizing open-source external libraries.

2 General Information

2.1 Summary

The software being tested is the overall workflow of the open-source repository including the interface between the TPG framework and the physics engine environment, MuJoCo. The software to be implemented within this repository is an interface that allows for experiments to be run with MuJoCo, a physics-engine simulator. This is in addition to creating a software development pipeline within the repository that allows for an easier development experience as changes are implemented.

The Verification and Validation (VnV) Plan developed by the Tangle team outlines a systematic approach to ensure that the TPG framework with MuJoCo integration meets all specified requirements and functions correctly. The plan details the verification and validation activities, including test methodologies, responsibilities, and success criteria that will be crucial for the project's success. It provides a guideline for evaluating the system's compliance with requirements, design specifications, and stakeholder expectations.

2.2 Objectives

Tangle should achieve the following objectives:

- Tangle allows TPG to perform reinforcement learning in MuJoCo environments (FR-1).
- Tangle should adhere to standardized software engineering standard and practices in the Software Engineering Body of Knowledge (SWEBOK). This includes requirements for coding style, unit tests, and CI/CD.
- Tangle should be easy to install and run on major operating systems (MacOS, Linux, and Windows).
- Define a unit testing strategy that verifies critical functional requirements as well as modules that it applies to

Out of scope for this VnV Plan:

- Exhaustive testing of previously existing TPG framework code
- Security testing beyond basic access control verification
- Usability testing with external users beyond the development team and Dr. Kelly's research group

2.3 Challenge Level and Extras

The challenge level of the project is **advanced** as agreed upon by the course instructor since this project is an extension of the current Tangled Program Graphs repository created by Dr. Stephen Kelly. The extra that will be tackled by this project is a Research Report. This extra will cover the research and results discovered between incoporating dynamic memory to enhance reinforcement learning within MuJoCo using Tangled Progam Graphs.

2.4 Relevant Documentation

The following documentation is considered to be relevant and may provide more context about the project outside of this document:

- **Problem Statement and Goals** (Tangle, 2024c)**:** This document specifies a more detailed outline of the project's goals and purpose. This involves important stakeholder information, inputs and outputs of the project, including information in regards to the project's environment.

- **Development Plan:** (Tangle, 2024a)**:** This document provides a guideline regarding all of the tools and technologies that will be utilized throughout this plan for verification and validation.
- **Hazard Analysis:** (Tangle, 2024b)**:** This document describes detailed hazards that may occur throughout the development of the project. It specifies the system boundaries and components, including mitigation strategies in the form of safety and security requirements that may need verification and validation.
- **Software Requirements Specification** (Tangle, 2024d)**:** This document specifies all the non-functional and functional requirements that the project should satisfy by the end of the capstone period. This is useful for creating test cases that will verify and validate that the requirements have been met.
- **Module Interface Specification:** This document describes how different components within the system will interact with each other. This is beneficial to determine the modules that will be present within the system, and help determine the scope of testing.

3 Plan

3.1 Verification and Validation Team

Name and Roles	Responsibilites

Calvyn Siong: CI/CD verification	Responsible for ensuring the continuous integration and continuous deployment (CI/CD) pipeline functions as intended, supporting automated testing, smooth deployments, and efficient integration of code changes. Verifies that the CI/CD pipeline adheres to project standards, minimizes deployment risks, and enables consistent checks for potential errors in the TPG framework.
Mark Cruz: Code verification	Ensures that the codebase adheres to established software standards and project guidelines, including readability, maintainability, and overall code quality. Additionally, responsible for reviewing unit tests created to improve code reliability.
Richard Li: Performance verification	Ensures that the framework's performance metrics, including response times, computational efficiency, and resource usage, meet the standards necessary for the TPG framework.
Cyruss Allen Amante: SRS verification	Validates that the TPG framework and its integrations fulfill both functional and non-functional requirements as outlined in the Software Requirements Specification (SRS). Ensures that each requirement is accurately implemented.

Edward Gao: Data verification	Responsible for validating and verifying all data utilized by the TPG framework, including inputs and outputs. This includes ensuring data integrity and accuracy for agent-environment interactions and simulator-generated data.

Table 2: Verification and Validation Team

3.2 SRS Verification Plan

To verify the SRS, routine inspections will be conducted to ensure that the TPG framework meets both functional and non-functional requirements as outlined in the SRS. These inspections will involve regular reviews by the development team and collaborators such as Dr. Stephen Kelly's team with the TPG framework, who can provide ongoing insights and feedback as the project progresses.

Through these routine reviews, task based inspection will be leveraged to audit the most up to date progress made towards the project. This would involve walking through key tasks or scenarios (i.e.: new MuJoCo environments, CI/CD integrations) that the system must support, and inspecting how each task complies to the SRS's standards. Each review will employ a structured, scenario-based checklist that focuses on the specific tasks that the TPG framework needs to support, such as configuring simulations, setting up interactions with MuJoCo, and evaluating reinforcement learning performance metrics. By inspecting the SRS against these real-world tasks, the verification team will ensure that each requirement is clearly defined and actionable, supporting the development team in implementing functionality that aligns with user and system needs. In addition, the tasks/features implemented will be cross checked with the list of issues during the review section to display how each task satisfies the issue to ensure traceability.

3.3 Design Verification Plan

The plan is to verify that the design of the modifications made to the TPG framework meets the intended specifications and supports function-

ality aligned with project objectives set by Dr. Stephen Kelly. Thus as the primary stakeholders, Dr.Kelly's team will actively participate in hands-on reviews to verify the framework's design.

This verification will include design review meetings and usability assessments. Design review meetings will be held monthly during regular project meetings, providing an opportunity for stakeholders to give informal feedback on the design. These reviews will address specific design elements, such as compatibility with the MuJoCo integration, degree of fitness changes to the new integrations, and robustness of the results.

In addition to the regular reviews, usability assessments will be conducted with additional members from Dr. Kelly's team, focusing on the testing infrastructure being added to the TPG framework. Feedback gathered from these sessions will guide iterative design improvements, ensuring the framework's usability.

3.4 Verification and Validation Plan Verification Plan

To verify the Verification and Validation Plan itself, peer review will be necessary to ensure that the listed test plans are properly conducted. The team will also conduct a review session with Dr Kelly when the project is completed. Dr. Kelly will review the Verification and Validation (V&V) document with the team to ensure that all tests outlined in the V&V plan were executed successfully. This collaborative review will confirm that the system meets the required specifications and that all verification and validation criteria have been addressed satisfactorily.

3.5 Implementation Verification Plan

The implementation verification process will focus on ensuring that our modifications to TPG and the MuJoCo integration meet the project's core requirements. This will be accomplished through several complementary approaches.

Code review will serve as the primary verification mechanism. All pull requests will require review by at least one other team member before merging. During these reviews, we will verify adherence to the Google C++ Style Guide, completeness of documentation, potential impacts on existing TPG

functionality, and the correctness of MuJoCo integration points.

Version control validation will be maintained through strict repository management practices. Branch protection rules will prevent direct pushes to the main branch, ensuring all changes go through the required review process. We will maintain regular synchronization between the GitHub and GitLab repositories to ensure consistency across development platforms.

Manual testing and inspection will verify that core TPG functionality remains intact. This includes verifying program generation and mutation, team/graph evolution, and action selection mechanisms. For the MuJoCo integration specifically, we will verify basic environment setup, proper handling of state and action spaces, and compare performance against the existing implementations such as CartPole as a baseline.

3.6 Automated Testing and Verification Tools

Our primary automated verification will be implemented through a GitHub Actions CI/CD pipeline. This pipeline will perform automated build verification across multiple platforms including Linux, Mac, and Windows to ensure cross-platform compatibility. It will also run basic integration tests and enforce code style requirements.

For manual verification, we will leverage TPG's existing suite of plotting and statistics tools to analyze performance and behavior. The OpenGL visualization capabilities built into TPG will allow us to inspect agent behaviors directly. Similarly, MuJoCo's built-in visualization tools will provide another avenue for verifying correct environment integration and agent behavior.

3.7 Software Validation Plan

The software validation plan begins with establishing CartPole as a baseline for comparison. We will compare TPG's performance between the existing environment implementation and the new MuJoCo implementation. This comparison will include validating expected agent behaviors through visual inspection using both OpenGL and MuJoCo visualizations, including verifying that fitness scores achieve similar thresholds in both implementations.

Core integration validation will focus on verifying TPG's ability to properly interact with MuJoCo environments. We will verify that TPG can successfully initialize MuJoCo environments, receive valid state information, send valid actions, and receive appropriate rewards. Throughout the training process, we will monitor the environment's response to agent actions and verify training progression through both fitness scores and visual behavior inspection.

Documentation validation will ensure that all necessary information is available to users and future developers. This includes verifying the completeness and accuracy of installation guides, with particular attention to environment setup steps and dependency management. Usage documentation will cover running experiments, visualizing results, and common troubleshooting procedures.

Research validation will be conducted through regular review meetings with Dr. Kelly to ensure the implementation supports the project's research goals. These reviews will validate that basic research capabilities are preserved and verify that new MuJoCo environments can be added as needed. We will assess TPG's behavior in baseline environments to ensure it provides a solid foundation for future research experiments.

The entire validation process will be iterative, with feedback incorporated throughout development. Our focus is on ensuring the basic integration is sound and providing a foundation that will support future research experiments. Rather than attempting to validate against specific performance criteria, we will validate that the implementation provides the necessary capabilities for exploring research questions about TPG's behavior in MuJoCo environments.

4 System Tests

4.1 Tests for Functional Requirements

The following section covers the test cases for functional requirements discussed in the SRS document. Every test is defined with control, type, initial state, input and output values, test case derivation and how the testing will

be performed. Using this approach of testing will maintain the high-quality experience that users may look for when interacting with a software.

Additionally, in order to perform these tests, follow this setup guide to get set up so testers can reproduce.

4.1.1 Interface that is Compatible with MuJoCo

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- FR-1
- FR-8

1. FR-SLN1

   Control: Manual

   Initial State: User has the repository open on their code editor with a terminal open to the experiment directory

   Input: This tpg-run-mpi.sh -n 4 script executes a given experiment

   Output: The output of the script is **Starting run ...** printed on the terminal. Note the experiment is running in the background.

   Test Case Derivation: The script successfully executes the experiment and the output is printed. A user could also run htop to see the processes of the experiment running to confirm. Also another script tpg-run-mpi.sh -m 1 can be run to output an OpenGL visualization of the experiment.

   How test will be performed: The test can be performed across different Operating Systems. The interaction with the system is through the command line.

2. FR-SLN2

   Control: Manual

   Initial State: User wants to run the experiment using the Digital Research Alliance of Canada computational clusters. They have the experiment directory of the experiment they want to run open on the terminal.

Input: The TPG framework code base and a specific shell script tpg-run-slurm.sh

Output: The experiment can be trained and run for long periods of time in the cloud environment. The .std log files can be downloaded from the cloud environment which provides details of the training process in the cloud.

Test Case Derivation: After the experiment is run in the cloud environment, the user can download the .std logs to analyze what happened during the training process.

How test will be performed: The test can be performed across different operating systems (Linux, MacOS, Windows) depending on what OS the user is using. The interaction with this test is through the command line.

4.1.2 Experiment Visualization

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- FR-2

1. FR-SLN3

   Control: Manual

   Initial State: Experiment has already started running. **Starting run ...** is printed on the terminal.

   Input: This tpg-run-mpi.sh -m 1 script visualizes the current experiment

   Output: OpenGL animation of the current experiment is displayed on the users screen.

   Test Case Derivation: Upon the script the above being run, the current experiment needs to be running in the background and can be validated running htop to see if those processes are running. When the script is run, the user will be able to see the animation appear on their display. The pop up is very visible and the user can see the experiment running in real time.

How test will be performed: The test can be performed across different operating systems. The interaction with the system is through the command line.

4.1.3 Github Actions CI/CD Pipeline

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- FR-3
- FR-4
- FR-5
- FR-7

1. FR-SLN4 Control: Automated + Manual

   Initial State: New PR has been made and the changes have been approved and ready to merge back into main (Automated). Or the user wants to manually trigger the pipeline to merge code changes into the main branch (manual).

   Input: New changes to the code base have been made and a PR has been made to merge the changes back into the main branch.

   Output: New code changes are merged to the main branch, if the validation fails then the user is notified that the pipeline has failed.

   Test Case Derivation: When new code changes are being merged to the main branch, the Github Actions CI/CD pipeline is triggered to run to build, test, and deploy the new code. The pipeline is designed to automatically perform all those tasks.

   How test will be performed: Whenever new code changes are made, pipeline will be triggered. The tests will run on any web browser that supports GitHub.

4.1.4 Software Engineering Practices

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- FR-6

1. FR-SLN5

   Control: Automated

   Initial State: A new code change has been pushed and merged into the main branch.

   Input: The latest commit in the main branch triggers the GitHub Actions pipeline.

   Output: The CMake-based test suite is executed, and all test cases are run automatically. The pipeline reports success if all tests pass or failure if any tests fail.

   Test Case Derivation: The system must detect changes in the main branch and automatically trigger the test suite to ensure no regressions or breakages are introduced. This verifies that new changes maintain compatibility with existing functionality.

   How test will be performed:

   - The GitHub Actions pipeline is configured to monitor the main branch for new commits.
   - Upon detecting a new commit, the pipeline checks out the code, builds the project using CMake, and runs all C++ test cases.
   - Test results are reported in the GitHub Actions workflow summary.
   - If any tests fail, the pipeline will mark the run as failed, notifying the team of the failure.

2. FR-SLN6 Control: Automated

   Initial State: User has just made code changes and pushed to a feature branch.

   Input: Code that has been modified by the user.

   Output: Within our CI/CD pipeline, a "pre-commit" pipeline is ran and if successful it lints the files that were modified.

   Test Case Derivation: The user is able to see formatting changes proposed by clang-tidy and clang-format (two C++ linters that abide by

the Google C++ style guide). The user can then choose to accept or reject the changes.

How test will be performed: Whenever new code is committed after a user makes modifications to a codebase, the files which they edited will be linted to ensure proper formatting and the user will have the opportunity to manually validate.

4.2 Tests for Nonfunctional Requirements

The following section covers the test cases for non-functional requirements discussed in the SRS document. This focuses on the usability, performance, maintainability, operation and security of the system. Every test is defined with control, type, initial state, input and output values, test case derivation and how the testing will be performed. For control, various types of testing will be used, ranging from automated unit and integration testing to individuals manually interacting with the TPG framework to validate the system's behavior. Using this approach of testing will maintain the high-quality experience that users may look for when interacting with a software.

4.2.1 Clear and Comprehensive Documentation

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 11.1.1, 11.3.1, 11.3.2, 11.4.1, 11.4.2, 11.5.1, 13.4, 14.1.2, 14.2.1

1. NFR-SLN1

   Control: Manual

   Type: Usability, Operational and Environmental, Maintainability and Support

   Initial State: System has documentation that covers all key sections is available on the repository, and is accessible to users

   Input/Condition: Users open the documentation to get started using the framework

   Output/Result: Users are able to locate and follow along the instructions without any confusion or having to refer an outside source (i.e. StackOverflow, Github Docs)

Test Case Derivation: Concise and comprehensive documentation of the framework including helpful concepts, and step-by-step guide is provided within the README files inside the repository

How test will be performed:

- Users will be granted access to perform tasks execution within the framework
- Users will be instructed to follow along the published documentation, and locate specific information
- At the end, a survey is available for users to provide a feedback on the documentation from a scale of 1 to 10

4.2.2 Real-time, Accurate Message Logging

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 11.1.2, 15.4.1

1. NFR-SLN2

   Control: Automated + Manual

   Type: Usability, Security

   Initial State: System is in idle, and ready to log messages for tasks

   Input/Condition: User performs an action or an error during the action occurs

   Output/Result: Logging of messages in real-time with low latency

   Test Case Derivation: Each method, and command available to the user includes some sort of logging to provide users with step-by-step insights

   How test will be performed:

   - Manual
     - Monitor message log in real-time, verify timestamp of each messages is accurate and confirm the action occurring at that time is aligned with the logs. Often, users will receive notification of a survey to provide a feedback on the system's logging from a scale of 1 to 10

- Automated
  - Create unit tests that checks message logs from certain functions and validate that the messages are sent with no later than 1 second delay through the timestamps

4.2.3 Customizable MuJoCo Simulation Parameters

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 11.2.1

1. NFR-SLN3

   Control: Manual

   Type: Usability

   Initial State: MuJoCo is integrated within TPG.

   Input/Condition: User modifies specified fields inside parameters.txt or overwrites default value of fields using command line flags.

   Output/Result: Simulations adhere to the desired parameters.

   Test Case Derivation: MuJoCo has default parameters that can be customized by the users if necessary, to adjust the behavior of every environment based on their specific needs.

   How test will be performed:

   - Modify MuJoCo parameters inside parameters.txt, run the simulations and verify that the behavior of the newly adjusted simulations complies with expected outcome from calculations
   - Within the parameters.txt, a survey is available for users to provide feedback regarding the experience of using custom parameters from a scale of 1 to 10

4.2.4 High Numerical Precision in Computation

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 12.2

1. NFR-SLN4

   Control: Automated + Manual

   Type: Performance

   Initial State: System is initialized with calculations in mind for computation.

   Input/Condition: Users perform calculations that require high numerical precision such as floating point with low computation errors.

   Output/Result: Calculation results are within acceptable tolerance limits such as computational errors below 0.0001.

   Test Case Derivation: Due to finite number of available bits for storing numbers, computers are bound to have an error on numerical computation if numbers are too small or too big.

   How test will be performed:

   - Manual:
     - Perform tasks that are precision critical and inspect their results by comparing to expected numerical values and verify that the computational error is below 0.0001
   - Automated:
     - Create and run automated testing for functions comprising of high precision results and assert that the computational error is below 0.0001 by comparing expected output to real output

4.2.5 Handling Invalid Inputs and Unexpected Conditions

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 12.3, 12.6, 15.5.2

1. NFR-SLN5

   Control: Automated

   Type: Performance, Security

   Initial State: System is idle and ready to process normal and unusual inputs.

Input/Condition: Unusual inputs to simulate model failures.

Output/Result: System shows meaningful error logs, and handles exceptions without terminating.

Test Case Derivation: System will use try and catch blocks where possible to handle errors gracefully.

How test will be performed:

- Create and run automated unit and integration tests to check various types of invalid inputs such as infinity, and NaN
- Verify that the system recovers gracefully and outputs a descriptive error message

4.2.6 Cross Platform Compatibility

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 13.1, 14.2.2, 14.3.2

1. NFR-SLN6

   Control: Manual

   Type: Operational and Environmental, Maintainability and Support

   Initial State: System is available to be downloaded across different OS environments.

   Input/Condition: Users perform core functionalities of the framework.

   Output/Result: System provides outputs that are similar across different OS environments.

   Test Case Derivation: TPG and the MuJoCo environment integration are used across different OS platforms (Windows, Linux, MacOS).

   How test will be performed:

   - Users will run the framework across different OS environments
   - Every user is tasked to run various core functionalities and verify that behaviors are similar across OS

4.2.7 Secured and Robust Repository Management

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 14.2.1, 15.1.1, 15.1.2, 15.2.1, 15.5.1

1. NFR-SLN7

   Control: Manual

   Type: Maintainability and Support, Security

   Initial State: System is available to be modified on a version control tool such as GitHub.

   Input/Condition: Repository is accessed by authorized and unauthorized users through various Git actions.

   Output/Result: Access attempts are blocked for unauthorized users, and granted for authorized users.

   Test Case Derivation: Repository must be protected to avoid unauthorized and defective code from getting integrated within the system.

   How test will be performed:

   - Simulate unauthorized and authorized access attempts on the repository
   - Validate that security protocols such as role-based access control, two-factor authentication, and protected branch are functional

4.2.8 Data Privacy through Obfuscation

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 15.3.1

1. NFR-SLN8

   Control: Automated + Manual

   Type: Security

   Initial State: System has the capability of storing and visualizing structured (CSV, JSON) and unstructured (log files, text files) data.

Input/Condition: Sensitive data is pushed into the public repository through Git.

Output/Result: Obfuscation of sensitive data is implemented, protecting privacy of any entity. This data is obfuscated using SHA-256 hashing, implemented within the framework, ensuring privacy compliance.

Test Case Derivation: Data, mostly stored in CSV or .txt files, are publicly stored across the framework.

How test will be performed:

- Automated: Implement unit tests to scan logs, stored files, and system messages for exposed sensitive patterns (e.g., regex detection of emails, credit card numbers). Validate that all detected sensitive data is transformed into its obfuscated equivalent.
- Manual: Manually check stored data through each file within the framework. Compare obfuscated outputs with expected masked formats to confirm compliance.

4.2.9 Compliance with C++ Coding Best Practices

The tests below provide the means to evaluate the following functional requirements referred to in the SRS document:

- 13.3, 17.1, 17.2

1. NFR-SLN9

   Control: Automated

   Type: Operational and Environmental, Compliance

   Initial State: Codebase implements an established Google C++ Style Guide with proper support and documentation.

   Input/Condition: Codebase is analyzed for its compliance with the style guide.

   Output/Result: Codebase follows the style guide, improving maintainability and readability.

   Test Case Derivation: Following a C++ style guide will ensure the cleanliness and efficient maintenance of the codebase.

   How test will be performed:

- clang-tidy and clang-format, standard linters and formatters for C++, are ran within the CI/CD pipeline to check for new code's compliance with the Google's style guide

4.3 Traceability Between Test Cases and Requirements

Req. ID	**System Test ID**
FR-1	FR-SLN1, FR-SLN2
FR-2	FR-SLN3
FR-3	FR-SLN4
FR-4	FR-SLN4
FR-5	FR-SLN4
FR-6	FR-SLN5
FR-7	FR-SLN4
FR-8	FR-SLN1, FR-SLN2
NFR-11.1.1	NFR-SLN1
NFR-11.1.2	NFR-SLN2
NFR-11.2.1	NFR-SLN3
NFR-11.3.1	NFR-SLN1
NFR-11.3.2	NFR-SLN1
NFR-11.4.1	NFR-SLN1
NFR-11.4.2	NFR-SLN1
NFR-11.5.1	NFR-SLN1
NFR-12.2	NFR-SLN4
NFR-12.3	NFR-SLN5
NFR-12.6	NFR-SLN5
NFR-13.1	NFR-SLN6
NFR-13.3	NFR-SLN9
NFR-13.4	NFR-SLN1
NFR-14.1.2	NFR-SLN1
NFR-14.2.1	NFR-SLN1, NFR-SLN7

NFR-14.2.2	NFR-SLN6
NFR-14.3.2	NFR-SLN6
NFR-15.1.1	NFR-SLN7
NFR-15.1.2	NFR-SLN7
NFR-15.2.1	NFR-SLN7
NFR-15.3.1	NFR-SLN8
NFR-15.4.1	NFR-SLN2
NFR-15.5.1	NFR-SLN7
NFR-15.5.2	NFR-SLN5
NFR-17.1	NFR-SLN9
NFR-17.2	NFR-SLN9

Table 3: Traceability Between Test Cases and Requirements

5 Unit Test Description

This section will be completed after the detailed design document has been completed.

5.1 Unit Testing Scope

For the purpose of our project, Dr. Kelly and our team identified the two most essential modules to be evaluated using unit-tests, which involve the behaviour-hiding module and the Mujoco module. The rationale for implementing unit tests for these two modules is that the behaviour-hiding module has to verify crucial functionality that is within the TPG class. The Mujoco module needs to be verified because it is a new and more complex environment setup, especially when compared to the classic control module.

5.2 Tests for Functional Requirements

This subsection will be completed after the detailed design document has been completed.

5.2.1 Behaviour-Hiding Module

The unit tests for the Behaviour-Hiding Module are designed to verify the crucial algorithm based methods within the TPG class. These tests ensure that the behavior-hiding mechanisms operate correctly, maintaining the integrity and expected outcomes of the system's operations. The tests are implemented with C++ and with the popular unit testing framework Catch. It is executed automatically through a continuous integration pipeline, ensuring that any changes to the module are validated properly.

5.2.2 Mujoco Module

The unit tests for the Mujoco Module cover various aspects of the Mujoco environment, such as initialization, terminal conditions, reward calculations, control and contact costs, health checks, simulation steps, observation retrieval, and reset functionality. The tests are implemented with C++ and with the popular unit testing framework Catch. It is executed automatically through a continuous integration pipeline, ensuring that the complex environment setup is consistently verified for correctness and performance..

5.3 Tests for Nonfunctional Requirements

Unit tests do not cover non functional requirements

References

Team 3 Tangle. Development plan. https://github.com/TPGEngine/tpg/blob/main/docs/DevelopmentPlan/DevelopmentPlan.pdf, 2024a.

Team 3 Tangle. Hazard analysis. https://github.com/TPGEngine/tpg/blob/main/docs/HazardAnalysis/HazardAnalysis.pdf, 2024b.

Team 3 Tangle. Problem statement and goals. https://github.com/TPGEngine/tpg/blob/main/docs/ProblemStatementAndGoals/ProblemStatement.pdf, 2024c.

Team 3 Tangle. System requirements specification. https://github.com/TPGEngine/tpg/blob/main/docs/SRS/SRS.pdf, 2024d.

6 Appendix

6.1 Symbolic Parameters

This section is not applicable as there are no symbolic parameters used within the project.

6.2 Usability Survey Questions?

Here are some questions that may be asked regarding usability in the form of a survey:

1. What operating system do you use?
   (a) Windows
   (b) Mac OS
   (c) Linux
2. On a scale from 1-10 (higher means better), how would you rate your installation experience?
3. On a scale from 1-10 (higher means easier), how easy would you say it was to execute a simulation environment using MuJoCo?
4. On a scale from 1-10 (higher means better), how would you rate the readability of the documentation?
5. On a scale from 1-10 (higher means better), how would you rate the usability and usefulness of the system's logging?
6. On a scale from 1-10 (higher means better), how would you rate the overall experience of utilizing and modifying the system's custom parameters?
7. If applicable, on a scale from 1-10 (higher means easier), how easy was it to implement changes to the code?
8. If applicable, did you have any trouble integrating your changes to the remote repository?
9. Do you have any feedback or suggestions when it comes to the usability of the system? Please write them down below.

Appendix — Reflection

What went well while writing this deliverable?

Due to the previous effort put into previous resources and documentation, our team was able to create well defined objectives with clear rationales based on our initial work. This groundwork made it much easier to develop a consistent direction for the Verification and Validation Plan, ensuring that each part of the project is aligned with the overarching goals. By referencing our Problem Statement, SRS, and Development Plan, we could confidently outline the project's needs and avoid a lot of back-and-forth on high-level decisions.

Developing an outline for a collaborative process with Dr. Kelly's team was another important part of this documentation. Establishing a feedback loop is important for our project, especially with MuJoCo integration, which is a challenging area. Regular check-ins ensure that we are not overlooking research requirements or deviating from the project's core objectives. It is reassuring to have a system for catching issues early, keeping us on track, and ultimately supporting a more robust integration.

What pain points did you experience during this deliverable, and how did you resolve them?

Some pain points include balancing clarity of our testing plan without a clear understanding of the effort or difficulty of it. Since certain aspects of MuJoCo integration and CI/CD setup were new to us, it was challenging to estimate how complex they would be to validate accurately. This lack of clarity makes it difficult to break down our validation plan into actionable steps, as we weren't sure of the exact methods or tools we would need.

To work through this, we decided to make certain high level goals for us to achieve in our testing plans, and also examined other open-source projects that implemented similar validation processes or functionality to what we are looking for to gain a better understanding.

In addition, making sure our validation plans addressed both functional and non-functional requirements from the SRS felt particularly challenging. It required us to carefully think through how we would test each requirement practically. To tackle this, we created a checklist derived directly from the SRS document, ensuring every key requirement had an associated test or

validation step. This checklist became a helpful guide in our regular review meetings, allowing us to confirm we weren't overlooking critical areas.

What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.

Calvyn:

To complete the verification and validation process, I'll need to focus on gaining a stronger understanding of memory management and debugging tools specific to C++. C++ can be challenging to work with due to potential issues like memory leaks and undefined behavior, which are especially important to detect and resolve in our project due to its reinforcement learning and simulation components. With a better understanding of these tools, I'll be better equipped to analyze our code for memory-related errors and optimize its performance, which will be crucial for ensuring stability and efficiency within our CI/CD pipeline.

Edward:

One of the key knowledge areas I will need to develop is understanding the various indicators of correct agent behavior across different MuJoCo environments. This includes interpreting fitness scores, analyzing plot outputs, inspecting animations, and recognizing expected behavioral patterns. Since each environment has unique characteristics and learning dynamics, this will require developing a deeper familiarity with TPG's learning progression and how it manifests across different scenarios. Currently, the workflow involving running experiments, checking logs, interpreting plots, and debugging issues is still somewhat opaque, requiring significant guidance from Dr. Kelly to understand the nuances of what we're observing.

Cyruss:

The knowledge and skills that the team will need to successfully acquire

to complete the verification and validation of the project include a variety of testing knowledge such as unit testing, and automation testing. These verification and validation techniques will be beneficial to easily debug and determine any failures within the system. With the use of automation testing, it will be beneficial to automatically execute repetitive tests. This will ensure consistency and efficiency while reducing human error.

Mark:

The team will collectively have to acquire knowledge in both dynamic and static testing, a few testing tools such as clang-tidy, clang-format, Google Test, and CI/CD integration using Github Actions. Dynamic and static testing will help the team to perform manual testing across the framework. Clang-tidy, clang-format, and Google Test will help automate the testing and verification process, saving plenty of man hours from the team. Lastly, the CI/CD integration through Github Actions will provide a comprehensive set of testing and verification prior to the team committing newly developed code into the codebase, saving less headaches in the future.

Richard:

The main knowledge areas for our project involves software engineering standards. An important thing for any open source framework is the ease of onboarding and utilizing the library. One way we can validate this is to be able to easily onboard and get an experiment running on any operating system that the user is using. The documentation also needs to be clear and concise to allow for the best developer experience. Specific knowledge related to cross platform C++ build tools, and containerizing the framework will be key for this. In addition, static code analysis will also be critical to evaluate the structure of our code. This is key because we want to ensure our code is robust, and also easy to use and build on top of since our supervisor Dr. Kelly wants this framework to be used as an open source tool.

For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?

Calvyn:

I believe that the first method I should use to gain a better understanding of debugging errors is to utilize the regular feedback sessions with Dr. Kelly to discuss testing, debugging, and other technical issues. These sessions would allow team members to ask questions, review complex aspects of the project, and receive guidance on best practices for testing and validation, helping the team refine their approach and stay aligned with project goals. Another approach would be to browse existing threads that discuss testing issues and solutions related to C++ can offer valuable context and practical examples. Analyzing how other developers approach similar challenges can help the team apply these strategies to their own project, learning from both successes and mistakes of others.

Edward:

There are two main approaches I plan to pursue to develop this knowledge. The first would be to build understanding incrementally through guided exploration. This involves starting with environments where we have clear expectations (like CartPole), consulting with Dr. Kelly to understand what indicators to look for, and documenting these learnings to apply to new environments. Regular communication with the team about observations and issues is crucial since everyone brings different insights to interpreting TPG's behavior. Secondly, I would take a systematic experimentation approach. This means making controlled modifications to environment parameters or TPG settings on separate GitHub branches, carefully documenting the impacts, and building a mental model of cause-and-effect relationships. By having a safe space to experiment without fear of breaking the main codebase, I can learn through trial and error while maintaining rigorous version control.

Cyruss:

Two approaches to acquiring the knowledge of both skills include hands-on approach testing with different tools without any guidance, or using on-

line self-learning techniques such as Linkedin Learning or Udemy. I will be pursuing the online self-learning approach as I work better with structured guidance and clear explanations. Additionally, these courses often include practical examples and quizzes, which will help reinforce my understanding.

Mark:
A few approaches that can be used to attain knowledge of these skills are reading documentations and watching visual tutorials. Documentations help users tremendously as most if not all are usually well made, especially if they are enterprise standard tools. Watching visual tutorials may be another way to master these areas and skills through the use of YouTube, or Coursera. Each person has a preferred way of learning. They can either go the non-visual part of reading documentations and articles or visual tutorials. It all comes down to what saves the contributor the most time, while having the most yield from the content.

Richard:

There are a few approaches I can take to attain knowledge. For one, there is a group in France that has built an open source framework on top of Dr. Kelly's TPG framework is called Gegelati. The Gegelati project is open source on Github and has implemented a lot of the CI/CD infrastructure and also enables cross platform development. Additionally, there are numerous C++ libraries that build with cross platform tools such as MuJoCo itself (developed by Google DeepMind) so seeing those examples will help guide me. Second, is to view YouTube videos on the topic to see how other developers tackle and approach this problem. Getting more opinions and seeing best practices in a visual format will be beneficial.