Verification and Validation Report: TPG

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

February 26, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

symbol	description
TPG	Tangled Program Graphs
DNNs	Deep Neural Networks
RL	Reinforcement Learning
SRS	Software Requirement Specification
FR	Functional Requirement
NFR	Non-Functional Requirement
SLN	System Level Number
VnV	Verification and Validation

Contents

List of Tables

List of Figures

This document cohesively summarizes the results of each test as specified in the VnV Plan documentation.

3 Functional Requirements Evaluation

3.1 MuJoCo Integration

Table 1: MuJoCo Integration Tests

Test Id	**Notes**	**Result**
FR-SLN1	When executing the appropriate script, all MuJoCo environments can be run. The best-performing agent within the policy can be visualized using OpenGL or an MP4 file.	Pass
FR-SLN2	MuJoCo environments within the TPG framework can be successfully run within the Digital Research Alliance, enabling research to be conducted by executing experiments.	Pass

3.2 Experiment Visualization

Table 2: Experiment Visualization Tests

Test Id	**Notes**	**Result**
FR-SLN3	When an experiment is running or finished training, the best performing policy can be visualized using the TPG CLI tool.	Pass

4 Nonfunctional Requirements Evaluation

4.1 Usability

4.2 Performance

4.3 Operational and Environmental

Table 3: Operational and Environmental Tests

Test Id	**Result**
NFR-SLR6	Pass

For NFR-SLN6, TPG now supports contributions from macOS, Windows, and Linux developers. Previously, only Linux was supported because TPG used SCons for C++ builds and Linux-specific dependencies from requirements.txt. With VSCode Dev Containers, a Linux development environment is automatically launched for all developers, ensuring a standardized setup. Simply follow the Wiki instructions to download all necessary Linux dependencies and build the C++ code reliably. Onboarding on a Macbook has been reduced from 2 weeks to just 5 minutes.

4.4 Compliance

Table 4: Compliance Tests

Test Id	**Result**
NFR-SLR9	Pass

The modified codebase is successfully analyzed using Clang-Tidy and Clang-Format within the CI/CD pipeline. Code change discussions take place through pull request conversations made to the main branch. All errors and warnings are generated based on the C++ Style Guidelines. Any critical

errors found during the linting process create blocking pull request conversations that must be resolved before merging into the main branch.

Figure 1: Example of a Linter Error

5 Comparison to Existing Implementation

The starting point for our project was a repository maintained by Dr. Kelly's research team. As is common with many academic projects, the original implementation focused on exploratory research rather than robust software engineering practices. While the code worked for experimental purposes, it lacked adherence to formal open source library standards—particularly in areas such as code modularity, automated testing, and the development workflow.

5.1 Enhancing Code Quality and Development Practices

Open Source Best Practices: We integrated industry-standard practices, including CI/CD pipelines, branch protection, and automated builds. Every pull request now triggers unit tests to ensure that new contributions do not break the existing logic.

Code Refactoring and Modularity: Previously, experiment logic was contained within a single `Main` function, making maintenance and extension challenging. We refactored the code into well-organized classes, enhancing

both modularity and maintainability. Additionally, we adopted ClangFormat with the Google C++ style guide and migrated from SCons to CMake—a modern and widely used C++ build tool.

5.2 Introduction of New Features

Unit Testing: We have implemented unit tests for MuJoCo environments and key algorithms like Register Machine Crossover and Team Crossover. These tests are integrated into our automated build pipeline to ensure consistent validation of new features.

Enhanced Experimentation and User Experience: Previously, users had to manually run `cd $TPG/experiments/generic` and execute a complex shell script with numerous parameters. This process was error-prone and inflexible. In contrast, our new CLI tool—built in Python using the Click library—allows users to run experiments from any directory with a simple command (e.g., `tpg evolve multitask`). The CLI provides built-in help and clear documentation, while automatically organizing outputs (logs, plots, replays) into designated directories, thereby streamlining the workflow.

Modern Development Environment Support: We containerized the development environment using Docker. This Linux-based container ensures a consistent deployment environment, reducing platform-specific issues during development and testing.

6 Unit Testing

7 Changes Due to Testing

7.1 Feedback from Rev 0

The feedback given by the instructor and teaching assistant during Revision 0 was essential in guiding the next steps as the team looks toward the final demonstration. Emphasis was placed on ensuring that usability testing was executed systematically rather than in the more ad-hoc manner initially planned by the team. Some additional changes to be made include ensuring that unit testing and benchmarking of the implemented environments are

cohesively executed and investigating whether the integration of deployment within the DRA is possible.

8 Automated Testing

As a result of the team's conversion from building the project using SCons to CMake, automated testing became significantly easier to execute and debug. To run any automated tests within a developer's local environment, a developer can simply execute a command to build the project. This not only compiles everything but also runs all automated tests. If a developer wishes to run only the tests, they must navigate to the directory where the tests were already compiled (typically `/build/tests`). From there, the command `ctest` can be entered into the command prompt. Similar to the compilation process, all automated tests are executed once this command is run.

From the repository's point of view, tests are executed using GitHub Actions or GitLab CI (depending on which repository is being viewed). Both linting and compilation are performed using the same commands that would be executed within a developer's local environment. These tests run when a new pull request is made to the main branch, ensuring that all tests pass before merging. The compiler workflow is also executed after merging into the main branch to ensure no errors or unintended changes in code behaviour have occurred. If any test or workflow fails, the logs of the workflow can be reviewed, providing a detailed summary of the reason for failure. This not only allows for easier debugging but also resolves the "works on my machine" issue.

9 Trace to Requirements

10 Trace to Modules

11 Code Coverage Metrics

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Reflection.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. Which parts of this document stemmed from speaking to your client(s) or a proxy (e.g. your peers)? Which ones were not, and why?
4. In what ways was the Verification and Validation (VnV) Plan different from the activities that were actually conducted for VnV? If there were differences, what changes required the modification in the plan? Why did these changes occur? Would you be able to anticipate these changes in future projects? If there weren't any differences, how was your team able to clearly predict a feasible amount of effort and the right tasks needed to build the evidence that demonstrates the required quality? (It is expected that most teams will have had to deviate from their original VnV Plan.)