Verification and Validation for Systematic Benchmarking Test Case Generation

Team 1

Emma Willson

Esha Pisharody

Grace Croome

Marie Hollington

Proyetei Akanda

Zainab Abdulsada

February 7, 2025

Table of Contents

1. Versions, Roles, and Contributions

1.1. Version History

Version #	Authors	Description	Date
0	All	Created first draft of document.	February 7, 2025
1	All	Revised document and added test results.	April 4, 2025

1.2. Table of Contributions

Group Member	Contributions (Sections)
Emma Willson	3.1, 3.4
Esha Pisharody	1, 2, 3.5
Grace Croome	3.4, 3.5
Marie Hollington	3.2, 3.5
Proyetei Akanda	3.7, 3.8
Zainab Abdulsada	3.3, 3.6

1.3. Team Information

Group Members	Contact & ID	Roles
Emma Willson	willsone@mcmaster.ca 400309856	Research Lead Developer
Esha Pisharody	pisharoe@mcmaster.ca 400325118	Developer
Grace Croome	croomeg@mcmaster.ca 400313932	Developer
Marie Hollington	hollim3@mcmaster.ca 400320562	Developer
Proyetei Akanda	akandap@mcmaster.ca 400327972	User Interface Lead Developer
Zainab Abdulsada	abduslaz@mcmaster.ca 400313736	Project Manager Developer

2. About the Project

2.1. Purpose of the Project

Lean, Idris, Agda, and Rocq are functional programming languages widely used as proof assistants. However, current user experience with these languages suggests that there are many low-level deficiencies. In this project, the aim is to systematically test these proof assistant languages to highlight potential areas for improving their performance. The goal is to create an automated code generator that generates tests written in these languages of increasing size, measures the time and memory complexity of running these tests, and displays the results in graphs on locally generated webpages. The user will interact with the code generator through a GitHub Actions, where they will have the ability to choose the specific test case and the number of operations to be carried out. The purpose of this document is to ensure that our software meets all specified requirements and standards outlined in the software requirements specification and fulfils its intended purpose for stakeholders. It outlines the testing plan for each component of the system defined in the design document. This will aid in improving product quality, minimize risks and costs (time) associated with future issue resolution, allowing us to deliver a reliable and effective result.

2.2. Naming Convention and Terminology

- **CLI:** Command line interface, a text-based system that allows a user to interact with computer programs.
- **Proof assistant:** A proof assistant is a software tool used to assist a user with formalizing a logical or mathematical proof, ex. Agda, Lean, Idris, Rocq, or Isabelle.
- **MHPG:** Mini Haskell Proof Grammar refers to the Haskell grammar we will be creating to format automated proofs.
- **EP:** Elementary Proofs are the base set of test cases written in MHPG which will be the foundation from which tests of increasing size are generated.
- **PAL:** Proof assistant language, referring to the four languages we are working with: Agda, Lean, Idris, and Rocq.
- **UNIX:** Uniplexed Information Computing system, a multiuser and multitasking operating System.

3. Component Test Plan

3.1. Command Line Interface

3.1.1. Unit Testing

When the CLI prompts the user for input, it is expected that the correct options will be selected, otherwise an error message will be displayed. It is also expected that the user will select a test size within a valid range, and if the selected size is out of range, an error message will be displayed. To ensure all tests are properly reachable through the CLI, each test option will be passed through the CLI, in addition to the same checks for the valid test size range.

3.1.2. Generate Range Workflow – Size Testing Results

Input	**Expected Output**	**Actual Output**
Negative Lower Bound	Error: Incorrect input, must be >=0	Error: Incorrect input, must be >=0
Upper Bound lower than Lower Bound	Error: Upper bound must be >= lower bound and <= max limit	Error: Upper bound must be >= lower bound and <= max limit
Upper bound above max upper bound	Error: Upper bound must be >= lower bound and <= max limit	Error: Upper bound must be >= lower bound and <= max limit
Correct lower limit and upper limit (including edge cases)	Compiles	Compiles
Set max memory < 1	Error: memory must be between 1-15GB	Error: memory must be between 1-15GB
Set max memory > 15	Error: memory must be between 1-15GB	Error: memory must be between 1-15GB
Number of Points in graph exceeds range	Only outputs points within range	Only outputs points within range
Invalid interval type	Error: Please provide valid input (linear, log, quadratic)	Error: Please provide valid input (linear, log, quadratic)
Number of points = 200	Error: Max 150 data points	Error: Max 150 data points

3.1.3. Generate Range Workflow – Test Selection Testing Results

Input	**Expected Output**	**Actual Output**
Select test from drop down in GitHub Actions	Compiles	Compiles
Locally input out of bound test number	Error: Available ID’s available between 1-21	Error: Available ID’s available between 1-21

3.1.4. Generate List Workflow – Size Testing Results

Input	Expected Output	Actual Output
List with spaces: 1, 2, 3, 4	Error: Incorrect input	Error: Incorrect input
Negative input	Error: Incorrect input, must be >=0	Error: Incorrect input, must be >=0
Highest element in list above max upper bound	Error: Upper bound must be >= lower bound and <= max limit	Error: Upper bound must be >= lower bound and <= max limit
Correct lower limit and upper limit (including edge cases)	Generates data and deploys webpage with results	Generates data and deploys webpage with results
Duplicate values in list	Compiles, ignoring any duplicates	Compiles, ignoring any duplicates
Providing 200 list elements	Error: Max 150 elements acceptable	Error: Max 150 elements acceptable

3.1.5. Generate List Workflow – Test Selection Testing Results

Input	Expected Output	Actual Output
Select test from drop down in GitHub Actions	Generates data and deploys webpage with results	Generates data and deploys webpage with results
Locally input out of bound test number	Error: Available ID's available between 1-21	Error: Available ID's available between 1-21

3.2. CI Pipeline

3.2.1. Unit Testing

The expected behaviour for the build pipeline is that it is triggered manually or on a push event, and it builds and pushes a container to a docker registry. Once complete, the registry can be checked to ensure the container was pushed successfully. On failure, GitHub Actions will flag where the failure occurred, allowing follow up for resolution.

The generate test cases pipeline(s) takes in a manual input from the user (a specific test case from a selectable list, and the test case size(s)) and follows through with the project structure: generate test case, record time and memory, visualize and produce webpage. The generate test cases pipeline(s) use the CLI. For testing, refer to the Generate-Range and Generate-List Workflow Test Results sections. Any errors that occur are displayed in the job logs of the failed workflow.

The startup-time pipeline is triggered manually and records the startup-times of each test case for each language. Once complete a JSON file with these recorded times will be stored as an artifact. On failure, GitHub Actions will flag where the failure occurred, allowing follow up for resolution.

The test pipeline takes manual input from the user (a specific test case from a selectable list, and the test case size) and then generates the translated test case at the given size in Lean, Idris, Agda, and Rocq and then type checks each file in their respective languages. The pipeline displays the output of the translated files and any error that occurs. On failure, GitHub Actions will flag where the failure occurred, allowing follow up for resolution.

3.2.2 Build Workflow – Trigger Testing Results

Trigger	**Expected Output**	**Actual Output**
Manual build	Push container	Push container
Push to repository	Push container	Push container

3.2.3 Startup-time Workflow– Trigger Testing Results

Trigger	**Expected Output**	**Actual Output**
Manual	JSON file stored	JSON file stored

3.2.4 Test Workflow – Test Case Size Testing Results

Input	**Expected Output**	**Actual Output**
Size 29	Error: Test size 0-20	Error: Test size 0-20
Size –1	Error: Test size 0-20	Error: Test size 0-20
Size 15	Compiles and Produces Test in the 4 PAL and prints them	Compiles and Produces Test in the 4 PAL and prints them

3.3. MHPG

3.3.1. Unit Testing

The expected behaviour is that all tests written in MHPG in *test.hs* are properly handled and translated. This itself ensures valid design and implementation of the grammar. To ensure validity, tests of various types need to be created, involving the various data structures presented in the grammar to ensure things work not only as standalones, but in unison as well.

In other words, testing for MHPG is simply reliant on a correct, compilable, runnable test case module.

3.3.2 Testing Results

Testing of MHPG is dependent on compilation of translated files, as long as testing for those is all correct, it is implied that MHPG is correct.

3.4. Test Case Module

3.4.1. Unit Testing

The expected behaviour is that all tests written in MHPG should be able to compile correctly in Haskell, ensuring correct MHPG syntax. Once this is ensured, it can be verified that the test cases are correctly translated to all PALs. These tests should also be able to take in any value of size within the given range and output a properly translatable test.

Unit testing for test cases is dependent on the translation of test cases. If these are correct, then the original written test case must also be correct.

3.4.2 Testing Results

Input	**Expected Output**	**Actual Output**
Tested all cases: input 0	Empty translated files with Module header	Empty translated files with Module header
Tested all cases: input 1	Compiled successfully	Compiled successfully
Tested all cases: input mid allowable range	Compiled successfully	Compiled Successfully
Tested all cases: max upper bound	Compiled successfully	Compiled successfully

3.5. Translators

3.5.1. Unit Testing

The expected behaviour of the translators is that they will take a test case written in MHPG and correctly translate it into the target languages: Lean, Idris, Agda, and Rocq. The resulting files after translation should compile without errors.

Unit testing will be done by type checking the output files for each PAL, to ensure correct translation. This will be done for edge cases of size 0, 1, and the maximum size of each test.

A key standard to maintain throughout the translation process is ensuring that no special shortcut syntax specific to any particular language is used. This is to ensure unbiased time and space utilization comparisons.

3.5.2 Testing Results

Translators working correctly per dependence on Test Case Module testing. As all compiled with excepted output, translators are correct.

3.7. Graph Visualization

3.7.1. Performance Tests and Metrics

Upon executing the graph visualization script via the CLI, the Python script leverages the matplotlib library to process a JSON file generated by the test case script. For each test case,

the script generates four performance graphs: **real time**, **user time**, **system time**, and **memory usage**.

The verification process ensures that:

- The script correctly parses the JSON file and extracts all required attributes (size, time, space) for each PAL.
- Graph legends are accurate and clearly displayed.
- Axis scaling remains appropriate and readable, even for large datasets.
- When multiple PALs are plotted on a single graph, visual clarity is maintained with no significant overlap.
- The script successfully handles a wide range of inputs including very large and very small values by testing across various configurations

3.7.2. Unit Testing

We verified that that the app dynamically processes data.json from GitHub Actions and the python script can process the file properly with no missing fields. If there are missing fields, those cases are handled with edge cases by displaying an error message. The graph images are base64-encoded and embedded directly into the HTML, and unit tests confirm correct in-memory generation and encoding.

3.8. Webpage

3.8.1. Performance Tests and Metrics

The primary metric we aim to test is the latency of how long it takes to generate and load the graphs on the website. Specifically, we monitored how the application handles the increased load when there are significantly more test cases. Additionally, we ensured that the website remained responsive and user-friendly across all devices even when displaying complex graphs.

3.8.2. Unit Testing

We ensured when users click on the website link from GitHub Actions they are able to view the website with no issues. On the website it displays error messages and its information like the size, error type and what language caused it, so we ensured that for each tests its showing the correct data.

3.8.3. Integration Testing

We ensured that the frontend is integrated with Flask backend. We verified that Flask routes respond with fully formed HTML pages containing valid, embedded graphs and there are no broken image links on Vercel. We ensured that the Flask backend is integrated seamlessly with Vercel and that there are no deployment issues.