Design Document for

Systematic Benchmarking Test Case Generation

Team 1

Emma Willson

Esha Pisharody

Grace Croome

Marie Hollington

Proyetei Akanda

Zainab Abdulsada

January 26, 2025

Table of Contents

1. Versions, Roles, and Contributions

1.1. Version History

Version #	Authors	Description	Date
0	All	Created first draft of document.	January 26, 2025
1	All	First major revision	April 4, 2025

1.2. Table of Contributions

Group Member	Contributions (Sections)
Emma Willson	4, 5
Esha Pisharody	1, 2, 3
Grace Croome	5
Marie Hollington	4, 5
Proyetei Akanda	5, 6
Zainab Abdulsada	3, 4, 5

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 and tables on locally generated webpages. The user will interact with the code generator through a local command line interface or GitHub Workflow, where they will have the ability to choose the specific test case and the number of operations to be carried out.

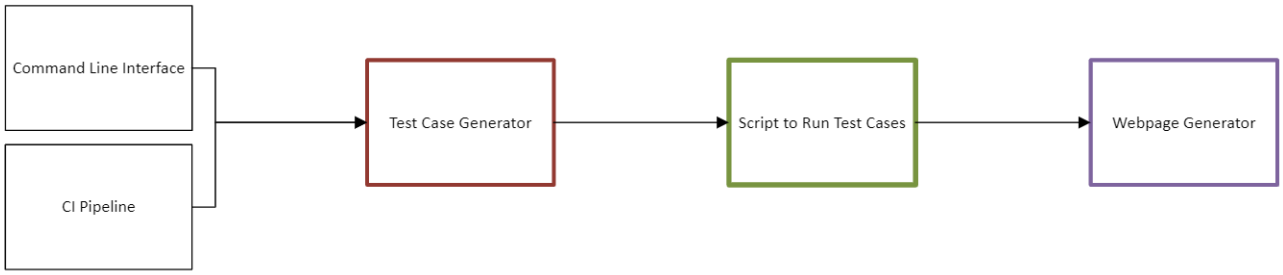
The purpose of this document is to provide a comprehensive overview and detailed description of all the components of the system, the relationships between each component and the requirements as defined in the software requirements specification, as well as a detailed breakdown of the user interface.

2.2. Naming Convention and Terminology

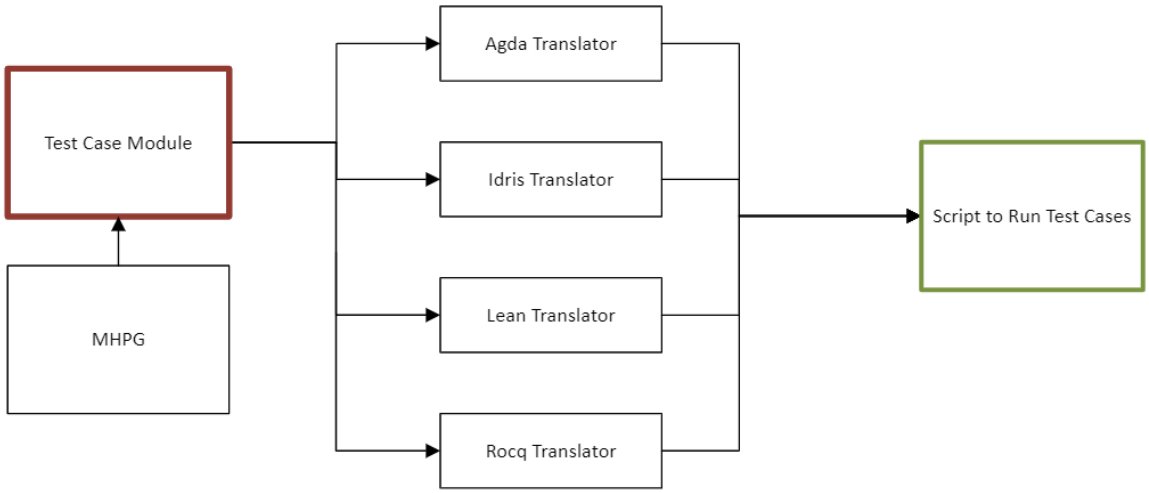
- **CLI:** Command line interface, a text-based system that allows a user to interact with computer programs.
- **Proof assistant:** A proof assistant is a software tool used to assist a user with formalizing a logical or mathematical proof, ex. Agda, Lean, Idris, Rocq, or Isabelle.
- **MHPG:** Mini 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 Diagram

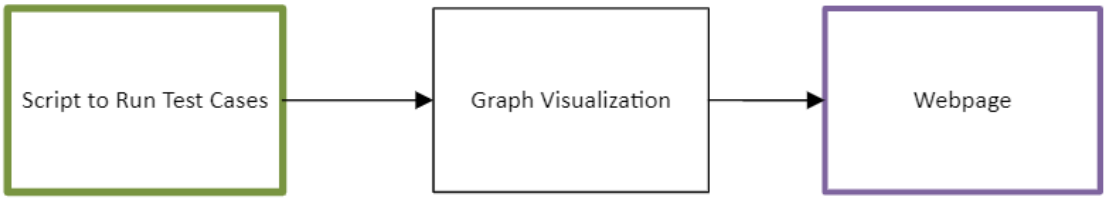
Overall Flow of Final Product:



Test Case Generator:



Webpage Generator:



4. Relationship Between Components and Requirements

System Component	Requirements Covered
Command Line Interface	• P2: The system will require users to select which type of test case is generated. • P2: The system will only allow users to select one type of test case per generation request. • P2: The system will allow users to select how many versions of each test case is generated, with a lower and upper limit. • P2: The system will validate user input for all parameters and provide an error message if the input is invalid. • Non-functional: Usability and Human.
CI Pipeline	• P0: The system will generate a set of tests of increasing sizes from the initial set. • P0: The system will compile the test cases in each PAL. • P0: The system will record the compilation time of each of these test cases in the four PALs.
MHPG	• P0: The system will contain a hard-coded initial set of EPs written in our proof grammar (MHPG). • Non-functional: Interfacing Requirements
Test Case Module	• P0: The system will be given a base set of EPs of various types. • P0: The system, when given a number of operations and a test from the initial set, will increase the size of the given test case by the provided number of operations.
Agda, Idris, Lean, Rocq Translators	• P0: Post-test case generation, the system will translate them into the four PALs. • P3: The system will incorporate Isabelle as one of the PALs available for translation. • Non-functional: Precision and Accuracy.
Script to Run Test Cases	• P1: The system will measure the time and memory complexity of each test case as determined by comparison of the same base case across increasing sizes.
Graph Visualization	• P0: The system will generate a JSON file including the time complexity, space complexity and size for each PAL and process it as input data.

	• P1: The system will visualize the measured data in graphs and tables. • Non-functional: Usability and Human
Webpage	• P0: The system will provide documentation on how to extend the test generator with new classes of tests. • P0: The system will display the compilation time and memory of each of these test cases in the four PALs. • P1: The system will provide a description for each graph displayed on the webpages, explaining the data being visualized. • Non-functional: Usability and Human

5. Detailed Description of Components

Command Line Interface	
Normal behaviour	The user can view the list of test cases which can be generated and select a test case as well as the size and number of tests generated. They can specify that a test is generated at specific sizes or a range of sizes.
API	Upon receiving user input, the CLI tool will call the test case module, potentially multiple times, with the given test choice and size(s). After receiving the generated test case(s), the CLI will pass them to the Agda, Lean, Idris, and Rocq translators.
Implementation	The JSON library in Go parses the time and space data.
Potential undesired behaviours	The CLI tool may terminate before receiving valid input due to uncaught parsing errors.

CI Pipeline	
Normal behaviour	The CI pipeline, when manually triggered, will generate increasing sizes of a given test case (eg. 1-100 operations) and translate the test cases into each of the 4 languages: Agda, Idris, Lean, and Rocq. After translation, each test case is type checked, and the pipeline records the time and space consumed during the type checking operation. The results are then visualized as graphs on a generated webpage.
API	N/A
Implementation	The pipeline is implemented with Github Actions. The test cases will be generated and type checked by a Go CLI tool in a docker container that has Agda, Lean, Idris, Rocq, and Haskell installed. The time and space data will be parsed and piped into a JSON

	file. A Python script will take the JSON file as input and generate a webpage with graphs visualizing the results.
Potential undesired behaviours	Test case generation or translation may fail for certain sizes or other edge cases. The type checking operation could timeout or stop due to excessive resource consumption.

MHPG	
Normal behaviour	MHPG stands for Mini Haskell Proof Grammar. This module contains the generalized grammar used to write test cases. The test cases will be generated in this grammar structure, which also serves as the start point for translation.
API	N/A
Implementation	The grammar will be built using Haskell, with each key language structural component and definition represented as required. This includes function and variable definitions along with int, bool, binary operations etc.
Potential undesired behaviours	The generalization of components from different PALs may result in confusion between constructors.

Test Case Module	
Normal behaviour	This module contains a list of scalable test cases which, when given the index of a test and an input number through the Command Line Interface, will generate a test case written in MHPG with a relative ‘size’ of the input number.
API	Upon being called by the CLI with arguments index and size, the test case module returns the test at that index, scaled to that size.
Implementation	Each scalable test case takes an integer n >= 0 and recursively builds a test case in MHPG. When called, a scalable test case will recurse n times. These scalable tests are declared in a list.
Potential undesired behaviours	Generated test cases may not be useful for some edge cases. Test case generation may be slow if the scaling is not implemented well.

Agda, Lean, Idris, Rocq Translators	
Normal behaviour	This module takes in a test case written in MHPG and translates it to a string containing a program in the target language. This is then written to a file of the corresponding language.
API	Upon being called by the CLI with argument test, the translators generate a program file containing that test translated to their language.
Implementation	Each rule of the MHPG is translated to a string and/or other rules (in a way that is unique to each target language). The

	untranslated test case is unwrapped and then translated recursively.
Potential undesired behaviours	Errors in translation could cause output files that do not run properly in the target language. Another potential undesired behaviour, due to the static nature of each test case's name, is the accidental overwriting of a previous test case with a different size.

Graph visualization	
Normal behaviour	The python script will first process the JSON file produced from GitHub Actions, loading the benchmark data from which contains timing/memory metrics, exit status, errors for different input sizes across languages. The app uses Python's matplotlib to create four types of performance graphs for each test case: Real Time, User Time, System Time, and Memory.
API	N/A
Implementation	The displayIntervalType function returns the interval type (log, linear, or quadratic) for graph labeling. checkExitStatus adds red or blue "x" markers on graphs to indicate memory or time errors, while plotArtificialZeros highlights missing timing values with cyan squares. get_label_position and adding_annotations work together to position and render language labels without overlap. The four main plotting functions generate performance graphs for each test case and return base64-encoded images. generate_graphs loops through all test cases to collect these images, and get_error_messages identifies and formats any runtime errors. The root Flask route / compiles the test case data, error messages, and graphs into a rendered HTML page using the index.html template.
Potential undesired behaviours	Potential undesired behaviours include incorrect graph rendering due to improper input data formatting or issues in the JSON data and cluttered graph labels or overlapping lines if there are many test cases and languages, which might hinder clarity.

Web Pages	
Normal behaviour	Flask will be used to handle routes for dynamically rendering the web page with the graph visualizations. Flask serves as the web framework for generating dynamic webpages that include the graphs. Then it is integrated with HTML/Tailwind CSS for the frontend where we display the benchmark results in the

	website. Since Vercel (our hosting platform) doesn't support saving image files, the app will generate graphs in memory (no files saved). Then it will convert them to base64 text strings and finally embed them directly into the webpage. It shows the four types of performance graphs (real/user/system time + memory) as embedded images.
API	Vercel API was used for integrating Flask with Vercel for deployment We also used: - GET /v6/deployments for getting list of preview deployments - DELETE /v13/deployments/{deployment_id} for deleting old or excess preview deployments
Implementation	The Flask backend uses the / route to generate performance graphs by processing test case data and encoding the plots as base64 images. It extracts the selected test case's name, description, interval type, and any runtime errors, then passes all this data to the index.html template using render_template. On the frontend, Tailwind CSS styles the layout, and Jinja2 templating is used to dynamically display the graphs, error messages, interval type, and explanatory content. This integration allows for a fully dynamic and visually responsive web interface driven by backend-generated data.
Potential undesired behaviours	Slow page loading times when generating large datasets or numerous graphs, affecting user experience.

6. User Interface

6.1. Command Line Interface

The user will interact with the code generator through a command line interface. The CLI will list the test cases available for generation as well as instructions for generating test cases. From here, the user is able to select a test case and number of operations based on the provided list and view the results through the graphical interface.

6.2. Graphical User Interface

Initial Figma design

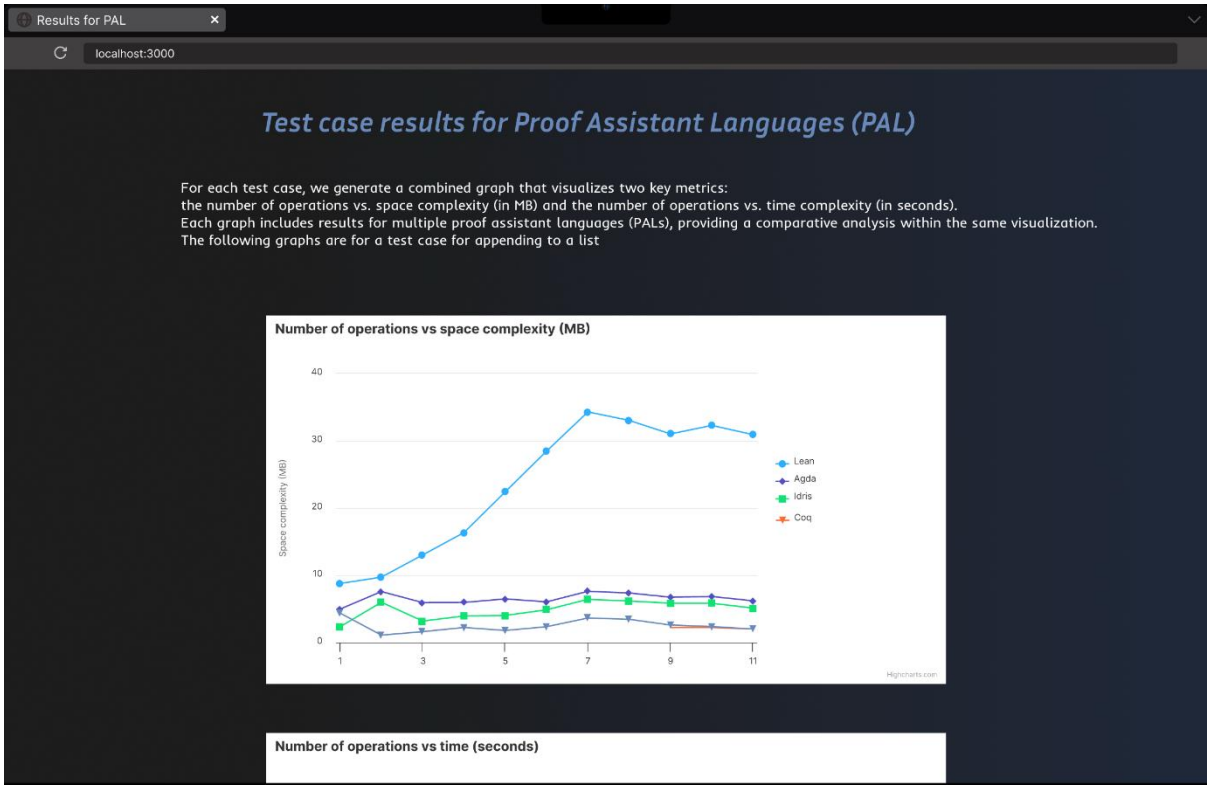


Figure 1: Webpage mock-up to display space complexity of test cases.

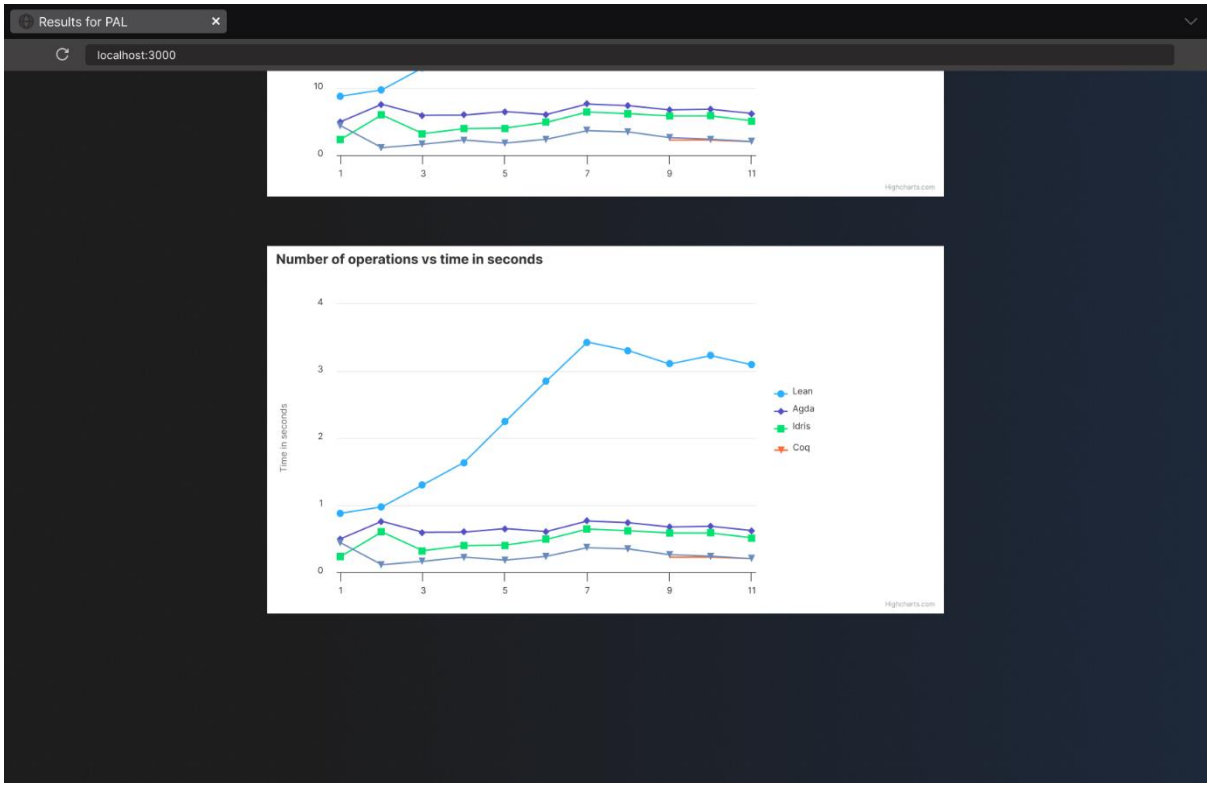


Figure 2: Webpage mock-up to display time complexity of test cases.

Explanation

For each test case, we generate a combined graph that visualizes two key metrics: the number of operations vs. space complexity (in MB) and the number of operations vs. time complexity (in seconds). Each graph includes results for multiple proof assistant languages (PALs), these metrics are plotted on a shared graph, with space complexity displayed on the primary y-axis and time complexity on a secondary y-axis. The example depicted in the Figma design illustrates the results incorporated into a website for a single test case. A typical test case might involve operations such as appending to a list.

Final Website and graphs

Website displays benchmark results in the website. It shows the four types of performance graphs (real/user/system time + memory) as embedded images. For each proof assistant, if there is a red 'X' marker it indicates that the test case exceeded memory, and a blue 'X' marker indicates a time limit exceeded. A bright blue square marker indicates that there is an artificial zero (startup time is larger than type check time) at that point. If an error like time/memory occurs, a message will be displayed detailing the size, language, and type of error. It will specify which language failed, the nature of the error, and the size on which it occurred.