Software Requirements Specification for Systematic Benchmarking Test Case Generation

Team 1

Emma Willson

Esha Pisharody

Grace Croome

Marie Hollington

Proyetei Akanda

Zainab Abdulsada

October 11th, 2024

Table of Contents

1. Versions, Roles and Contributions

1.1. Version History

Version #	Authors	Description	Date
0	All	Created first draft of document.	October 11, 2024

1.2. Table of Contributions

Group Member	Contributions (Sections)
Emma Willson	2.3, 2.4, 2.5, 2.7, 4.1
Esha Pisharody	1, 2.1, 2.2, 2.6, 3.1, 4.3
Grace Croome	2.3, 2.4, 2.5, 2.7, 3.2, 4.1, 5
Marie Hollington	3.2, 4.1, 4.4
Proyetei Akanda	3.2, 4.1, 4.4
Zainab Abdulsada	2.1, 2.2 2.4, 2.6, 3.1, 4.3

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. Purpose of the Project

2.1 User Business

Through this project we focus on modern interactive proof assistants (such as Lean[1], Idris[2], Agda[3] and Coq[4]), which are widely used in formal verification to ensure the correctness of mathematical proofs and software systems. Despite their importance, these systems have not been thoroughly tested for efficiency in terms of time and memory complexity. This project seeks to systematically test and compare these proof assistants to identify areas that can be optimized, highlighting potential areas for improving their performance. This initiative is driven not by a specific business problem, but by a significant opportunity to enhance the effectiveness of these tools. By pinpointing design inefficiencies and areas for improvement, this project aims to produce a comprehensive analysis, to guide future research in the optimization and development of more efficient proof-checking systems.

2.2 Goal of the Project

The goal of this project is to design and build an automated code generator that creates a series of tests of increasing size that will test the efficiency of modern interactive proof assistants including Lean, Idris, Agda, and Coq. Testing will first be performed on these assistants as 'language', then we will move forth with proof testing. The deliverables will include a research paper outlining our findings through systematic testing, in addition to documentation on how to extend the test generator with new classes of test proofs, and the creation of test suites for the new class of proofs translated into each of the proof assistants. The project will also include a command line interface enabling users to view the time and memory complexity and provide a link redirecting them to local webpages presenting visualizations of the measured data.

By providing this analysis and method of performance evaluation of the various proof assistants, this project will enable researchers to identify key inefficiencies in order to optimize future designs of proof-checking systems. Furthermore, the generated webpages will enhance user exploration and analysis of the different proof assistants, allowing for easy comparison.

---

[1] https://lean-lang.org/
[2] https://www.idris-lang.org/
[3] https://agda.readthedocs.io/en/latest/getting-started/what-is-agda.html
[4] https://coq.inria.fr/

2.3. Stakeholders

2.3.1 Primary

Dr. Moradi and Mehrdad Eshraghi Dehaghani; Dr. Jacques Carette & research team; COMPSCI 4ZP6 Team 1; other programming language researchers

2.3.2 Secondary

Developers of Lean, Idris, Agda, and Coq, as well as users of these proof assistants.

2.4. 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, Coq, or Isabelle[5].
- **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 Coq.
- **UNIX:** Uiplexed Information Computing system, a multiuser and multitasking operating System

2.5 Relevant Facts and Assumptions

- The user is proficient with the logic underlying the writing of proofs.
- Test cases are syntactically valid.

2.6 The Scope of the work

2.6.1 The Current Situation

Currently, modern interactive proof assistants have not been thoroughly tested for efficiency. In correlation, we do not find any tools to directly allow users to interactively compare various proof assistants.

[5] https://isabelle.in.tum.de/

2.6.2 Work Partitioning

Step	Description
Create sample tests	Create a set of tests, in each PAL, to have a solid basis on how to create MHPG.
Set up Continuous Integration	Set up the project with proper Continuous Integration infrastructure (including the installation of Agda, Lean, Idris, and Coq and the infrastructure for running the generator). This will make it easier for us to implement automated testing via Continuous Integration.
Create MHPG	Proceed with creating the Haskell Grammar used to write test cases.
Initialize EP	Create the elementary tests using MHPG.
Increase proof size	Given user input for type of proof and preset number of operations, corresponding algorithm for proof type will be called to increase EP size.
Translate	Once a complex proof is generated, it will be translated into the four PALs.
Compile and time	Then a script will be called to compile the proof in each PAL, timing compilation of each PAL.
Record and present	Once data is recorded, it is presented in graphs and tables formatted through a set of webpages.

2.7. Scope of the Product

The product will test the time and space efficiency of the different PALs, through basic and proof-based tests, when provided with a proof type through the CLI. It will then provide a visual representation of the measured data through graphs and tables.

3. Constraints and Functional Requirements

3.1. Mandated Constraints

3.1.1 Solution Constraints

Description: The project will utilize the proof assistants (Lean, Idris, Agda, and Coq).
Rationale: These proof languages were chosen due to their prominence in research, and because they are all implemented in similar functional languages.
Fit Criterion: The solution must be able to translate the tests to each of these languages, and in turn analyze their time performance.

Description: The code generator and translator will be implemented in Haskell.

Rationale: Haskell was chosen for its compatibility with the four proof assistants, due to its functional programming nature.
Fit Criterion: The solution must be able to generate test cases, run and analyze proof results.

Description: The end users shall interact with the final product through a command line interface.
Rationale: The final product must be accessible, through UNIX systems.
Fit Criterion: Users must be able to select test type and receive an output of the measured space and time utilization.

3.1.2 Partner or Collaborative Applications

The project will need to interface with external proof assistants: Lean, Idris, Agda, and Coq.

3.1.3 Off-the-Shelf Software

Description: The project will need a main repository with version control, so GitHub will be used.
Rationale: Using GitHub will allow all team members to seamlessly collaborate.
Fit Criterion: Each member must be able to commit and maintain versions effectively.

Description: The project will use the Python library matplotlib to create graphs for visualization.
Rationale: Using matplotlib will allow the data collected on the efficiency of proof assistants to be visualized.
Fit Criterion: The visualization library must be easy to use and produce graphs that can be easily interpreted.

Description: The system shall use the automated testing scripts that have already been implemented in each of the proof assistants.
Rationale: Compilation and testing in each PAL is necessary for the project.
Fit Criterion: These scripts are usable and reliable with their associated proof assistants.

3.1.4 Anticipated Workplace Environment

- The system will be used in an academic and research-based setting.
- Users can work both in-person and remotely.
- Environments of such are shared by multiple researchers through collaborative efforts.

- Backend should be seamlessly installable for future research improvement.

3.2 Functional Requirements

The following functional requirements are ranked by priority from high to low.

P0 (Minimum Viable Product)

With this collection of requirements, we will be able to complete our primary task of translating a variety of proofs into the base group of four PALs. After this, it will be possible to thoroughly compare the results we obtain from each PAL and answer our original questions about comparative efficiency.

- The system will contain a hard-coded initial set of EPs written in our proof grammar (MHPG).
- The system will be given a base set of EPs of various types (propositional, natural induction/deduction).
    - By covering a broad range of test types, we will be able to determine specific instances where certain languages are inefficient in comparison with other languages.
- 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.
- The system will generate a set of tests of increasing sizes from the initial set.
- Post-test case generation, the system will translate them into the four PALs.
- The system will compile the test cases in each PAL.
- The system will record and display the compilation time of each of these test cases in the four PALs.
- The system will provide documentation on how to extend the test generator with new classes of tests.
- The system will support installation on UNIX environments.

P1

With this additional collection of requirements, more detailed results will be generated, and comparisons can be more efficiently made.

- 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.
- The system will visualize the measured data in graphs and tables on multiple webpages.
- The system will provide a description for each graph/table displayed on the webpages, explaining the data being visualized.
- The system will be available as a single downloadable package that enables users to install the complete system.
- The system will support installation on MacOS and Windows environments

P2

With this additional collection of requirements, users will have a more customizable experience and will be able to interact more with the interface.

- The system will require users to select which type of test case is generated.
- The system will only allow users to select one type of test case per generation request.
- The system will allow users to select how many versions of each test case is generated, with a lower and upper limit.
- The system will validate user input for all parameters and provide an error message if the input is invalid.
- The system will record the translated test cases in their respective PAL and store them in scripts.
- The system will record the compilation logs of each test proof and store them in a .txt file, for the purpose of supporting development (error capture), and future research.

P3

With this additional requirement, a comparison of efficiency across a broader range of proof assistants will be possible.

- The system will incorporate Isabelle as one of the PALs available for translation
    - While all four original PALs have similarities in their implementations, Isabelle is a popular proof assistant that differs more clearly from the others. Hence, this requires further research and may result in further issues with translations of the same proof being equivalent.

4. Non-Functional Requirements

4.1 Usability and Human Requirements

- The installation process should be straightforward, with minimal steps required to set up the system. A documentation should guide users through the installation, ensuring all necessary components are installed properly without requiring manual interference.
- The system should provide clear visualization of the resulting data for the time and memory complexity as we increase the size. It should provide graphs and charts for the user to easily understand the benchmarking process.
- The installation process should be divided into distinct phases, with minimal steps required for each. A documentation should guide users through each phase of the installation process, ensuring all necessary components are installed properly without requiring manual interference. Separate phases of use will be defined, especially for time-consuming components like proof assistants
- The system should be compatible with UNIX

- Any webpages produced by the system must have a layout that is understandable and found to be clear by key stakeholders, with concise explanations of the graphs and tables.
- All error messages should be clear, informative, and provide guidance to help users identify and resolve the problem.

4.2. Performance Requirements

4.2.1 Precision or Accuracy Requirements

- The translated code will have the same logical meaning as the input.
- The translation of accepted code will be able to be compiled with its associated proof assistant.

4.3. Operational and Environmental Requirements

4.3.1 Expected Physical Environment

- Our product will be primarily used in research environments, where it will be accessed on standard desktop/laptop computers.

4.3.2 Requirements for Interfacing with Adjacent Systems

- The MHPG must be translatable into all proof assistants available.
- System shall support Chromium-based browsers.

4.4. Security Requirements

- **Version Control Integrity:** While access to the GitHub repository can be open to the public for the benefit of transparency and collaboration, certain administrative actions (such as merging pull requests or modifying core functionality) should be limited to key stakeholders, including professors, teaching assistants (TAs), and authorized developers. This ensures proper oversight of contributions while still encouraging open access to the project.
- **Backup and Recovery:** Regular backups of program code must be maintained to ensure data recovery in the event of a system failure or breach. For example, team members must regularly commit their code to GitHub. In the event of a device failure, such as a laptop crash, the code can still be easily accessed from the GitHub repository, ensuring no work is lost. We will ensure that every group member commits their code consistently, allowing seamless access and recovery if local files become unavailable.

4.5. Legal Requirements

- Our software will use the open-source MIT license.

5. Risks and Predicted Issues

5.1 Risks

- As we must predetermine the base tests to work on, there is a risk of not being thorough enough and missing a case that has an unexpected result of time and memory complexity.
- Inefficiencies in how we create the translation could cause unnecessarily high times computed for certain test cases and incorrect conclusions. Potential shortcuts in languages must also be accounted for.

5.2 Predicted Issues

- We predict that it will be difficult to be certain that all the test cases are translated with exactly the same meaning across languages.