Prolog Visualisation

Rosanne Saparamadu - 300530238

***Abstract*—Logical programming languages, for instance Prolog, are powerful tools for solving complex problems through declarative programming. Comprehending and analysing such logical programs can be challenging due to Prolog's dense textual notation. This can create challenges such as understanding complex logic, and difficulty in identifying errors. To mitigate these issues, the Prolog Visualisation project aims to develop a visualisation tool that can display a Prolog clause in a graphical manner, for improved comprehension and analysis of logical programming. The objective of the project is to develop a layout algorithm that produces visualisations which express the semantics of the Prolog code being visualised. The deliverables of the project will consist of a web application, both server-side and client-side, which allows the user to enter Prolog code and generate a corresponding visualisation for it.**

I. Introduction

THE motivation behind the Prolog Visualisation project stems from the observation that Prolog programs are often easier to write than read [1]. The density of logic in Prolog programming, makes it challenging to understand the program's structure and behaviour solely through textual representation. The minimalist nature of Prolog, where essential elements are expressed concisely and predicates with multiple arguments rely on ordering alone for distinction, further adds to the challenge of understanding Prolog programs. To overcome these obstacles, the development of graphical visualisation tool is hoped to be useful to provide a visual overview of the program's structure, aiding comprehension and analysis.

Furthermore, the Prolog Visualisation project aims to tackle the difficulty in identifying errors within Prolog programs. Prolog code can be prone to errors, such as incorrect rules, which can be challenging to identify when working with textual representations alone. By providing a graphical visualisation of Prolog code, the tool can visually highlight potential errors or inconsistencies in the program's logic, enabling programmers to identify and resolve errors and inconsistencies more effectively.

Therefore, the graphical representation of the Prolog Visualisation tool will enable the programmers to better understand program structure and behaviour, and also facilitate error detection and inconsistencies. As Simon Holland states in [2], "Some aspects of Prolog programs are identified that appear to be clearer for novices when presented in the graphic formalisation." Where, graphic formalisation represents the "…equivalent to the standard textual notation for Prolog."

The Prolog Visualisation project is primarily aimed at enhancing the understanding and error detection in Prolog programming. While its direct link to some of the United Nations Sustainable Development Goals (SDGs) may not be immediately evident, it can indirectly contribute to several goals related to People, Prosperity, and Planet.

Goal 3: Good Health and Well-being (People): The Prolog Visualisation tool may not directly address health concerns but it can indirectly promote well-being. By making Prolog programming more accessible and comprehensible, it can reduce the frustration and cognitive load associated with complex logic within Prolog programs, thus contributing to the mental well-being of programmers. Clear and visually enhanced code can lead to more efficient problem-solving, ultimately reducing stress and improving the overall well-being of individuals working with Prolog.

Goal 4: Quality Education (People): The Prolog Visualisation project strongly aligns with this goal by facilitating a better understanding of Prolog code. The graphical visualisation provided by the tool can assist students and programmers in comprehending the complex logic and relations within Prolog programs. This, in turn, can enhance the educational experience and proficiency in logical programming languages.

Goal 9: Industry, Innovation, and Infrastructure (Prosperity): The Prolog Visualisation project aligns with this goal by promoting innovation in the field of programming languages. By developing a visualisation tool for Prolog, the project aims to enhance the infrastructure of logical programming and contribute to the advancement of software development practices.

Goal 12: Responsible Consumption and Production (Prosperity & Planet): The Prolog Visualisation project can indirectly promote responsible consumption and production. By making Prolog programming more accessible and understandable, it can help programmers write more efficient and error-free code. This efficiency can lead to less wastage of computational resources, contributing to responsible consumption in the digital realm.

It's important to note that not all projects directly address or align with all sustainability goals. In the case of the Prolog Visualisation project, its primary focus is on education, productivity, and innovation. This means it may not have a direct or significant impact on goals related to environmental conservation or poverty eradication. However, it can indirectly support these broader sustainability goals by fostering an environment of education and innovation that can lead to the development of more sustainable solutions in the future. In this way, the project plays a part in contributing to the broader sustainable development agenda, even if it is not its primary focus.

This project was supervised by Thomas Kühne (primary).

II. Final product and key findings of the evaluation

In the development of the Prolog Visualisation project, a set of key aspects and requirements, including both functional and non-functional, were initially outlined in the preliminary report. This served as a foundational guide for the development process. While significant progress has been made in creating the final product, it is essential to evaluate which key aspects and requirements were successfully implemented and, conversely, understand why some aspects were modified or not met.

A. Key Aspects and Requirements Met in the Final Product:

Prolog Code Input (Functional Requirement): The final product successfully provides a text input area where users can enter Prolog code. This feature allows users to conveniently enter their Prolog code for visualisation.

Prolog Code Extraction (Functional Requirement): The system accurately parses the entered Prolog code to extract its structure, including rules, facts, conjunctions, and relationships. This extraction is fundamental to the subsequent visualisation process.

Prolog Code Parsing and Abstract Syntax Tree (AST) Generation (Functional Requirement): The system constructs an Abstract Syntax Tree (AST) representation of the parsed Prolog code. This AST accurately captures the structure of the Prolog code, aiding in the visualisation process.

Visualisation (Functional Requirement): The final product utilises the constructed AST to render a visual representation of the Prolog code by incorporating nodes and edges. Although the final product doesn't include labels for the edges as depicted in the example models, it still manages to successfully convey the structure of the Prolog code. This aspect remains in harmony with the project's motivation, as it provides users with a valuable tool to visualise and understand the logic within their entered Prolog code.

Performance (Non-Functional Requirement): The final product exhibits efficient performance in terms of parsing, generating graphical notation, rendering, and responding to user interactions. The system's performance ensures a smooth user experience, with quick response times.

Reliability (Non-Functional Requirement): Despite running locally without deployment, the system has been designed to be reliable and stable, ensuring a seamless user experience with minimal errors and downtime.

Compatibility (Non-Functional Requirement): The final product remains compatible with modern web browsers, ensuring accessibility across different platforms.

B. Key Aspects and Requirements Not Met in the Final Product:

Layout Design (Functional Requirement): The final product deviated from the initial layout design plan, opting for a more user-centric approach. Instead of a fixed layout strategy, users are given the flexibility to manually arrange nodes to create their preferred layout. This adjustment was prompted by the complexity of designing a one-size-fits-all layout strategy for various Prolog code inputs. While it represents a shift from the original plan, this approach empowers users to customise the visualisation to their individual preferences and requirements.

Deployment (Non-Functional Requirement): The final product has not been deployed to a web server and still runs on localhost. Time constraints and other university course priorities led to this aspect being postponed. However, future deployment remains a possibility to make the tool accessible to a wider user audience.

Usability (Non-Functional Requirement): The user interface allows for straightforward input of Prolog code, and the interactive feature that enables users to manually adjust node positions enhances usability. However, the final product's graphical notation could be further improved to enhance usability. While the current design effectively conveys the structure of Prolog code, the addition of edge labels, similar to those found in the example models, could make the visualisation even more meaningful. This enhancement would contribute to a better understanding of the Prolog code structure and further improve the user experience.

C. Key Findings of the Evaluation with Relevant Performance Metrics:

The evaluation of the Prolog Visualisation web application is grounded in empirical and analytical evidence by using the following performance metrics:

1) **Comparison of Model Output to Actual Output:** This metric revealed distinctions in the labelling of nodes and edges between the model and actual outputs. The presence of edge labels in the model output was a significant technical aspect that enhanced the clarity of visual representations. The representation of head and tail variable nodes and the choice of shapes for list nodes were also identified as technical considerations affecting the application's design.
2) **Time Taken for Parsing and Visualising:** The recorded parsing and visualisation times demonstrated the application's parsing efficiency and the responsiveness of visual rendering. Shorter parsing and visualisation times indicated enhanced efficiency, which is an important technical aspect. These recorded times played a pivotal role in evaluating the application's user experience and technical performance.

III. Background Research

In the development of the Prolog Visualisation project, it is imperative to consider a broad spectrum of research areas. This background research section not only delves into the literature concerning user interface (UI) design principles and prototyping methods but also evaluates existing solutions that have been developed to tackle the complexities of visualising Prolog code.

A. *User Interface Design Principles and Prototyping Methods*

In the book “Designing Interfaces: Patterns for Effective Interaction Design”, the author Jennifer Tidwell states that Button Groups play an important role in enhancing the clarity and usability of an interface. They contribute to making the interface self-explanatory by organising buttons into distinct clusters that are easily distinguishable within a complex layout. As for the arrangement, the article states that Button Groups can either be aligned in a single column or placed in a row if they are not excessively wide. Hence, by incorporating Button Groups, a visual hierarchy of actions is created, allowing users to identify related and important functionalities [3].

In the context of the Prolog Visualisation web application, Button Groups can effectively display the buttons labelled “Clear” and “Visualise”, as these buttons are relevant to the text input area, in terms of clearing the code entered into the text input area or visualising the code entered into it. Also, by placing these two buttons below the text input area in a row arrangement with a reasonable gap between them, helps prevent inadvertent clicks and ensures a user-friendly experience.

According to this book, utilising an input prompt can also cleverly provide assistance in cases where the purpose of a text input area may not be immediately evident to the user. By placing the input prompt directly within the text area where the user will type, the input prompt becomes unmissable and advantageous. This approach eliminates the need for users to speculate about the purpose of the text input area or labels, as the input prompt itself conveys the necessary information [4]. The article also states that “The prompt must be put back when the user erases the value, and the requested information must be familiar to the user (such as name or email)” [4], [p. 369].

The text input area within the Prolog Visualisation web application, designed for receiving Prolog code, can effectively employ input prompts. By incorporating an input prompt such as “Enter you prolog clause here…” directly within the text input area, users are explicitly informed that this input area is intended for entering Prolog code. This intuitive approach ensures clarity and encourages users to input the appropriate Prolog code with ease.

Another book called “User Interface Design and Evaluation” discusses on the significance of protypes in the design process, describing them as experimental and often incomplete designs. Prototypes work in two ways depending on the stage of the design process. In the early stages, prototypes can facilitate communication and idea-sharing between UI (User Interface) designers, users, and stakeholders, aiding in the clarification of requirements. In the later stages, prototypes are valuable for exploring and demonstrating interactions and ensuring design consistency [5].

The book also discusses that prototypes come in two types: low-fidelity and high-fidelity. Low-fidelity protypes are characterised by their basic and simplified representations such as sketching or screen mockups. While low-fidelity prototypes provide users with an indication of the overall look and feel of the UI, they offer limited details regarding the functionality of the UI. In contrast, high-fidelity prototypes are based on software which means users are able to experience on the overall look and feel of the UI [5].

For the development of the Prolog Visualisation web application, low-fidelity screen mockups will be utilised to design the UI. This choice is made because high-fidelity prototypes, which closely represents the final product, are not suitable for requirements gathering. Also, high-fidelity prototypes are less flexible during testing and changes are more difficult to implement compared to low-fidelity prototypes, which allow for easier modifications and iterations.

B. *Existing Solutions*

The ”Prolog Visualization System Using Logichart Diagrams” by Yoshihiro Adachi offers a comprehensive solution aimed at supporting Prolog programming education. The paper presents a Prolog visualisation system that employs Logichart diagrams to make Prolog programs more accessible and understandable. This system addresses the need to facilitate the teaching and learning of Prolog, which can be challenging for beginners due to its unique mechanisms, including powerful pattern matching and automatic backtracking [6]. While these mechanisms are powerful, they can also introduce complexities, especially for those transitioning from procedural programming languages like C and BASIC.

The core of the solution involves Logichart diagrams, which are designed to provide a clear and intuitive visualisation of Prolog program execution flow. This representation serves two primary purposes. First, it visually traces Prolog execution as seen in Fig 1 below, highlighting goal calling, success, and failure. Second, it offers real-time visualisation of dynamic changes in a Prolog program, particularly when extra-logical predicates like ’assertz’ and ’retract’ are invoked. Additionally, the system provides insight into variable substitution processes through real-time display within a text widget.

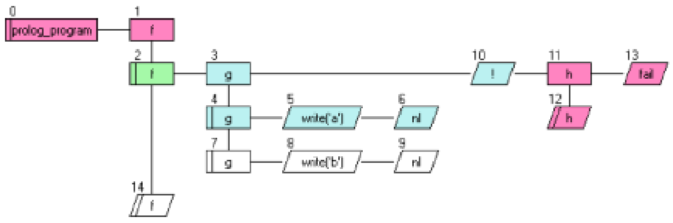

Fig. 1. Visual trace of Prolog execution.

Adachi’s ”Prolog Visualization System’s” relevance to that of the Prolog Visualisation project introduced in the current paper, lies in its shared objective of enhancing the understanding and debugging of Prolog programs. Both projects aim to provide users with tools to interact with Prolog code more effectively. Adachi’s system accomplishes this goal by offering visual representations that make it easier to comprehend Prolog’s unique execution flow and the impact of dynamic changes [6]. Similarly, the Prolog Visualisation project of this paper endeavours to provide a user-friendly tool to visualise and understand Prolog code. Both solutions target the domain of Prolog education and understanding.

The advantages of Adachi's "Prolog Visualization System" are evident in its ability to create Logichart diagrams, which provide a visual structure that is more closely aligned with the structure of Prolog code itself. This visual correspondence simplifies the process of understanding Prolog programs. Additionally, the system's real-time visualisation of execution and dynamic changes offers valuable insights into the behaviour of Prolog programs, further aiding in understanding and debugging. However, it is essential to note a potential disadvantage indicated in the paper. The system's current challenge is visualising and navigating large execution trees. As mentioned in [6], ongoing work aims to address this issue. While this may represent a limitation for dealing with complex and extensive Prolog programs, it's a limitation that is acknowledged and under active development.

In summary, the "Prolog Visualization System Using Logichart Diagrams" by Yoshihiro Adachi presents a valuable approach to visualising Prolog programs, closely aligning with the goals of the Prolog Visualisation project discussed in the current paper. The system's use of Logichart diagrams, real-time visualisation, and support for understanding Prolog execution flow makes it a noteworthy reference in the context of Prolog education and visualisation [6]."

In contrast to existing solutions, the Prolog Visualisation project offers a fresh and user-centric approach to addressing the challenges associated with understanding and debugging Prolog code. While solutions like the "Prolog Visualization System Using Logichart Diagrams" by Yoshihiro Adachi primarily focus on visualising Prolog program execution flow through Logichart diagrams, the Prolog Visualisation project takes a step further by providing programmers with an intuitive and interactive graphical representation of their Prolog code. Not only does it visualise program structure, but it also emphasises error detection and highlights potential inconsistencies, aiding in effective debugging. The Prolog Visualisation tool serves as a practical and efficient means for programmers to grasp the intricacies of their code, offering a more holistic understanding of Prolog logic. By meeting a set of key functional and non-functional requirements, the Prolog Visualisation project ensures seamless Prolog code input, accurate parsing and Abstract Syntax Tree generation, efficient performance, reliability, and compatibility across modern web browsers. This user-centric and feature-rich approach sets the Prolog Visualisation project apart, empowering programmers with a comprehensive tool for Prolog comprehension and error detection.

IV. Tools and Methodology

In the development of the Prolog Visualisation web application, a carefully chosen set of programming languages, software libraries, frameworks, and development tools played a critical role in shaping the design and functionality of the final system. Each tool brought distinct advantages to the project.

A. Programming Languages

JavaScript: JavaScript served as the primary programming language for the front-end development of the web application. Its versatility and wide adoption in web development made it an ideal choice. JavaScript allowed for the creation of interactive user interfaces and seamless communication with the back-end, enabling real-time updates and dynamic visualisations.

B. Software Libraries and Frameworks

Node.js [7]: Node.js, the runtime environment, formed the backbone of the back-end development. It greatly benefitted the project by facilitating non-blocking, event-driven architecture. This ensured efficient handling of tasks such as Prolog code parsing and Abstract Syntax Tree (AST) generation while maintaining smooth interactions with the front-end components.

Express.js [8]: Express.js, a web application framework for Node.js, streamlined the development of server-side components. It provided essential features for routing, middleware, and server-related functions. This accelerated the creation of a robust and responsive web application.

Tau Prolog [9]: Tau Prolog, a Prolog interpreter written in JavaScript, proved invaluable as a pre-processor. It prepared the entered Prolog code for parsing and AST generation by removing any whitespaces, enhancing the system's performance and compatibility.

PEG.js [10]: PEG.js, a powerful parser generator for JavaScript, played a critical role in creating a custom Prolog code parser. Its ability to define a specific grammar for Prolog facilitated precise code parsing and syntax analysis. This ensured that the visual representation accurately reflected the structure and logic of the Prolog code.

D3.js [11]: D3.js, renowned for its data visualisation capabilities, was chosen for visualising the AST graph generated from the entered Prolog code. Its flexibility and extensive features empowered the web application to present the entered Prolog code's structure and logic in a visually appealing and informative manner.

C. Development Tools

Visual Studio Code [12]: Visual Studio Code, as the integrated development environment (IDE), played a significant role in enhancing productivity during the development process. Its support for Node.js web application deployment and a wealth of extensions and plugins streamlined the coding, debugging, and project management, ensuring code quality and efficiency.

Git [13]: Git, the chosen version control system, enabled systematic tracking of project changes and management of different code versions. This ensured code stability throughout the development cycle. Also, its ability to create branches, merge changes, and maintain a clean version history ensured project integrity and helped manage different aspects of the application effectively.

The choice of this toolset supported an iterative development approach, allowing for structured progress, regular feedback, and adjustments as required. This methodology, along with open communication and guidance from the project supervisor, ensured that the Prolog Visualisation web application aligned with defined requirements and maintained a high standard of functionality and performance. The seamless integration of these tools, each carefully selected for its specific advantages, resulted in a final product with a web application designed to provide users with a powerful tool for understanding, analysing, and visualising Prolog code.

V. Design and Implementation

A. Conceptual Design

1) **System Architecture for the Final Product:** The Prolog Visualisation web application adheres to a client-server architecture, retaining its structure outlined from the preliminary report. On the client side, the application is presented through a modern and interactive web interface. This interface enables users to engage with the application by allowing them to input Prolog code and customise the visual presentation of their parsed code as preferred. The server side, which functions as the backend, remains responsible for carrying out the following processes such as parsing the entered Prolog code and visualising the nodes and edges graph.
2) **Components and Modules for the Final Product:**
    a) **Front-end UI (User Interface):** The front-end of the web application is an important component. It offers a user-friendly and responsive interface which was designed using a combination of HTML, CSS, and JavaScript. The UI encompasses a text input area where users can easily enter their Prolog code. The visual representation area is now dynamic and enables users to move nodes themselves, ensuring a layout that aligns with their understanding. This flexibility allows users with the ability to create layouts that best suit their needs, enhancing usability and user satisfaction.
    b) **Back-end Processing:** The backend component is responsible for handling the processing and analysis of Prolog code to generate visualisations. The detailed processes have been refined and enhanced for the final product.
        - **Pre-processing of Prolog Code:** The Prolog code entered by the user undergoes pre-processing to extract its structure and essential elements such as Prolog rules. This is achieved through the utilisation of Tau Prolog, which acts as a pre-processor to extract rules and also remove any whitespaces from the entered Prolog code. This pre-processed Prolog code is then handed over to the subsequent components for further processing.
        - **Prolog Code Parsing and Abstract Syntax Tree (AST) Generation:** Upon extraction of the rules using Tau Prolog as a pre-processor, each rule is transferred to a parser built with PEG.js. This parser leverages a customised grammar file to meticulously parse each preprocessed rule, resulting in the generation of the Abstract Syntax Tree (AST). This generated AST is a concise yet comprehensive representation of the entered Prolog code's structure.
        - **AST to Graph Conversion:** Next, the AST, a structural representation of the parsed Prolog code, is utilised to extract nodes and edges. In this context, the extracted nodes represent different elements of the Prolog code, including predicates, arguments, and variables, while the edges represent the connection between these nodes. These extracted nodes and edges will then contribute to the generation of the graph visualisation.
        - **Graph Visualisation:** This component leverages the D3.js library for rendering a graphical representation of the nodes and edges that have been extracted. Hence, this graph will portray various nodes using diverse shapes, where predicates are depicted as squares, and other nodes such as variables and arguments are represented as circles. Furthermore, each node contains labels which display the names of the associated predicates, arguments, and variables for easier identification and understanding. The graph also visually represents the edges as lines, helping users grasp the structure, relationships, and logic within the entered Prolog code.
        - **Layout Design:** In contrast to the preliminary report, the final product does not rely on a fixed layout design strategy that works for various entered Prolog code. Instead, it allows users to interact with the graph by allowing them to reposition nodes, affording them the flexibility to arrange nodes and configure layouts in accordance with their individual preferences. This user-driven approach to layout design in the final product encourages an exploratory perspective as it enables users to customise the graph visualisation to suit their unique requirements.
3) **Interfaces with External Systems for the Final Product:** In the final product, as with the preliminary report, Prolog interpreters and parsers are used within the Prolog Visualisation web application itself. They play an important role in handling tasks such as extracting rules from the entered Prolog code and parsing it to generate the AST graph. Therefore, there are no interfaces with external systems in the context of the Prolog Visualisation project.
4) **Requirements and Constraints Impact on Design Choices for the Final Product:** In the development of the Prolog Visualisation web application for the final product, the defined requirements and constraints have played a significant role in shaping design choices.

These requirements and constraints guided the development process, ensuring that the web application met its intended objectives and delivered a seamless user experience.

- **Functional Requirements:** The defined functional requirements heavily influenced the design choices in the final product. Due to the complexity of implementing the one-size-fits-all layout strategy detailed in the preliminary report, the decision was made to adopt a user-driven layout, which provides users the flexibility to define their graph layouts. This adaptable design approach aligns closely with the requirement for the graphical notation to display the structure of entered Prolog code meaningfully. On top of enabling users to take control of the visual layout, the design still accommodates to diverse Prolog input codes, making the tool more versatile.
- **Non-Functional Requirements:** The final product continues to excel in terms of efficiency, reliability, and user-friendliness. The system's performance remains a priority, ensuring the rapid parsing, generation, and visualisation of Prolog code while also offering smooth interactions with minimal latency. The reliability and stability of the application have been meticulously maintained, creating a seamless user experience with minimal errors or downtime. The user interface is not only visually appealing but now more adaptable, allowing users to dictate their layout preferences, thus further enhancing its intuitiveness. These non-functional requirements have steered the design choices, resulting in a web application that promises a high standard of functionality and performance.
- **Compatibility Constraint:** The requirement for compatibility with various browsers and devices significantly affected design decisions. The responsive design of the user interface and the adoption of web standards that account for varying browser behaviours and device specifications ensure that the application provides a consistent and user-friendly experience across different platforms and screen sizes. These choices result in a web application that is accessible and usable on a wide range of devices and web browsers, addressing the compatibility constraint effectively.

The final design choices of the Prolog Visualisation web application have been heavily influenced by the requirements and constraints outlined. They were important in ensuring that the application was not only functional and efficient but also adaptable to different user preferences and capable of providing a consistent, reliable, and user-friendly experience across diverse platforms and devices.

B. Sustainability Considerations:

This section delves into the considerations of sustainability in the design of the Prolog Visualisation web application. It assesses how the project addresses environmental, social, economic, and technical sustainability aspects in its design.

1) **Environmental Sustainability:** The Prolog Visualisation web application, by its design and nature, doesn't significantly impact environmental sustainability. This is primarily because it does not rely on cloud-based servers or involve resource-intensive computations. The web application operates on the user's local device, performing Prolog code parsing and visualisation within the user's web browser. As such, it doesn't consume excessive energy or material resources associated with cloud-based services or large-scale data centres. Hence, the project's environmental impact is minimal, considering that it is essentially a client-side application that doesn't require extensive server infrastructure.
2) **Social Sustainability:** Regarding social sustainability, particularly in terms of privacy and equity, the Prolog Visualisation web application follows a privacy-conscious approach. It does not collect, store, or transmit any user data. The only input it receives is the Prolog code entered by users, which is inherently non-sensitive information. As a result, there are no privacy issues or concerns related to individuals, communities, or societies associated with the project. Furthermore, the web application is designed to be openly accessible to all users without bias or discrimination, promoting equity in its accessibility and usage. It provides a free and inclusive tool for anyone interested in visualising Prolog code, thus aligning with principles of social sustainability.
3) **Economic Sustainability:** The Prolog Visualisation project has been developed with an awareness of both short-term and long-term economic requirements. The project's open-source nature and utilisation of commonly available web technologies help ensure that it remains cost-effective for both developers and users. It doesn't require expensive software licenses or proprietary systems, which could otherwise present economic challenges. Additionally, the absence of cloud-based servers or other resource-intensive infrastructure keeps operating costs low. Long-term sustainability is also supported by its simplicity and avoidance of resource-intensive technologies, making maintenance more affordable and feasible.
4) **Technical Sustainability:** The technical sustainability of the Prolog Visualisation web application is underpinned by its open-source nature and sensible technology choices. The project is open source, with its complete source code available in a public repository, ensuring transparency and accessibility for developers and contributors. This open nature supports technical sustainability by allowing for potential future deployments or extensions. Additionally, the project relies on well-established and widely used technologies such as HTML, CSS, JavaScript, and libraries like D3.js. These choices enhance its longevity, as these technologies are expected to remain supported and compatible, contribut-

ing to the project's ongoing technical sustainability. The absence of complex or proprietary technologies also simplifies maintenance and adaptation, making it easier for future developers to work with the codebase.

The Prolog Visualisation web application demonstrates a commitment to sustainability in multiple aspects. While its impact on environmental and social sustainability is minimal due to its client-side nature and privacy-conscious design, it aligns with economic sustainability by being cost-effective and upholds technical sustainability through its open-source architecture and technology choices. These considerations collectively position the project as a responsible and sustainable solution for visualising Prolog code.

C. *Implementation*

VI. Evaluation

A. *Performance Metrics*

The performance evaluation of the Prolog Visualisation web application involves a comprehensive comparison of the model output with the actual output for two example Prolog code snippets, 'transform.pl' and 'sift.pl'. These comparisons are significant in assessing the quality, accuracy, and effectiveness of the tool. They also involve recording the time taken for parsing and visualising the results, offering insights into the efficiency of the application.

1) **Comparison of Model Output to Actual Output:**
   a) **transform.pl:** I initially rearranged the nodes and edges within the Prolog Visualisation web application to match the layout of the model output as seen in Fig 4. I successfully achieved a layout similar to the model output as portrayed in Fig 5 below, but I observed several distinctions between the two outputs.

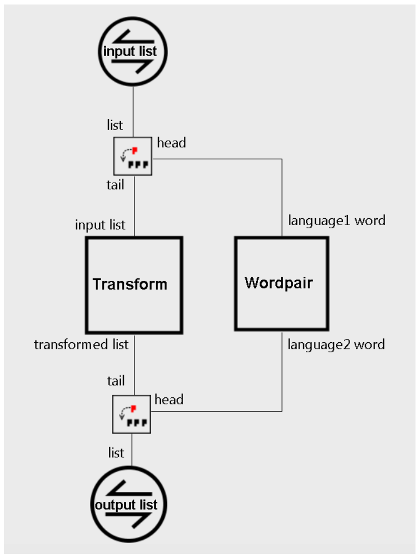

Fig. 2. Model output for transform.pl.

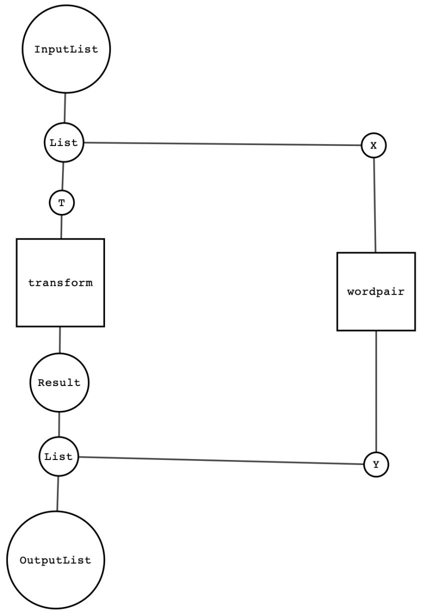

Fig. 3. Actual output for transform.pl.

i) The model output features labels for both nodes and edges, while the actual output lacks labels for the edges. These edge labels in the model output provide users with additional information, indicating whether a connection involves the 'head' or 'tail' of a list node.

ii) In the actual output, the head and tail variable nodes of list nodes are displayed on the graph. This is not the case in the model output. For instance, in the actual output, nodes labelled 'X,' 'Y,' 'T,' and 'Result' represent the heads and tails of list nodes, which are absent in the model output. Instead in the model output, these variables are labelled 'head' (represented by node 'T' in the actual output), 'tail' (represented by node 'Result' in the actual output), 'language 1 word' (represented by node 'X' in the actual output), and 'language 2 word' (represented by node 'Y' in the actual output).

iii) Furthermore, the list nodes in the model output are depicted in a square shape to distinguish them as a different type compared to other nodes. In contrast, the actual output represents list nodes in a circular shape, similar to other nodes in the entered Prolog code, such as variables or arguments."

   b) **sift.pl:** Similar to when I evaluated transform.pl previously, I initially rearranged the nodes and edges within the Prolog Visualisation web application to match the layout of the model output as seen in Fig 6. I successfully achieved a layout similar to the model output as portrayed in Fig 7, but I observed several distinctions between the two outputs.

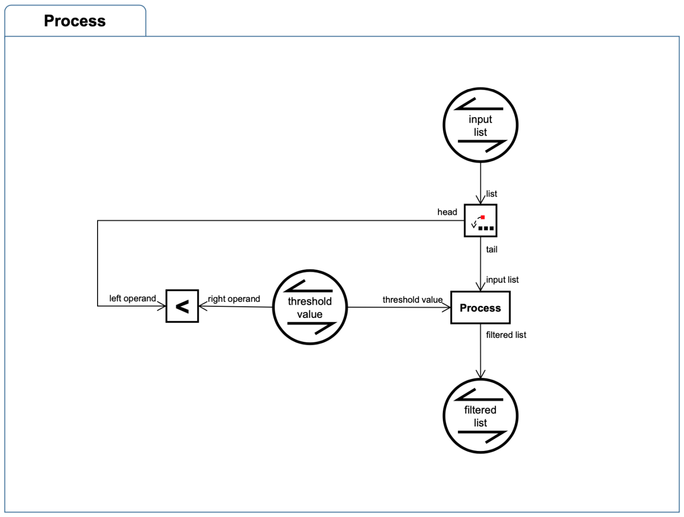

Fig. 4. Model output for sift.pl.

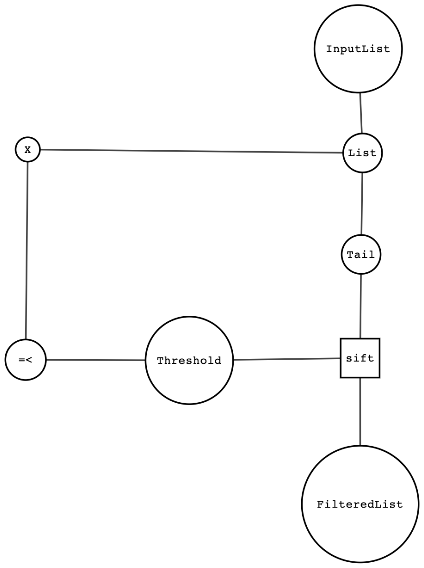

Fig. 5. Actual output for sift.pl.

i) The model output includes labels for nodes, just as the actual output does. However, the model output goes a step further by providing labels for edges, which are absent in the actual output. These edge labels provide users with additional information, indicating whether a connection involves the 'head' or 'tail' of a list node connecting to another node. Furthermore, the model output provides specific edge labels, such as 'input list' indicating that the tail of the list node serves as input for the 'sift/process' node and 'filtered list' indicating the output of the 'sift/process' node. Additionally, the model output visually represents the 'left operand' edge entering the "$<$" condition node from the left, with an arrowhead pointing left, and similarly, the 'right operand' edge entering from the right with an arrowhead pointing to the right side of the condition node.

ii) Once again, the actual output depicts the head and tail variable nodes of list nodes on the graph, which is not found in the model output. This contrast arises because in the model output, these variables are represented as edges. For instance, in the actual output, the node labelled 'X' corresponds to the edge labelled 'head' in the model output and the 'Tail' node corresponds to the edge labelled 'tail' in the model output.

iii) Furthermore, list nodes in the model output are portrayed in a square shape to distinguish them as a distinct type compared to other nodes. Conversely, in the actual output, they are represented in a circular shape, similar to other nodes in the entered Prolog code, such as variables or arguments."

c) **Evaluation Significance:** The comparison between the model output and the actual output holds significant importance in assessing the quality and accuracy of the Prolog Visualisation web application. This performance metric plays a crucial role in evaluating the tool's effectiveness in rendering Prolog code visually. It allows for identifying both commonalities and differences between the two outputs, which are essential in the following ways:

- **Quality Assessment:** Comparing the model and actual outputs offers a means of evaluating the tool's correctness in visualising Prolog code. It provides a basis to assess whether the visual representation accurately captures the structural elements, relationships, and logic of the entered code. Discrepancies between the two outputs can indicate areas for improvement and enhancement.
- **User Experience:** The presence or absence of labels for both nodes and edges in the model and actual outputs significantly impacts user experience. Labels enhance user understanding by providing additional information, such as identifying whether a connection involves the 'head' or 'tail' of a list node. Thus, it influences the clarity and interpretability of the visualised Prolog code.
- **Representation of Variables:** Notably, the representation of head and tail variable nodes of list nodes directly influences the completeness of the visual output. While the actual output displays these nodes, their absence in the model output may have implications for user comprehension. Therefore, this metric highlights the need to balance clarity and completeness.
- **Visual Elements:** The shape and design of

list nodes in the model and actual outputs affect the visual aesthetics of the representation. This metric highlights the significance of visual consistency and design choices, particularly in ensuring that list nodes are appropriately distinguished from other nodes.

2) **Time Taken for Parsing and Visualising:**
    a) **transform.pl:**
        - The time taken for parsing was 44.09999996423721 ms.
        - The total time taken for visualisation was 72.19999998807907 ms.

    b) **sift.pl:**
        - The time taken for parsing was 34.39999997615814 ms.
        - The total time taken for visualisation was 59.80000001192093 ms.

    c) **Evaluation Significance:** The time taken for parsing and visualising Prolog code serves as an essential metric to evaluate the efficiency and responsiveness of the Prolog Visualisation web application. The evaluation significance of this metric is as follows:
        - **Efficiency Assessment:** Recording the time taken for parsing and visualising provides insights into the efficiency of the application's core processes. The parsing time reflects the speed at which the application processes the Prolog code. A shorter parsing time indicates higher efficiency. Similarly, the visualising time represents the speed at which the application generates the visual representation. A quicker visualisation time suggests efficient rendering.
        - **User Experience:** Efficiency in parsing and visualisation directly contributes to a smooth and responsive user experience. Users expect prompt feedback when interacting with the tool. Recording these times ensures that the application meets user expectations for responsiveness. Faster processing times enhance user satisfaction and usability.
        - **Optimisation:** Understanding the time required for parsing and visualisation helps identify potential bottlenecks in the application. It guides development in optimising the tool's performance, in this case, by addressing areas where processing speed can be improved. This, in turn, results in a more user-friendly and efficient application.

In summary, the comparison of model output to actual output and the recording of parsing and visualisation times serve as comprehensive performance metrics. They provide a thorough evaluation of the Prolog Visualisation web application's correctness, efficiency, and user-friendliness. The identified differences and timings will help guide improvements and optimisations to enhance the tool's performance and effectiveness.

B. *Results: Empirical and Analytical Assessment*

1) **Comparison of Model Output to Actual Output:**
In the comparison of the model output to the actual output, the Prolog Visualisation web application was assessed for its correctness, accuracy, and effectiveness in rendering Prolog code visually. This evaluation provided significant insights into the technical aspects of the tool.
The examination of 'transform.pl' revealed several distinctions between the model output and the actual output. Notably, while the model output included labels for both nodes and edges, the actual output lacked labels for edges. This technical aspect influenced user experience and interpretability. The presence of edge labels in the model output provided users with additional information, improving the clarity of the visualised code.
Furthermore, the assessment uncovered differences in the representation of head and tail variable nodes of list nodes. In the actual output, these nodes were displayed on the graph, but in the model output, they were represented as edges. This distinction highlighted the need to balance visual completeness with user understanding, emphasising a technical choice in the application's design.
The representation of list nodes, displayed as squares in the model output and circular shapes in the actual output is also another distinction. It highlighted the need to balance accuracy for visualisations between the two outputs.
2) **Time Taken for Parsing and Visualising:** The empirical results obtained from recording the time taken for parsing and visualising entered Prolog code provided an analytical assessment of the application's efficiency and responsiveness. These technical perspectives played a critical role in understanding the tool's performance.
The parsing time, which was 44.1 ms for 'transform.pl' and 34.4 ms for 'sift.pl,' reflected the speed at which the application processed the Prolog code. The parsing process is a fundamental technical aspect of the application's functionality, and shorter parsing times indicate higher efficiency in this core process.
The total time taken for visualisation was 72.2 ms for 'transform.pl' and 59.8 ms for 'sift.pl.' This time accounted for the speed at which the application generated the visual representation of the Prolog code. Visualising time is another critical technical perspective that influences the user experience. Quicker visualisation times enhance user satisfaction and usability, demonstrating the application's responsiveness.
The evaluation of these technical aspects also highlighted the efficiency and optimisation of the Prolog Visualisation tool. Understanding the time required for parsing and visualisation allow for the identification of potential areas for improvement and performance optimisation.

In summary, the empirical and analytical results of the Prolog Visualisation web application, drawn from the comparison of model output to actual output and the time taken for parsing and visualising entered Prolog code, provide a holistic assessment of the solution's technical performance. These results shed light on the correctness, user experience, efficiency, and optimisation of the tool, offering a view of its technical aspects.

C. Limitations and Further Improvements

This section recognises the limitations and considers opportunities for further improvements in the final product of the Prolog Visualisation web application.

Limitations:

1) **User Collaboration:** The final product primarily caters to individual users, and the tool lacks collaborative features. Introducing user accounts within the application could enable users to save and share their visualisations with others. Implementing a dedicated section for users to showcase their visualisations and fostering interactive discussions through comment sections would promote user collaboration and community learning. The absence of such collaborative elements represents a limitation in the tool's potential for fostering shared learning experiences.
2) **Error Handling and Validation:** The final artefact of the project also exhibits limitations in terms of error handling. Robust error handling and validation mechanisms play a pivotal role in providing users with meaningful feedback and improving the overall user experience. The existing tool lacks comprehensive error handling capabilities, which can lead to user confusion in case of issues during parsing or mistakes in the Prolog code. Enhancing the error handling functionalities would address these limitations and contribute to a more user-friendly experience.
3) **Labelling Consistency:** The final product of the Prolog Visualisation web application exhibits a limitation in the consistency of labelling. While the model output provides labels for both nodes and edges, the actual output does not include edge labels. This inconsistency between the model and actual outputs can impact user experience by influencing the clarity and interpretability of the visualised Prolog code.
4) **Visual Aesthetics:** Another limitation relates to the visual aesthetics of the application. While the design of the final product aims for clear and informative representations, there is room for improvement in terms of visual appeal and customisation. Providing users with options to customise the appearance of graphs generated by the application, such as node shapes, colours, or layout styles, would enhance the tool's visual aesthetics, aligning it with individual preferences and improving user engagement.

Further Improvements:

1) **Scalability Enhancements:** Improving the scalability of the visualisation component is a priority. Implementing techniques such as zooming and panning can enhance the user experience by allowing users to navigate and explore larger graphical visualisations, even on smaller devices. This ensures that the tool remains user-friendly and functional for a wider range of Prolog code complexities.
2) **Integration with Online Prolog Resources:** To support users in their learning journey, integrating the web application with online Prolog resources would be beneficial. This involves incorporating links or references to online Prolog documentation, tutorials, and other learning materials directly within the application. Thus, users can conveniently access these resources while working on their Prolog projects, promoting continuous learning and skill development.

VII. Conclusion

In conclusion, the final product for the Prolog Visualisation web application offers a user-friendly solution for visually representing Prolog code. Through a comprehensive design and implementation process, this tool successfully provides a platform for users to interact with and understand the structural elements and logic of Prolog programs. The application's evaluation, based on performance metrics and empirical results, highlights its strengths in accuracy and efficiency, while also shedding light on areas for enhancement. Acknowledging limitations and suggesting future improvements ensures that this project remains committed to continuous development, furthering its potential as a valuable resource for Prolog learners.

References

[1] T. Kühne, "A Visual Notation for Declarative Behaviour Specification." Accessed: May. 28, 2023. [Online]. Available: https://homepages.ecs.vuw.ac.nz/ tk/publications/papers/visual-notation-kuehne.pdf
[2] S. Holland, "1992-PPIG-4th-Holland.pdf." Accessed: May. 28, 2023. [Online]. Available: https://www.ppig.org/files/1992-PPIG-4th-Holland.pdf.
[3] J. Tidwell, "Chapter 6: Doing Things: Actions and Commands," in *Designing Interfaces: Patterns for Effective Interaction Design,* O'Reilly Media, Inc, 2005, pp. 245-247.
[4] J. Tidwell, "Chapter 6: Doing Things: Actions and Commands," in *Designing Interfaces: Patterns for Effective Interaction Design,* O'Reilly Media, Inc, 2005, pp. 369.
[5] D. Stone, C. Jarrett, M. Woodroffe and S. Minocha, "Chatpter 6 — Thinking about requirements and describing them," in *User Interface Design and Evaluation,* Elsevier, 2005, pp. 114-120.
[6] Y. Adachi, "Prolog Visualization System using LogicHart Diagrams," arXiv (Cornell University), Mar. 2009, [Online]. Available: https://arxiv.org/pdf/0903.2207
[7] "Node.js," [Online]. Available: https://nodejs.org/en. [Accessed 29 May 2023].
[8] "Express - Node.js web application framework," [Online]. Available: https://expressjs.com/. [Accessed 29 May 2023].
[9] "Tau Prolog: A Prolog interpreter in JavaScript," [Online]. Available: http://tau-prolog.org/. [Accessed 29 May 2023].
[10] "Parser Generator for JavaScript," [Online]. Available: https://pegjs.org/. [Accessed 29 May 2023].
[11] "D3.js - Data-Driven Documents," [Online]. Available: https://d3js.org/. [Accessed 29 May 2023].
[12] "Visual Studio Code - Code Editing. Redefined," [Online]. Available: https://code.visualstudio.com. [Accessed 29 05 2023].
[13] "Git," [Online]. Available: https://git-scm.com/. [Accessed 29 May 2023].