University of Dublin

TRINITY COLLEGE

Visualization framework for verification tools with graph output

Tianyi Zhang

Supervisor: Dr. Vasileios Koutavas

A Dissertation

Submitted to the University of Dublin, Trinity College

In partial fulfillment of the requirements for the degree of

Master in Computer Science

April 2021

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

_____

Name

30 April 2021

Abstract

In the field of formal verification, the use of abstract data structure is very common, many of these data structures such as finite state machine can be modeled using a graph. While graph models can be very helpful in understanding the data, they can only be used alongside the support tool that produces the visualization. For both novice and expert verification tool users, it is not uncommon to be experimenting with several different verification tools. It is common for these tools to produce text-based outputs without providing its users with any visualizations to aid understanding. A web framework for showcasing such tools is proposed in this dissertation. The framework features a user-facing interface for displaying an interactive graph visualization produced by the verification tools with the ability for the user to input source code or select from sample sources. The interface also features pathfinding and layered data exploration. The verification developer is capable of easily integrate their verification tools with the backend of the framework using flexible configurations that define both the frontend and backend behaviors, integrated within a Docker container for portability giving the ability to showcase the tools on any host.

Acknowledgment

I would like to thank my supervisor Dr. Vasileios Koutavas for providing guidance throughout the course of this project. He provided advice and insights with great clarity and acted as the stakeholder to my project artefact, providing valuable feedback from a user's point of view. The completion of this project could not have been possible without his help.

I would also like to thank my family for supporting me through the difficult times of Covid-19, especially in keeping the indoor environment suitable for research and studies.

Contents

Section 1 Introduction

“Computer-based visualization systems provide visual representations of datasets intended to help people carry out some task more effectively.” [1] It is capable of transforming complex and abstract data into visual form, allowing uses to have an easier understanding and exploration of such data. Among the numerous numbers of different visualizations available today, graphs are the most common abstract structure found in computer science today for abstract information modeling. In practice, graph models can only be used alongside the support tool that produces the visualization. [2]

As the focus of this dissertation, graph visualization is applied to verification tool, where usage of data structures that can be represented by graph models are frequent. Formal verification is an aspect of formal methods that aim to prove the properties of the specified system. [3] A reasonable level of knowledge is usually required for the understanding of these verification tools due to abstracted models and complex algorithms. Most tools however are only capable of producing text-based outputs as this is the most simplistic form of output. Extra time must be invested to produce graphical output. Comparing to text-based output, graphical output is capable of improving user understanding as stated earlier. This can be very helpful in formal verification where models and algorithms can be difficult to understand even for experts of the subject area.

This dissertation presents a framework for visualizing graph outputs of such verification tools. Providing standardization to the output format, customizable configuration for generating and displaying an interactive graph visualization. The framework also serves as a collection interface for a number of such tools, making tools accessible through a single interface. The main target users of this framework are verification tool developers who would like to integrate their tool with visualization framework and researchers who would like to experiment with the underlying verification tools. It is important to note that the visualization framework is only a tool for displaying a visualization of graphs outputted by the verification tool, therefore it is not a replacement for comprehensive interaction with the original verification tool.

At the very high level, the framework involves a frontend UI for rendering and interacting graph visualization and a backend server that executes the verification tools for an output.

Although this dissertation focuses more on verification tools, in practice this framework should be capable of supporting a more general set of software that produces graph models using the defined output format and execution mode.

1.1 Motivation

In the field of formal Verification, data structures that can be represented as graphs are widely used. As an example, a common approach is the use of model checking. In this approach, the system is modeled by a finite state machine. The verification process then aims to traverse the finite state system to prove the desired property. [3] Numerous other different verification methods make use of structures such as labeled state transitions, trees, etc. Indirect usage of graph models for verification tools is also notable where formal models can be expressed as a structure that can be represented as graph models. These will be briefly talked about in Section 1.2

As stated in the introduction, most of these verification tools produce text-based outputs which would be difficult to read and understand. There are two options available in such a scenario:

1. The user converts the output to visualization using an external tool.
2. The verification tool provides the feature to produce the visualization

In option 1, different verification tools may output graph models in a different format, the user must be knowledgeable on which visualization software to use. The user will also need to have access to that tool in his/her environment. In option 2, the verification tool developer will need to invest time in producing a visualization feature. This would involve typical tasks such as finding a suitable library, incorporate it into the tool, testing, etc. This would usually take up the time that could be used to optimize the tool itself. User environment may also become a concern when choosing the visualization software to use for the tool.

A framework for visualizing the outputs of these tools becomes very useful in the above cases. The user can get direct access to the visualization and the developers do not need to consider the choices of software to use as it will be standardized with the framework. another use case for the framework would be to showcase a collection of tools. It is typical when researching that the researcher will be using a number of tools, a single point of access to these tools' visualizations can be very helpful. As another example, a professor could be showcasing a number of tools to the students, in this case, is it also very helpful to have the tools and visualizations in one place.

In the next sub-section, a number of tools that are example use cases for the framework will be briefly explored.

1.1.1 Motivation Examples - Pifra (Pi-Calculus Fresh-Register Automata)

Pifra is a tool for generating labeled transition systems (LTS) of pi-calculus models represented by fresh-register automata (FRA). Using the paradigm of fresh-register automata, the tool generates a finite LTS from pi-calculus models with possibly infinite states. Model-checking can then be performed on the generated model which lays the ground for further verification techniques such as bisimulation and modal logic assertion. [4] The tool is a typical use case for the framework described in this dissertation. It produces a direct output in the form of LTS which can be directly modeled as a directed multigraph. Figure 1 displays the output of Pifra in the DOT language. This output can be used to generate a static visualization of the graph using Graphviz.

```
s0 = {(1,#1),(2,#2)} |- (#2(&2).0 | $&1.#1'<&1>.#2'<&1>.0)
s0  2 1   s1 = {(1,#1),(2,#2)} |- $&1.#1'<&1>.#2'<&1>.0
s0  2 2   s1 = {(1,#1),(2,#2)} |- $&1.#1'<&1>.#2'<&1>.0
s0  2 3*  s1 = {(1,#1),(2,#2)} |- $&1.#1'<&1>.#2'<&1>.0
s0  1'1^  s2 = {(1,#1),(2,#2)} |- (#2'<#1>.0 | #2(&1).0)
s1  1'1^  s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  2'1   s4 = {(2,#2)} |- #2(&1).0
s2  2 1   s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  2 2   s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  2 3*  s3 = {(1,#1),(2,#2)} |- #2'<#1>.0
s2  t     s5 = {} |- 0
s3  2'1   s5 = {} |- 0
s4  2 2   s5 = {} |- 0
s4  2 1*  s5 = {} |- 0
```

Figure 1 Pifra text output in dot language

1.1.2 Motivation Examples – Symbolic-Bisim

Symbolic-bisim is a tool currently under development by Vasileios Koutavas. The tool is a verification tool that checks for equivalence of two programs in ML language. In general, the tool checks whether two programs are equivalent by verifying if one program is simulated by the other. The tool uses semantics described in [5] which uses Game semantics to model bipartite LTS. The output of the tool as shown in figure 2 represents state transitions, is difficult to read. In such a case the LTS can be visualized to provide more clarity on the output traces.

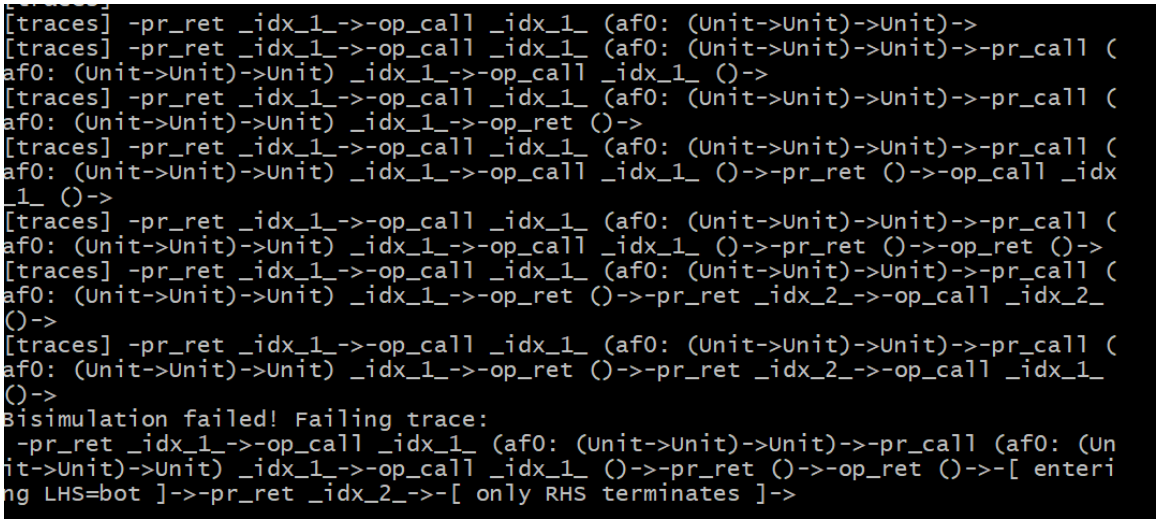

Figure 2 Symbolic-bisim output traces

1.1.3 Motivation Examples – HorSat

HorSat is a model checker for higher-order recursion schemes (HORS) based on saturation. The technical detail of the tool is described in [6]. In simple terms, HORS is a tree-generating functional program. The tool verifies if a given property is satisfied by the tree generated by HORS. The tool is heavily involved with tree structure. The inputs to the tool are the recursion scheme which generates a tree and a tree automaton describing the property to verify for. The output when the property is violated is also either a path of the tree or a subtree. Figure 3 shows an example path of the tree outputted in text when input is not satisfied. The majority of the structures in this tool can be represented by a graph visualization.

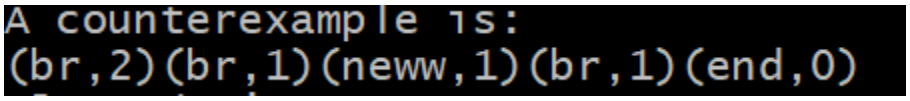

Figure 3 HorSat unsatisfied property output example

1.2 Dissertation structure

Section 2 Related work – briefly describes similar tools that are used to showcase verification tools

Section 3 Methodology – describes the software engineering process undertaken by the dissertation focusing on requirements engineering

Section 4 Implantation – walks through the implementation of the final artefact including architecture design, user interface, technology usage, visualization presentation, and configurations.

Section 5 Evaluation – focus on usability evaluation for the entire system and briefly touches on security and performance

Section 6 Future Work – describes the works that are still needed to improve on the existing artefact

Section 7 Conclusion – Summary of the contributions

Section 2 Related Work

2.1 LLRÊVE

LLRÊVE [7] is a web interface for a tool that checks the program equivalence of C programs. The web interface is shown in figure 4. It has many nice features to it. First, it has a selection of examples programs for immediate use if a user simply wants to try out the tool. The UI is simplistic, with directions on how to use the tool, this is very effective in terms of usability.

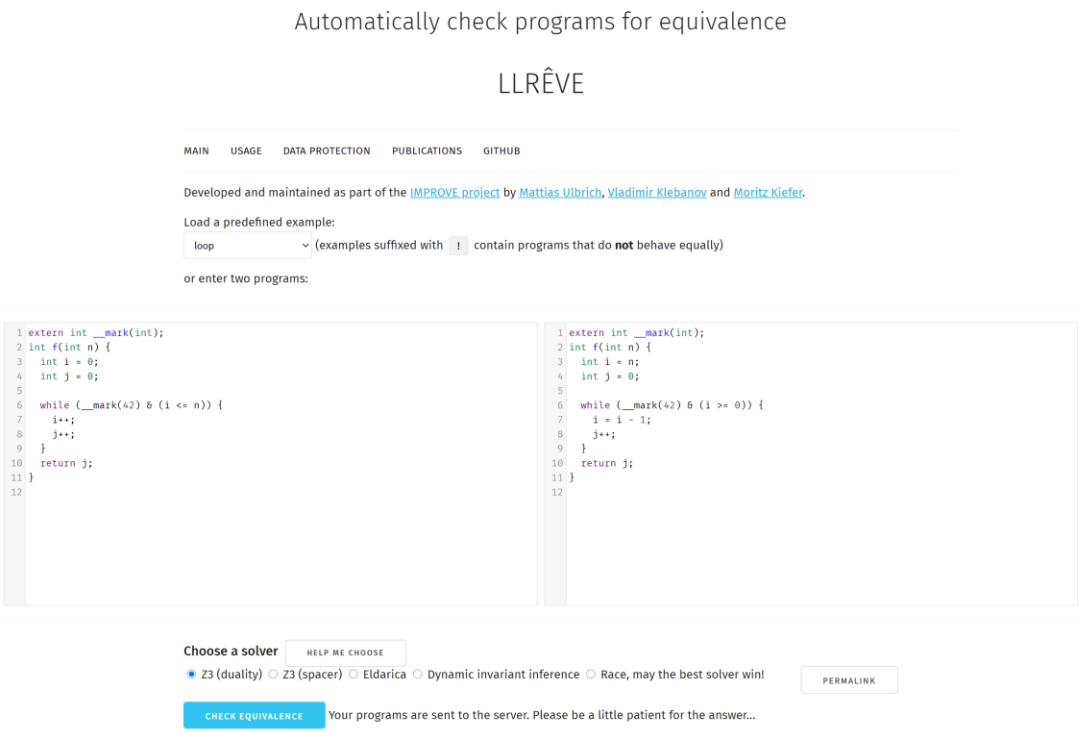

Figure 4 LLRÊVE homepage

Figure 5 displays the output LLRÊVE. The immediate information presented to the user is the result of the verification highlighted in bright color which brings users' attention to the most important information. However, the outputs of LLRÊVE are completely textual. As stated above, visualization can help with the understanding of the outputs which was not implemented by LLRÊVE. LLRÊVE is also specific to the regression-based verification tool running underneath. Therefore, the UI cannot be integrated easily with use by another tool.

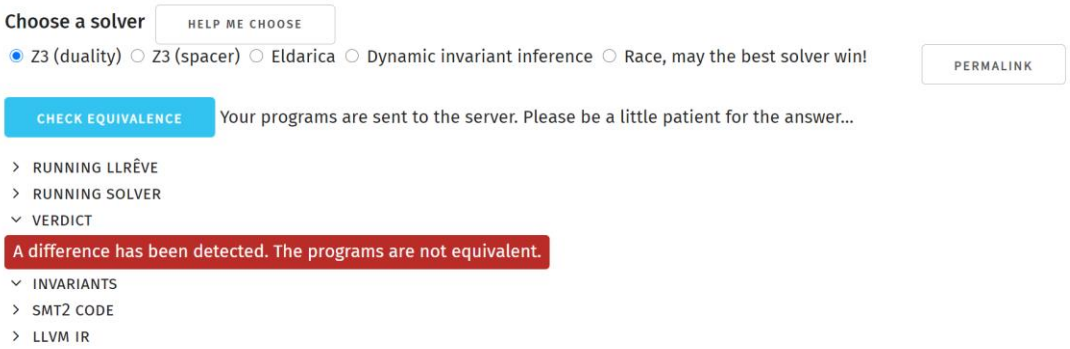

Figure 5 Output of LLRÊVE with inequivalent program

2.2 UPPAAL

UPPAAL [8] is an integrated tool environment for modeling, validation, and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). [8] UPPAAL is developed in java and distributed as an executable. Figure 6 displays the user interface example as posted on its website. It is noted that the tool provides a visualization of the real-time system in question and displays layer information using interactive buttons featured by the UI. The tool also provides comprehensive functionalities for exploring, validating, and verifying the system model. However, the tool is again very specific to the real-time system programs integrated with UPPAAL although it provides a concise and interactive representation of the real-time system in a graph model. The delivery of the software in the form of an executable program can also have the downside of the need to be download and user environment dependencies making the software less accessible than the other examples presented in this section.

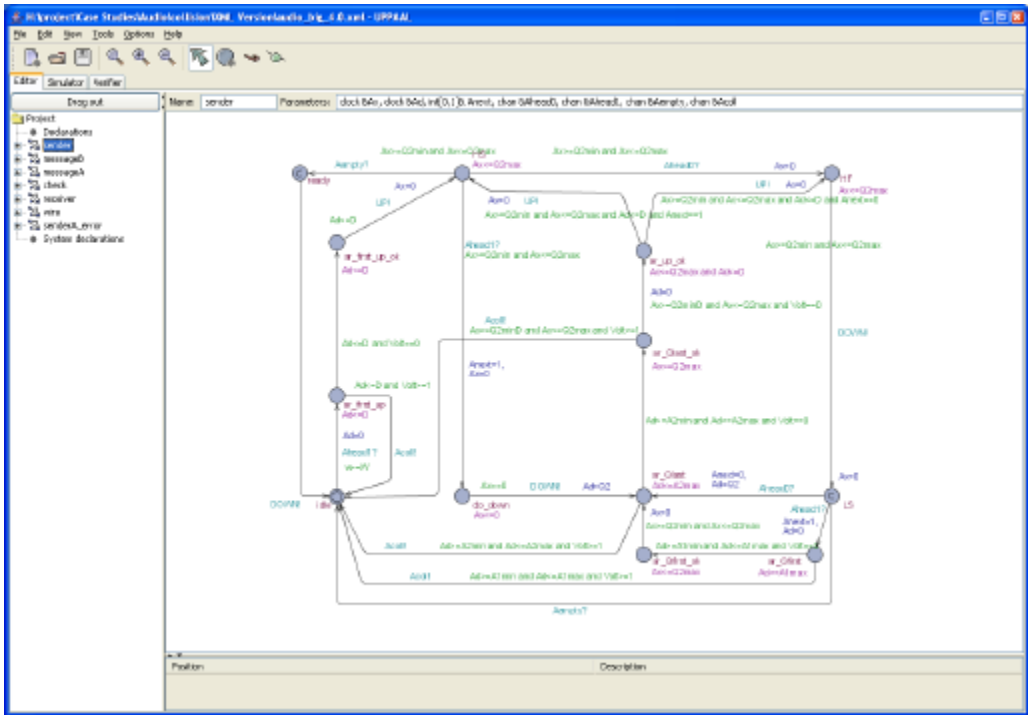

Figure 6 UPPAAL user interface

2.3 Rise4Fun

Rise4Fun [9] is a web interface for a collection of software engineering tools. Figure 7 shows the collection of tools on the Rise4Fun homepage and figure 8 shows tools pages on Rise4Fun where the main components include input box, tutorial, samples, and output. Its main benefit is the ability to allow users to experiment with different tools through a single interface which is one of the motivations of this dissertation. Tool developers can integrate their tool on Rise4Fun by setting up their own service API that communicates with Rise4Fun using defined input and output message structure. Then the tool can be accessed and displayed on Rise4Fun. Rise4Fun defines the information of the tools including tutorials, samples, descriptions, etc. in the metadata. The metadata is then used to configure the UI. It also supports defining input language syntax highlighting for a tool by defining language syntax definition in JSON format. These features make it a convenient platform for experimenting with different tools. The presence of tutorials and examples also helps users understanding the tool. The downside of Rise4Fun is that it does not provide interactive visualization support for the tools. It is dependent on the tool service API to return any visualization in a format that could be rendered by the browser. It also does not support optional parameters that would sometimes be present when executing a program from the command line. The service API per tool can also create inconsistent experiences for the user as the performance is dependent on the host of the service API.

rise4fun

a community of software engineering tools

all tutorial automata concurrency design encoders infrastructure languages security synthesis testing verification language

microsoft

agl Automatic Graph Layout

alive Optimization Verifier

bek A domain specific language for writing and analyzing common string functions

bex A domain specific language for writing and analyzing string encoders and decoders

boogie Intermediate Verification Language

civl Concurrent Intermediate Verification Language

code contracts Language agnostic modular program verification and repair with abstract interpretation.

counterdog Theorem-prover for Counterfactual Datalog

dafny A language and program verifier for functional correctness

esm Empirical Software Engineering and Measurement Group

formula Formal Modeling Using Logic Programming and Analysis

formula2 Formal Modeling Using Logic Programming and Analysis

f* A verification tool for higher-order stateful programs

heapdbg Runtime heap abstraction

iz3 Efficient Interpolating Theorem Prover

pex Automatic test generation using Dynamic Symbolic Execution for .NET

quickcode Progamming-by-example technology for learning string transformation programs

concurrent revisions Parallel and Concurrent Programming With Snapshots

rex Regular Expression Exploration

seal Side-Effects AnalysiS

touchdevelop Program your phone on your phone.

vcc A Verifier for Concurrent C

visual c++ VC++ compiler (Latest front-end build)

z3 Efficient Theorem Prover

multicore programming group, imperial college london

gpuverify-cuda A verifier for CUDA/OpenCL kernels

gpuverify-opencl A verifier for CUDA/OpenCL kernels

nova-lincs@fct/unl

dift Dependent Information Flow Types: Typechecker

otto-von-guericke-university magdeburg - department of distributed systems

Figure 7 Rise4Fun homepage

iz3 Microsoft Research

Do these formulas have an interpolant?

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (declare-const z Int)
4 (compute-interpolant
5    (= y (* 2 x))
6    (= y (+ (* 2 z) 1)))
7
```

▶ tutorial home video permalink

'▶' shortcut: Alt+B

```
unsat
(<= 0 (+ (* (- 1) y) (* 2 (div y 2))))
```

samples
even_odd
simple
quantifier
sequence
incremental
tree
two_store
equality

about iZ3 - Efficient Interpolating Theorem Prover

iZ3 is a high-performance interpolating theorem prover. iZ3 supports arithmetic, arrays, uninterpreted functions, and quantifiers.

Figure 8 iz3 tool page on Rise4Fun

Section 3 Methodology

3.1 Engineering Process

There are many software engineering frameworks and methodologies out there as described in [10]. The framework adopted by this project is the agile framework, specifically the rapid application development (RAD) methodology. RAD, being a member of the agile family, employs a short and iterative development life cycle. RAD specifically features rapid iterations of development with a short planning period. It focuses on receiving customer feedback and producing rapid prototypes. A typical RAD consists of three main phases. The requirements planning phase, the user design phase, and the construction phase [11]. The requirement phase involves requirement elicitation which will be discussed later. The user design phase involves developing prototypes and analyze the prototype with users. This phase is iterative with constant change in prototypes as requirements changes. The construction phase involves the implementation of the final product. Although user interference is still valid in this phase. This methodology was suitable for this project for two specific reasons. First, the project has a key focus on usability of the framework both for verification tool users, and verification tool developers. Due to the customer-facing nature of the project, RAD enables consistent feedback retrieval from users and therefore consistent improvement to the project as requirements evolve. The second reason is the limitation in time. The project development took place in a short period of approximately less than 4 months. RAD not only enables a fast development cycle but also a consistent delivery of products in the form of prototypes.

3.2 Initial Requirements

The first step in RAD is the elicitation of requirements. The initial requirement was formed with the traditional data-gathering technique [12]. An interview was set up with the user (Vasileios Koutavas) of the product. The interview established the main stakeholders of this project.

1. Verification tool developers
2. Verification tool user

V. Koutavas serves as both of these stakeholders therefore he is capable of providing views from both sides of the stakeholders. Where stakeholder is mentioned in the rest of the dissertation, it is referring to V. Koutavas. Due to the time constraint of this project and the need for ethics approval for human interactions. The only stakeholder examined is V. Koutavas. Therefore, it can be subjected to bias and incomplete coverage of

requirements. It is advisable to communicate with more than a single source of stakeholders for a future project to obtain a richer understanding of needs.

The interview also established the initial requirements by identifying the needs of the stakeholders.

1. Establish a framework for visualizing graph outputs of multiple verification tools
2. The tool should be accessible by the stakeholders
3. Produce interactive visualizations for exploration of data

It was identified by the stakeholder that a large number of verification tools only have text-based outputs and a large number of verification tools have graph-related outputs. This led to the first requirement of a visualization tool for graph outputs. The tool should be easily accessible to developers and researchers. It is typical for verification tool developers to showcase their tool and users of that tool to experiment with the tool on ad hoc. Therefore, the verification framework should be accessible in terms of how easy it is for users to experiment with the underlying verification tool and how easy it is for developers to showcase their tools using the framework. It was also identified that it is valuable to understand the information contained within graphs, nodes, and edges. It would be very useful to be able to interact with the graph to have a better understanding of the output. The requirements obtained here were very crude, as described in [12], it can be difficult for people to fully describe their need. Therefore, prototypes are established to aid the extraction of needs from the stakeholder in the next phase of user design.

3.3 Requirements Refinement

As described above, the requirements are refined using prototyping. There are a total of 3 prototypes excluding the final version.

3.3.1 Initial Prototype

To satisfy the initial requirements, the initial prototype employed a web application to support high accessibility over the internet. Using a web app, the user does not need to download the tools and install them in their own environment, the developer can also easily showcase their tools to anyone. The high-level architecture of the initial prototype can be seen in figure 9.

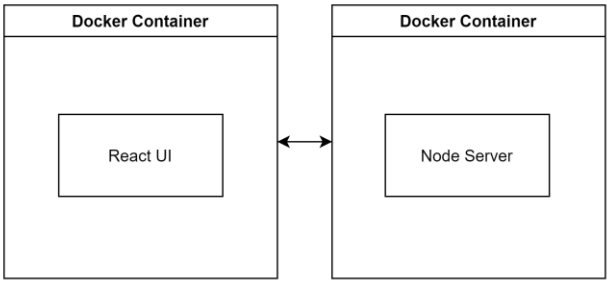

Figure 9 Initial prototype architecture

The framework consists of a frontend UI using React.js and a backend server using Node.js. This setup satisfies the accessibility requirement by allowing developers to set up their services in the backend server and visualize the outputs to the users using React's highly performant client-side rendering. React also support the use of D3.js which is a well know JavaScript visualization library that is used to produce interactive visualizations. To produce visualizations for multiple tools, there is a need for a standardized output format. The format employed for this project is DOT. As seen in figure 9, the components are run in Docker containers. This is used to allow developers to create an isolated environment on the server for the execution of different verification tools that may potentially need various environment setups. It also makes the server very portable, meaning the developer can take the container and serve it on any host without worrying about the host environment. This feature enables the developer to a consistent environment for integrating their tools with the framework.

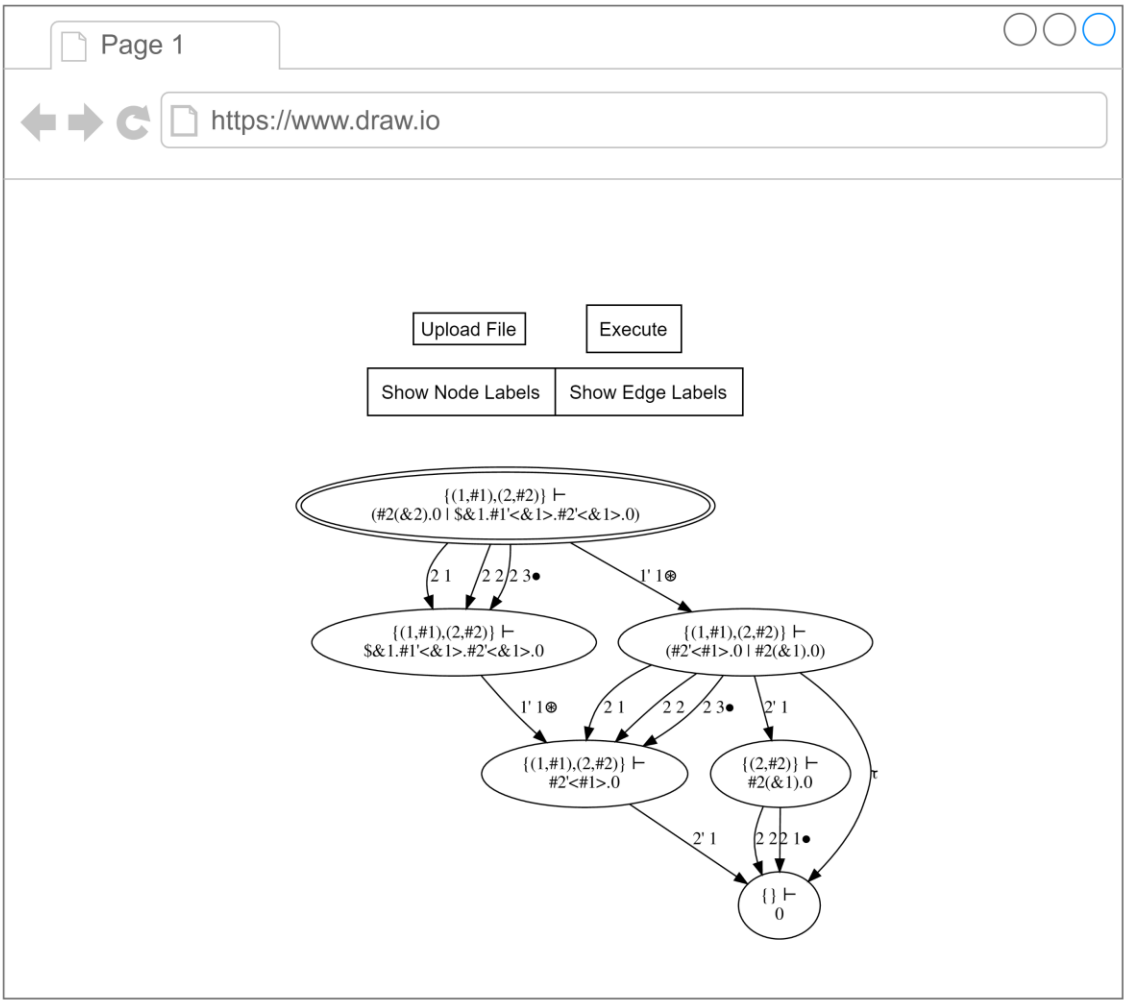

Figure 10 Initial prototype UI

Figure 10 displays the initial UI prototype. Here the user is capable of uploading a file as the input for the execution of the verification tool in the backend. The output DOT is rendered in the UI using d3-Graphviz [13] library. Concerning interactivity, the visualization is capable of zoom by mouse scroll, panning by dragging using left click, and showing/hiding node/edge labels by clicking on respective buttons. These interactions allow the user to navigate through the graph and optionally show/hide information.

Feedbacks were obtained from the stakeholder on the initial prototype. First of all, it is very common for users to input their source in the form of text. Using files as the input is rather tedious since a file must be created first and it is easy to just copy and paste into a text if a file does exist. Uploading a file also raises security concerns where verification tool developers can be a researcher serving it on a university network, uploading arbitrary files to it may cause security risks. The stakeholder also conveyed that it would be very useful if by clicking on one of the nodes or edges, it would show data contained in it.

This would allow for fine-grained exploration of the data while not piling all the information to the user at once.

The requirements are refined by extracting the good feature from the initial prototype and the feedback obtained from the stakeholder. The new requirements are listed in Table 1

Table 1 Requirements following prototype 1

Functional Requirements	Non-functional requirements
1. User should be able to choose which tool to use 2. User should be able to obtain an interactive visualization using the tool with the source input 3. The user should be able to zoom and pan the visualized graph 4. User should be able to see data for a node/edge by clicking on it 5. User should be able to show and hide labels 6. User should be able to input source code using text 7. The developer should be able to easily integrate a tool to the backend	1. The tool should be accessible by the stakeholders

3.3.2 Second prototype

Following the refined requirements, a new UI prototype was established. The high-level architecture remains unchanged. The UI prototype can be seen in figure 11

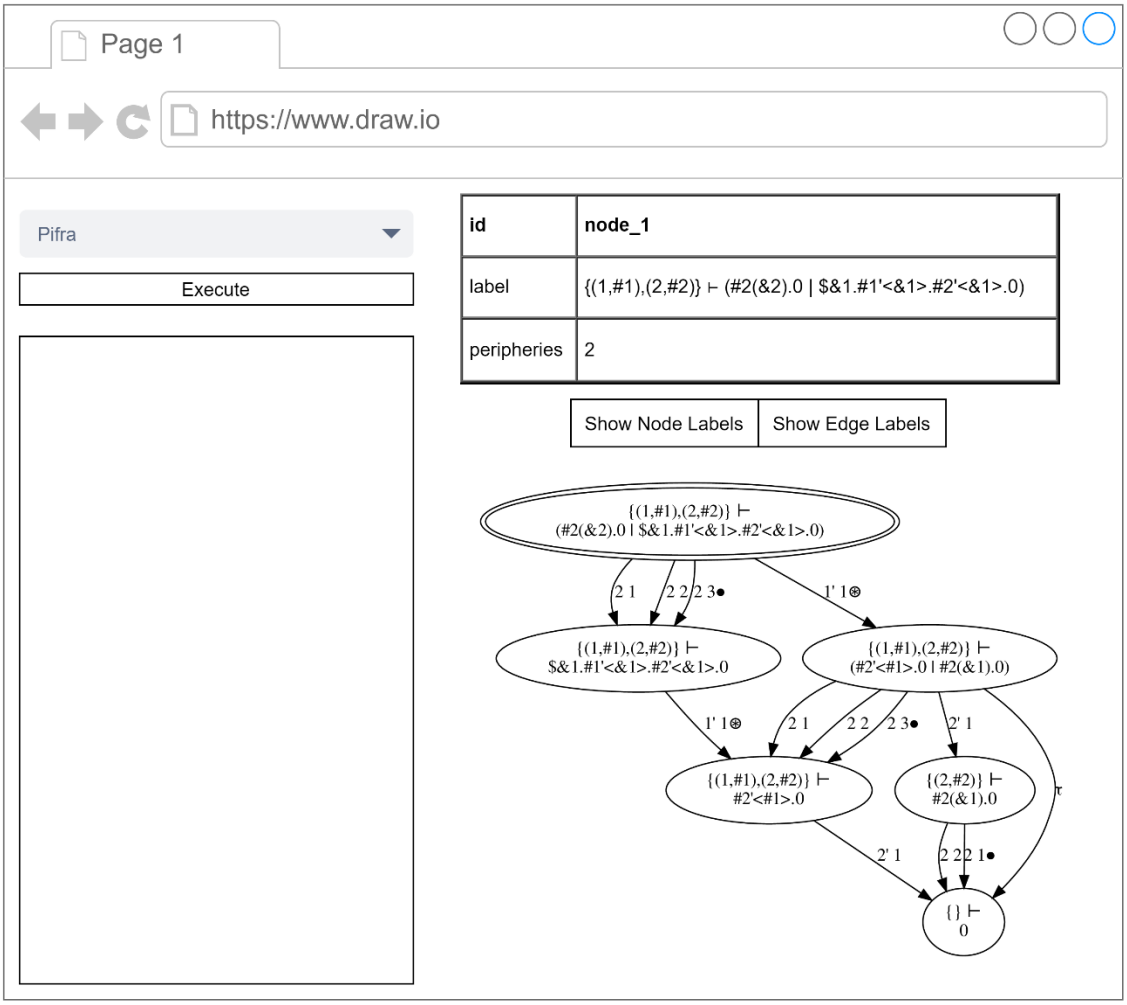

Figure 11 Second prototype

The new requirements from the previous iteration were added to this prototype. First, the user will be able to select a node/edge and see the information of the component in the table above the main visualization. File upload was replaced by an input box on the left side of the screen. The user also has the access to select the tool they wish to use from the dropdown menu on the top left.

This prototype was also shown to the stakeholder for gathering feedbacks as the previous iteration. For this iteration, the stakeholder raised one more use case for the visualization. For bigger graphs, it can be very useful to be able to focus on particular subgraphs or paths. It can help the user to navigate the graph better if the user is capable of highlighting paths between the two nodes. It was also noted that the UI should supply a number of example source input files for the user to experiment with. This had the benefit of showcasing the usage of the verification tools and providing guidelines for novice users.

It was also noted at this point that different verification tools may have optional execution parameters, some of the parameters may even require an input value from the user. Therefore, the need for the UI to send back parameter values was established.

Established requirements from information gathered from this prototype are displayed in table 2

Table 2 Requirements after the second prototype

Functional Requirements	Non-functional requirements
1. User should be able to choose which tool to use 2. User should be able to obtain an interactive visualization using the tool with the source input 3. The user should be able to zoom and pan the visualized graph 4. User should be able to see data for a node/edge by clicking on it 5. User should be able to input source code using text 6. The developer should be able to easily integrate a tool to the backend 7. Users should be able to input different parameters for tool execution	1. The tool should be accessible by the stakeholders 2. User should be able to show and hide labels 3. Example Files should be provided for on-demand usage of the tools 4. Visualization should highlight paths between two nodes

3.3.3 Third Prototype

The third and final prototype incorporated the requirements gathered from the previous iterations. Again, the high-level architecture remains unchanged. The UI prototype is shown in figure 12

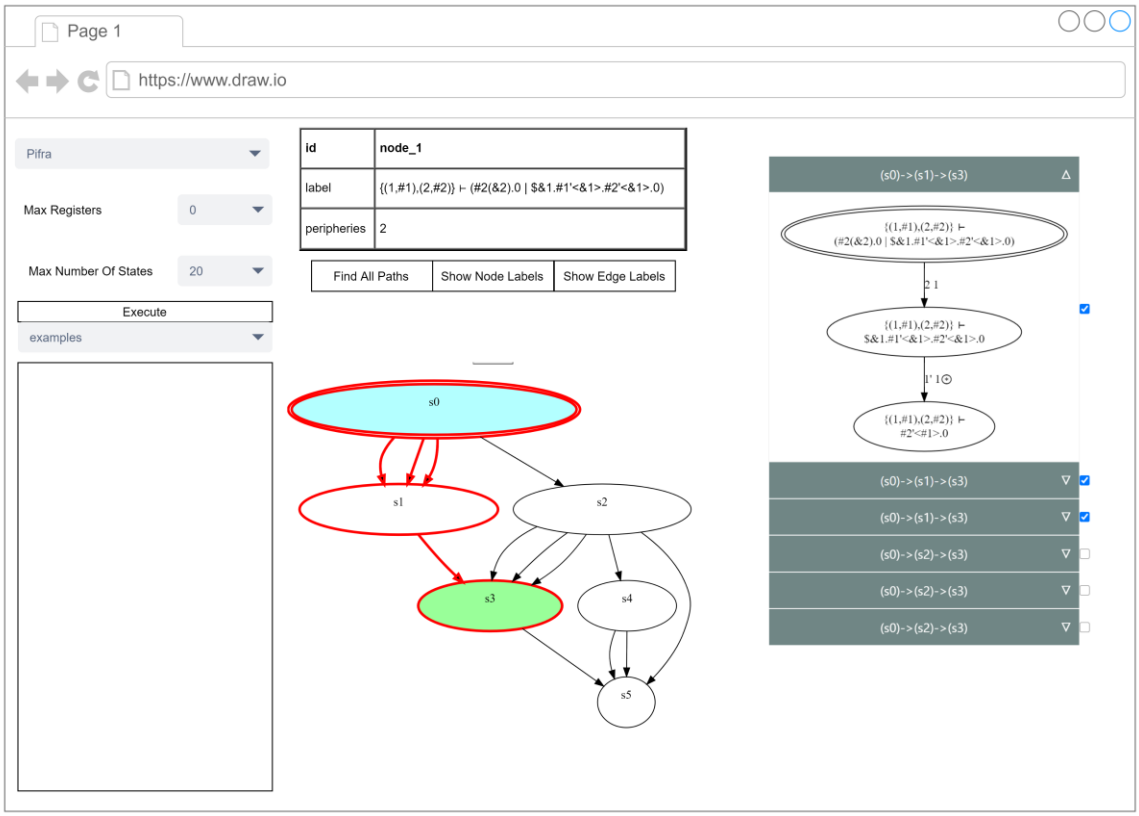

Figure 12 Third UI prototype

The main improvement following from the last iteration is the addition of the non-functional requirement features to improve usability. This includes the addition of parameters (max registers and max number of states shown for Pifra), examples files, and the ability to find all paths between two nodes highlighted in blue and green. The paths found are shown on the right is a set of collapsible components, on opening each component, a subgraph containing the path is rendered. By ticking the box on the right of the path the path is highlighted to guide navigation.

The feedback from the stakeholder on this particular prototype is mainly focused on usability. The main argument here is the lack of guidance on the usage of the UI itself and the underlying verification tools. From a user point of view, it is difficult to figure out what can be done to the graph visualization, for example, it is not stated anywhere how to select source nodes and destination nodes for finding paths. It also not apparent what the parameters do with the tool executed in the backend. The main component being the visualization also only takes up a small portion of the screen.

As a result of the above feedbacks, a number of usability requirements were established. The final list of requirements can be found in table 3

Table 3 Final requirements

Functional Requirements	Non-functional requirements
1. User should be able to choose which tool to use 2. User should be able to obtain an interactive visualization using the tool with the source input 3. The user should be able to zoom and pan the visualized graph 4. User should be able to see data for a node/edge by clicking on it 5. User should be able to input source code using text 6. The developer should be able to easily integrate a tool to the backend 7. Users should be able to input different parameters for tool execution	1. The tool should be accessible by the stakeholders 2. User should be able to show and hide labels 3. Example Files should be provided for on-demand usage of the tools 4. Visualization should highlight paths between two nodes 5. Provide usage information for the UI components 6. Provide information on the verification tools through UI 7. Provide descriptions for parameters

Section 4 Implementation

4.1 Overall Architecture

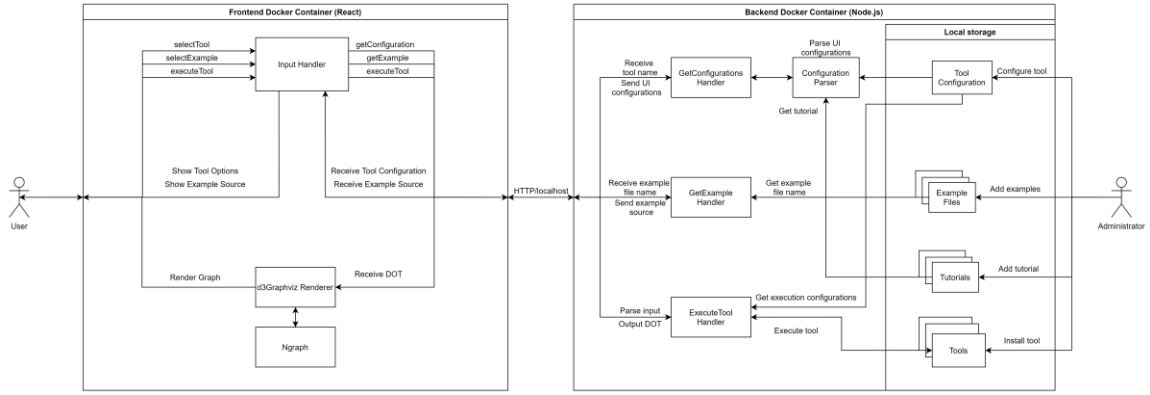

Figure 13 Architecture diagram

The overall architecture diagram can be seen in figure 13. The system consists of a user-facing frontend developed in React and an administer-facing backend developed using Node.js. User represents verification tool users interacting with the UI to generate and interact with the visualizations. The administrator represents the person who is responsible for setting up the backend verification tool integrations. This would usually be the verification tool developer integrating their tools with the framework. As mentioned before, both the frontend and backend components are placed in a docker container for isolation.

There are two main components in the front end. The input handler and a visualizer. The input handler is responsible for handling user inputs and making requests to the backend for data retrieval. There are 3 main request routes that the system takes. getConfiguration, getExample and executeTool. getConfiguration request is sent every time the page loads to obtain the latest configuration for the UI components to display. This will be explained in more detail later. getExample retrieves the source file of the example provided by the tool and executeTool send the input source with parameters to the backend for the execution of the verification tool. On receiving a response containing the DOT source of the graph. The DOT file is rendered in the visualizer component using d3-Graphviz [13]. The file is also parsed into Ngraph [14] for use in implementing interactivity.

On the backend server, three main request handlers are corresponding to the request routes from the frontend. The administrator is responsible for setting up the necessary configurations and files needed in the filesystem of the docker container. There are 4 main components that the administrator should set up. However, only the configuration file and tool installation are required. If the developer does not provide examples or tutorials, the system will still work.

4.2 User Interface

In this section, a detailed view of the user interface of the project's end product will be discussed in detail. For demonstration purposes, the framework is integrated with Pifra mentioned previously.

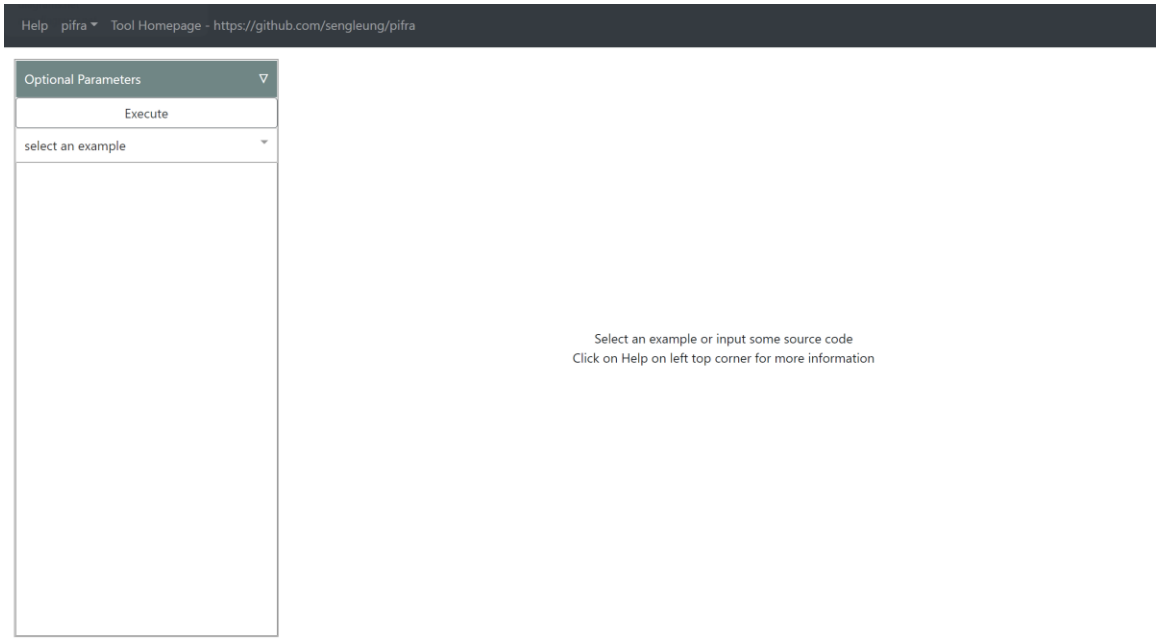

Figure 14 Example UI on the first load

In figure 14, the UI on the first load is shown. As mentioned before, configurations for the UI are obtained from the server for rendering the components. This includes the tools selection of the top left of the screen, the example file names under the execute button, the Pifra tutorials that will be shown if help on the left top corner is clicked, and the optional parameters for the execution of Pifra. Improving from the last prototype. The UI in the final product is much more informative on how to use both the visualization and the underlying verification tool with the addition of the help button. On licking the help button, a description of the UI and the verification tool will be displayed to the user. The tool developer could also provide a tool homepage that will be placed beside the tool selection for the user to navigate to for more information. Optional parameters are by default collapsed such that the user does not need to know about them and use the default values if they just want to simply test out the tool. Example files are available for selection if they are defined by the configuration. On clicking execute button, the input source and parameters will be sent to the server for execution.

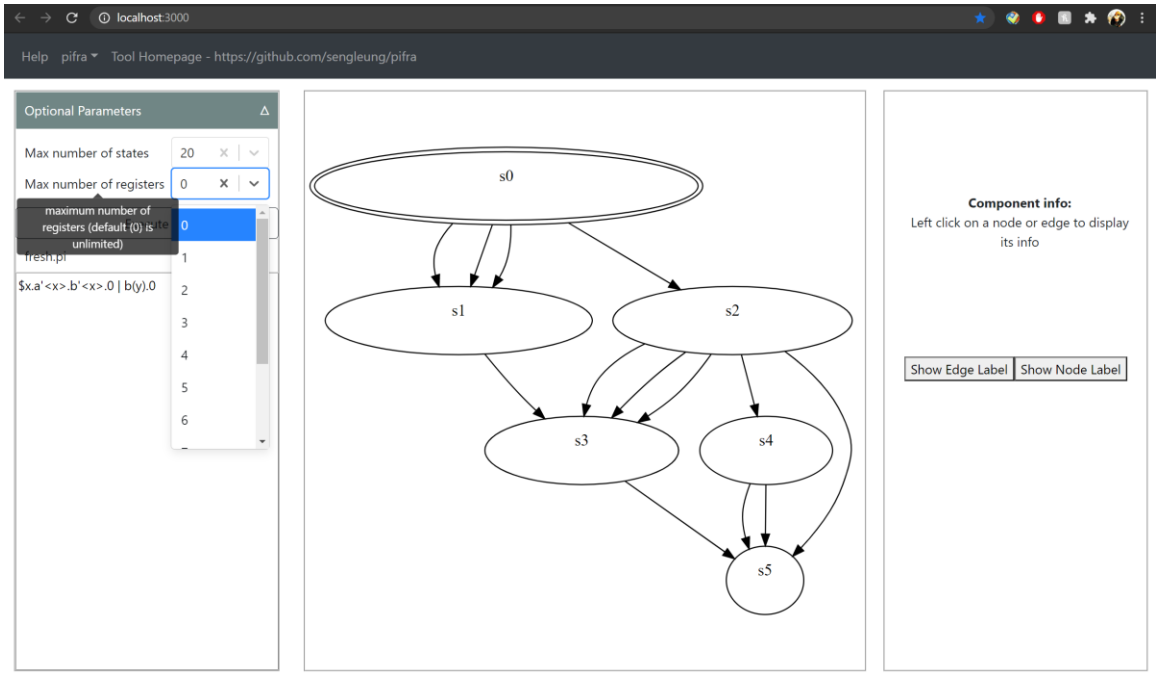

Figure 15 Example UI with a visualization rendered

In Figure 15, the visualization is rendered using the output DOT file obtained from the inputs shown on the left. Description of the parameters is displayed when the mouse hovers over its label. A set of predefined parameter values are also available for selection. On receiving the output files, the visualization is displayed in the middle of the screen, taking up the majority of the screen. Component information table and show label buttons were also in the middle of the prototype were moved to the right to give more space for the main visualization. At its initial state, very little information is displayed to the user. Node and edge labels can be shown to the user by clicking on the respective buttons on the right side of the screen. The area where component information will be displayed shows a message that guides the user in clicking on nodes/edges to explore the graph in more detail.

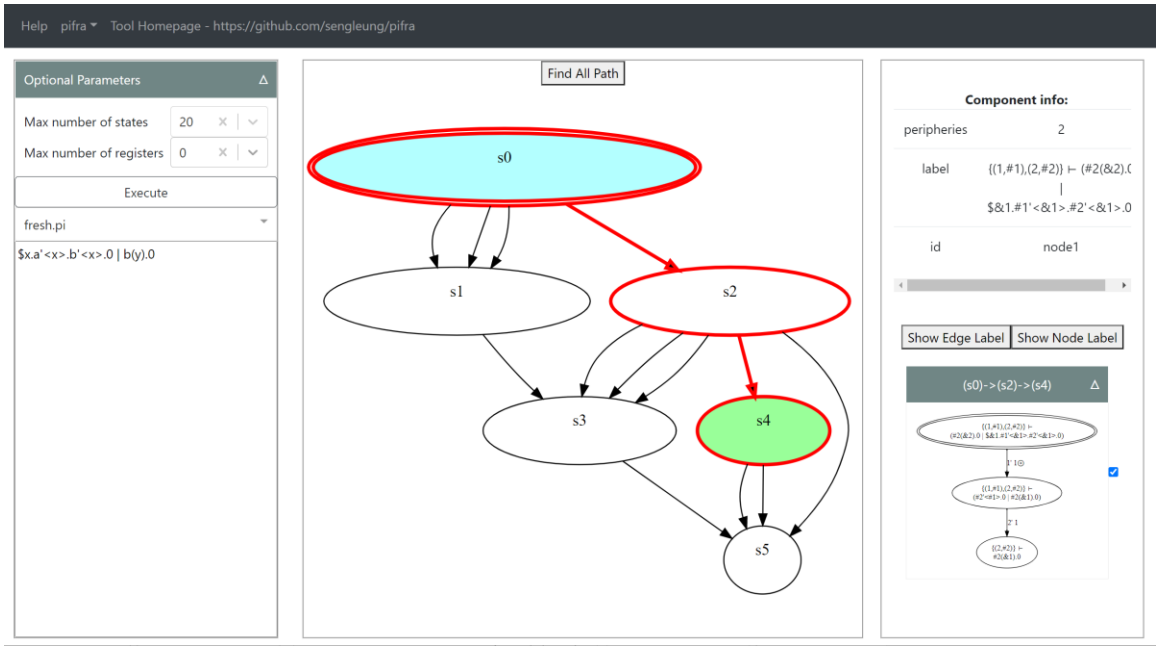

Figure 16 Example UI with paths and component data

In figure 16, by left-clicking on a node, it marks it as the source node indicated by the blue color. A destination node can be selected by using shift + left-click on a node which is indicated in green. When both the source node and destination node is selected, a button will show on the top of the graph. By clicking the button, the path between source and destination node will be found and displayed on the bottom right. Each path is contained in a collapsible component and collapsed by default. When the collapsible component is triggered, the visualization of the path will be rendered. The tick box beside each path collapsible component allows the user to highlight a path in bright red when toggled.

4.3 Technologies used

In this section, the main technology used to build the system will be discussed in more detail with respect to why they were chosen and how they were used.

4.3.1 Docker

Docker is a platform for containerized applications. Container technology has been around for many years but Docker has been very popular due to its high performance. [15] Comparing to the virtual machine environment, Docker has much better performance and start-up time. This is due to the essential difference between Docker and a virtual machine where Docker virtualizes the host operating system while a virtual machine virtualizes the host hardware. This would also mean we can serve multiple containers on a single host without wasting resources as would using a virtual machine. [15] [16]. The main components provided by Docker are docker images, docker containers, docker registries, and docker client. [15] for the purpose of this dissertation, docker registries and docker client will not be explained. A docker image is considered a snapshot of work which comprised in a file and contains all the system dependencies and parameter. On the other hand, a docker container is considered an application that is ready to run. It is created using a docker image. [15]

Docker is an integral part of this system to provide an isolated environment to run the web apps. This is especially true in the case of the backend server. It provides a way for verification tool developer to integrate their tool with the framework without considering the host environment of the server as the advantage of Docker is the ability to deploy on any host. Consider a scenario where a university researcher is developing a verification tool on a personal machine. If the researcher would wish to showcase the tool on a university host machine, the host machine may need to be properly configured with a suitable environment or the tool needs to be modified to fit the environment on the university machine. Using Docker in this case, helps the researcher to focus on the tool rather than wasting time on considering where it would be served.

Specific to this dissertation, two ways allow one to install verification tools in the resulting docker image. The first method is installing it via commands that are executed when building the image using the Dockerfile. An example with Pifra is shown in figure 17 where the command to install Pifra is preceded by RUN. A second method directly copies the executable into the container volume where the files for the container are stored. Volumes are a virtual filesystem over the host machine. This copy of the file can

also be specified in the Dockerfile using the COPY directory. The files copied can then be directly accessed in the container.

```
# Configure Go
ENV GOROOT /usr/lib/go
ENV GOPATH /go
ENV PATH /go/bin:$PATH

RUN mkdir -p ${GOPATH}/src ${GOPATH}/bin

# Install Glide
RUN go get -u github.com/Masterminds/glide/...

WORKDIR $GOPATH

CMD ["make"]

# install pifra
RUN go get -u github.com/sengleung/pifra/pifra
```

Figure 17 Example Dockerfile snippet

4.3.2 React.js

React is a JavaScript library that is lightweight and contains view only. It is not a framework like Angular.js which means it does not have state managers, routers, and API managers in the core library React is a component-based technology that allows encapsulated components to manage their own states and behaviors. React also features the use of virtual DOM, which is a significant performance improver. Virtual DOM operates by placing an extra layer between the real DOM and the user layer. The virtual DOM is a virtual representation of the real DOM stored in memory. Any changes in components are absorbed by the virtual DOM and the algorithm will identify which part in the real DOM has been modified and only updates those parts of the DOM. This means a significant performance improvement. [17]

The use of React for the front end of this dissertation is largely associated with the performance gained from virtual DOM updates. This is considered for the interactions of graphs, especially when graphs become larger in the number of nodes and edges, updates to the graph component should benefit from the virtual DOM updates. The second reason for using React is because it is lightweight, it is very easy to build prototype UI with its component-based model. With the help of React Hooks [18], rapid prototyping can be done using functional components that are relatively simple to build. Comparing to traditional prototyping methods which produced static prototypes, developing prototypes in React can help with simulating behaviors of the end product to provide more context for the stakeholder. Code used for prototyping can also be used for the end product which means development was already started as part of the prototyping tasks.

4.3.3 Node.js

Node.js is an asynchronous event-driven JavaScript runtime. It can handle many connections concurrently by using callback functions. [19] It is typically used as server-side JavaScript which integrates nicely with React. However, this may not be the best choice for the backend in this dissertation. This will be explained in the evaluation section later on.

Node, more specifically Express [20], a minimalist web framework for Node is used on the backend for handling requests as described in previous sections. The main task of the server is to execute the tools using the input source and parameters. This is done using child_process.execFile [21] function provided by Node. There are many ways that Node can use to execute a file as described in [22], execFile was purposely used because it does spawn a shell when executing the file. This is important to prevent command-line injection attacks in which malicious input can be passed in and used to run unintended commands if the command was dynamically built and run in a shell. [23]

4.4 Visualization

In this section, the components used for producing the graph visualization will be discussed in more detail.

4.4.1 DOT

Producing visualizations for different tools without a standardized output format is very difficult since there is no technology that is capable of visualizing any arbitrary graph representations. To produce consistent visualization experiences for the user. DOT language is used as the standardized output format that should be employed by tools that wish to integrate with the framework.

DOT is a graph description language that was first proposed in [24]. The language is fully supported by Graphviz visualization software as described in their documentation [25]. Using DOT language graph can be described in the text and customized with attributes. An example DOT file is shown in figure 18. As depicted in the figure, attributes can be defined within the square brackets. There are many useful attributes that are recognized by Graphviz, including changing shape, color, labels, etc. documented. This feature allows the visualization in the UI to be displayed with a consistent layout defined by Graphviz while also keeping the customization of the graph to the verification tool developer to create their customized graph for their use cases.

```
digraph {
    s0 [peripheries=2,label="{(1,#1),(2,#2)} ⊢
(#2(&2).0 | $&1.#1'<&1>.#2'<&1>.0)"]
    s1 [label="{(1,#1),(2,#2)} ⊢
$&1.#1'<&1>.#2'<&1>.0"]
    s2 [label="{(1,#1),(2,#2)} ⊢
(#2'<#1>.0 | #2(&1).0)"]
    s3 [label="{(1,#1),(2,#2)} ⊢
#2'<#1>.0"]
    s4 [label="{(2,#2)} ⊢
#2(&1).0"]
    s5 [label="{} ⊢
0"]

    s0 -> s1 [label="2 1"]
    s0 -> s1 [label="2 2"]
    s0 -> s1 [label="2 3●"]
    s0 -> s2 [label="1' 1⊛"]
    s1 -> s3 [label="1' 1⊛"]
    s2 -> s4 [label="2' 1"]
    s2 -> s3 [label="2 1"]
    s2 -> s3 [label="2 2"]
    s2 -> s3 [label="2 3●"]
    s2 -> s5 [label="τ"]
    s3 -> s5 [label="2' 1"]
    s4 -> s5 [label="2 2"]
    s4 -> s5 [label="2 1●"]
}
```

Figure 18 Example DOT file

4.4.2 D3-graphviz + D3.js

D3-Graphviz [13] is a JavaScript library that renders the graph described in DOT into an SVG using Graphviz library underneath. The rendering is implemented using D3.js which is a well-known JavaScript visualization library. D3-Graphviz library provides basic interactive features such as panning, zooming, transition animation, etc. Apart from those features, more advanced interactions are implemented using D3 to manipulate DOM components. D3-Graphviz renders nodes and edges in the form of DOM elements such that some attributes defined in the original DOT file such as labels are not easily accessible using D3 element selection. Some attributes are also completely inaccessible from the DOM elements, for example, the attribute peripheries are only used to determine the number of shape elements used for that node. Another issue with the library is that if the original DOT source does not provide a unique id attribute for the nodes and edges, there is no way of determining which DOM element maps to which node/edge from the original DOT. This poses a challenge when information needs to be extracted from the DOT file and displayed to the user as in the use case of the component information table. It also meant graph algorithms were not supported by using the DOM elements alone.

4.4.3 Ngraph

To overcome the challenges described above from using D3-Graphviz alone, ngraph is used to produce a more usable runtime graph data structure. Ngraph is a collection of graph-related packages. This dissertation uses ngraph.graph, ngraph.fromdot and ngraph. todot. Ngraph.graph provides a graph data structure for JavaScript. [14] It supports both muti-graph and directed graph. Each node/edge can store custom information in the form of objects. The data structure is simplistic and suitable for representing the DOT source. Ngraph.fromdot parses a DOT file and generates a ngraph graph object while ngraph.todot creates a DOT file from ngraph graph object

Ngraph is used in both frontend and backend. In the backend, a ngraph graph object is produced from the output DOT of the verification tool. This graph object is used to give each node and label a unique id so that the DOM components can be directly mapped to nodes/edges in the DOT file. We cannot completely rely on the verification tool to always supply id attributes to its node/edge. Therefore, this step is rather necessary. In the frontend, a ngraph object is also produced to provide a usable data structure. It currently has two main use cases. The first use case is retrieving component information when a user clicks on a node/edge. The id of the DOM component is used to retrieve data of the node/edge in the graph object with the same id. The second use case is performing a pathfinding algorithm. Using the graph object, a DFS is used to find all of the paths between two nodes. [26] In the case of a graph representing structures like a finite state machine. Cycles are very likely to happen with recurring states. DFS does not natively support cycles. The DFS is slightly modified to support cycles in the path by remembering the number of times the node has been visited in a dictionary instead of only keeping whether it has been visited or not. A path is then invalidated if the number of times a node has been visited exceeds a defined threshold. The current threshold defined is 1. This covers the scenario where self-transition is a part of the path. A standard DFS would ignore such a path. An example of this is shown in figure 19

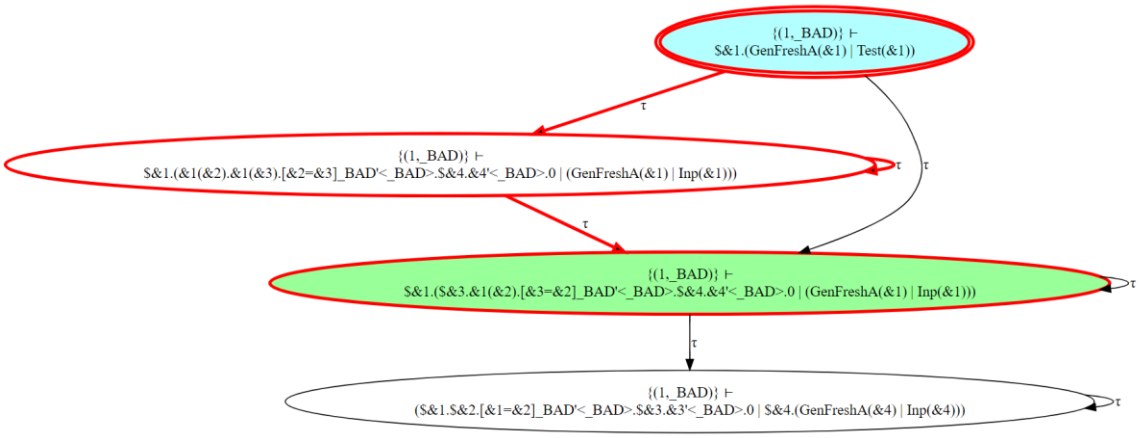

Figure 19 Path with cycle in the form of self-transition

4.5 Configuration

To integrate various verification tools with the framework, a configuration file is needed to define both configurations for the frontend and backend. For this dissertation, a single configuration file setup is used. This means the frontend will need to query the backend for its UI configurations. This removes the need to configure at two places. The configuration file itself is a JSON file. This is readily readable by JavaScript. The configurations are designed to give the verification tool developer high degrees of customizability. An example of configurations for Pifra is shown in figure 20

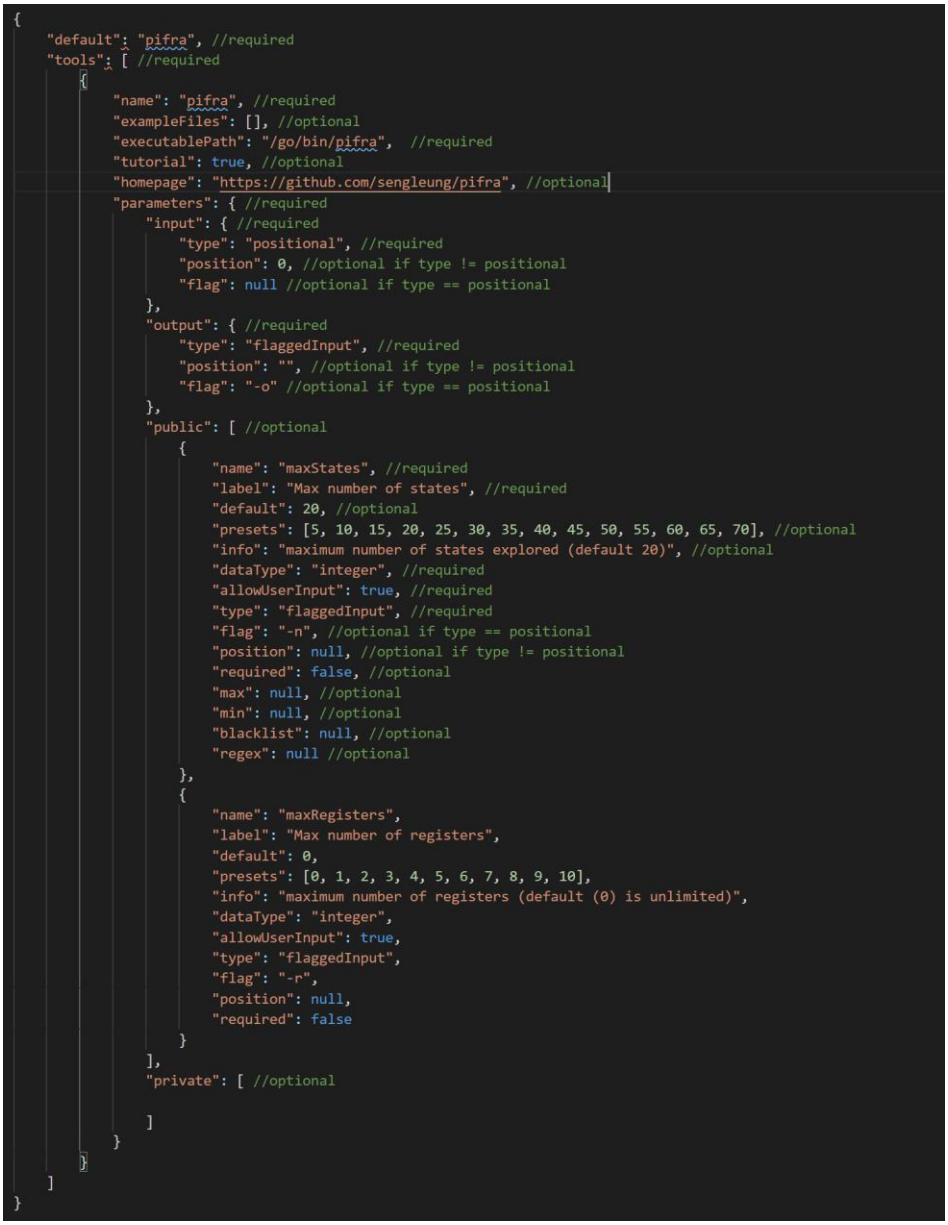

Figure 20 Configuration file for Pifra

The top-level object has two attributes, default and tools. These are required for the functionality of the system. Which means the system requires at least one tool to work? Default is a string of the name of the tool that is chosen to be the default tool to show when loading the UI. This should match the name given in the actual configurations of the tool. Tool's attribute is a list of tool configuration objects with each object representing the configurations for that tool. The tool configuration object has 3 required attributes, name (used to access configurations), executablePath (used to run the tool), and parameter (a list of parameter metadata for the execution of the tool). The executable path should be a path within the docker container. ExampleFiles attribute is an optional list of file names under the directory /tools/{toolName}/examples in the server source folder. If a value is not supplied for this, files from that path are read to form the list. The

tutorial attribute can either be true to indicate the tutorial file named tutorial.md should be located in the directory /tools/{toolName}/ or a string in markdown format. Tutorials are displayed in markdown on the UI. There is also an optional homepage attribute to allow the display of the tool homepage on the UI.

The parameters attribute is divided into 4 groups. Input, output, public and private parameters. Only input and output are required for the functionality of the system. Input and output parameter objects represent the input and output command-line argument respectively. They have the same attributes type, position, and flag. The options are minimal as the path of input and output files are predefined in the code execution. This is done to limit variation in the project structure. The type attribute defines the argument type. There are in general 3 types of arguments, positional, flag, and flaggedInput. Positional arguments require a position attribute that indicates the position of the argument in the execution command. The positive index starting at 0 represents the position from left to right, while a negative index starting at -1 represents the position from right to left. A flag argument represents a single flag being set or unset as the argument. This type requires the flag attribute to function. Finally, a flaggedInput represents a flag followed by some input string. The input string for input and output is predefined as the path to the respective files. A flag attribute is also required for flaggedInput

Public parameters represent parameters that will be shown to the user in the UI while private parameters are there for completeness and niche use cases. A private parameter represents arguments set for execution by the backend server. This is not provided to the user. For example, this could be used to configure default execution settings when executing the tool but the user does not need to know about it. For a parameter object, name label, datatype, allowUserInput, and type are required. Name is the identifier used internally for identifying the parameters while the label is used for display in the UI. Datatype is required for type checking of the input. This is especially important when allowUserInput is set to true. Currently, 4 data types are supported, string, integer, float, and Boolean. These are checked during the input sanitization process. allowUserInput is a Boolean attribute that specifies whether the UI will allow the user to enter an arbitrary value. There are also optional attributes for individual use cases. The default attribute is mostly optional but it is required if allowUserInput is false since the input is either obtained from the user or the default attribute. Presets are an optional list of predefined values for the parameter. The user is capable of selecting from this list for use instead of manually inputting values. Info attribute is the text displayed to the user when the mouse has hovered over the parameter in the UI. This is intended to provide information about the parameter to the user. The attribute required is an optional Boolean value that specifies whether the parameter is required to have an input selected or inputted by the user. If set to true, UI will prevent the user from sending execution requests until a value is provided by the user for the parameter. It is by default false if not provided. Four sanitization parameters are also available. Max and min are optional when the datatype is either integer or float. It specifies the value to be in a limited range. A blacklist is a list of characters or strings that should not appear in the input string. Regex is a regular

expression string for JavaScript that can be provided such that a string input is always checked against the regex for validation. Validation is carried out using [27]

Section 5 Evaluation

For the evaluation of contributions in the dissertation, it is important to identify the user class being evaluated as the evaluation is done for someone as depicted in [28]. The user class that will be evaluated in this dissertation includes user and user-developer. This corresponds to the two main stakeholders of this dissertation being verification user and verification tool developer. There are many aspects of the software that can be evaluated including source code quality, usability, security, performance, business logic, architecture quality, data quality, and open-source code use as listed in [29]. This not an exhaustive list as the dimension in evaluable components is difficult to completely capture in the first place. The main area of evaluation focus that should be covered by this dissertation is usability evaluation. This is the most important aspect for evaluation because the project artefact is intended to be used by people and with it being user-centric by design, usability is of utmost importance. A brief evaluation of security and performance will also be given.

5.1 Usability Evaluation

The vast number of usability evaluation methods out there means that it is very difficult to capture the entire evaluation model. As summarized by [30] there are no universal ways of capturing usability and there are no evaluation methods that clearly determine whether or not a system is usable. This dissertation will try its best to determine usability using heuristic evaluation considering the limited resource of the project. Methods such as user testing, discount methods, and model-based methods mentioned in [30] do not fit well for the purpose of this dissertation. User testing is very much favored by the nature of this project, however, due to the limitation in time and ethics approval, it is not possible to set up user testing. The only accessible user of this project is the project supervisor V. Koutavas, in which evaluation could be biased when based on a single user.

Before evaluation can be done, a set of principles are to be established for heuristic evaluation. As mentioned in [30], there are numerous collections of guidelines for usability, however, a distilled collection of 10 usability heuristics have been established by Jakob Nielsen [31]. The heuristics from the list provided by Nielsen that will be used by this dissertation include visibility of system status, user control and freedom, error prevention, consistency and standards, recognition rather than recall, flexibility and efficiency of use, aesthetic and minimalist design, and finally help and documentation. Evaluation will be carried out for the above heuristics where applied, for both user interface and backend integration. In table 4, the definitions for the heuristics will be summarized from the original lists published by Nielsen [31]

Table 4 Usability heuristics

Visibility of system status	Provide the user with current system status so they learn about the outcomes of actions and determine the next steps
User control and freedom	Allow user to freely regret an action to avoid user feeling stuck and frustrated.
Error prevention	Errors are best to be prevented rather than notified. Check for errors before executing an action.
Consistency and standards	Consistent design with other software standards to meet users' expectations such that ideally, the user should not wonder what components do.
Recognition rather than recall	Minimize user memory load by removing the need of remembering information from previous states.
Flexibility and efficiency of use	Provide different ways in which the system can be executed such that different users may choose paths that are the best fit for them.
Aesthetic and minimalist design	Irrelevant information should be eliminated and visual design should support the user's goal.
Help and documentation	Help should be provided concisely and should be easy to search for. List concrete steps that need to be carried out

5.1.1 User Interface

Visibility of system status:

The visibility of system status is not very well satisfied for several cases. First of all, when requesting the backend, there is no information given to the user as to whether the requests have been made or if the request is being handled by the server. With the example of Pifra, in most situations where the graph is relatively small, the server usually returns the outputs rather rapidly. However, when processing is required, the user is not presented with information to show that the server is processing the request. The same holds for the pathfinding feature where the processing state is not visible to the user. This poses confusion for the user especially the problem of the user not know whether the tool is being executed or something has failed. This should be improved by implementing a feedback pipeline with the backend. On the other hand, the graph itself has good visibility. On hover on a node/edge, the borders of the component are thickened to show the user a component can be selected and when selected, the component border will remain thickened to show the selected component. A color filling is also present for a node to indicate the source and destination node.

User control and freedom:

The current UI support regrets of user actions to some degree. For example, the use of dropdown selection avoids the need for the user manually typing a selection and re-typing when the input is unintentional. Where ever a selecting dropdown component is used, the user is free to reselect the options at no additional cost. In the case of path selection, the user is also capable of freely toggling the highlights of paths. However, the user is not free to regret the request to the server. This means if a user inputs a source, the user is not capable of canceling the processing on the server. This can be problematic when the input source is big and the execution of the tool takes longer than usual. In such a case the user is capable of immediately making another request to the backend with a different input source but the previous execution is not stopped, meaning taking unnecessary resources on the backend. This should be improved by allowing the cancelation of execution both on the frontend and the backend.

Error prevention:

The UI has several error-checking mechanisms. Including checking whether the input is empty and whether all required parameters have a value before sending a request to the backend. Feedback is given to the user in the above actions in the form of dismissible text boxes. With the use of these error checking, in most cases, the user can expect a valid request to be made to the backend when an example file is selected. In the case of when sources are manually inputted, the UI does not provide any error checking mechanisms such as source code validation, formatting, etc. This could be a major downfall to usability and should be supported in the future. In the case of the pathfinding feature, the UI also checks whether a source node and a destination node are selected before showing

the button to run the pathfinding algorithm. This removes errors in incorrectly selected nodes in the pathfinding algorithm. The user is also only capable of selecting a single source node and a single destination node for the same reason.

Consistency and standards:

There are similarities in the UI to others that have similar functionalities. For example, the use of a navigation bar for top-level information such as homepages, help, and top-level selection (tool selection in this case) is a common pattern to be found in many websites. Most importantly, the help button should be somewhere that the user expects to avoid user frustration when they are unable to find instructions. The general layout of the UI is inspired by [32] which provides some degree of consistency for users. The direct use of the Graphviz library for the visualization should also provide the user with expected output standards to the other Graphviz visualizations.

Recognition rather than recall:

For the use of the UI, the user should not need to remember any information. First of all, information of the graph itself is selectively presented to the user, although it may be hidden it is not lost, the user can simply enable the information they need by pressing on the components. The Ui also does not have page navigations, all information the user needs such as the parameter, input, the path is given to the user on the same page of the graph. This allows the user to refer back to information when needed.

Flexibility and efficiency of use:

In terms of flexibility of the UI, the user is flexible in choosing the tool, parameters, and input source they want to use, however, there is a level of restriction that is tied to the configuration defined by the verification tool developer in the backend. The user is not capable of freely executing the original tool as it would when downloaded on its own. The graph visualization also grants some degree of flexibility in the way it is presented. The user can choose to enable or disable labels, select node/edge, and display the path for the level of information they need.

Aesthetic and minimalist design:

The final layout of the UI is minimal in terms of its visual components. The level of UI visuals displayed to the user is mainly defined in three levels. The first level is when the user has not executed a tool yet, at this level, only the input section on the left is displayed. The next level appears when a graph is rendered. A graph canvas taking up most of the screen is display in the middle as the main component. The last level of information is displayed when the user selects a node/edge or finds paths on the right side of the screen, shown through the component information table and path lists respectively. The component information table could be improved as currently, it displays all attribute information. Some of which can provide little or no value to the user, for example, id and other Graphviz specific attributes. Such information could be hidden from the user to further enhance the minimalist design. However, some Graphviz attributes can be useful

in situations, for example, URL, it can be useful to view the URL being directed into. Which attributes are to be displayed should be evaluated in the future to produce a table containing only useful information for the user. The gradual display of information is kept minimal for users at each stage of its interaction. In the three levels, parameters, and path subgraphs are also hidden by default within a collapsible component to hide the information until the user reveals them when needed. The input section on the left is kept for users to refer back to their original input that produced the graph. However, it is arguable that it can be hidden once the graph is rendered and only displayed again when the user wants. On the graph itself, the node/edge labels are not shown at the start by default to allow the user to gradually adapt to information load. However, Due to how the Graphviz library function. The node is pre-sized determined by the label. Meaning if the label contains a large number of texts, the node's shape can look odd when the label is hidden as shown by the example in Figure19. This affects the aesthetic of the UI. It would require some workarounds such as dynamically modify the DOT file but would mean putting more workload on the CPU. An alternative could be to ask verification developers to place the labels into a custom attribute for display in the component information table rather than the Graphviz label attribute. However, this would depend on whether the developer feels they need to do this to prevent oddly shaped nodes.

Help and documentation:

In most cases, helpful information is given to the user. The main source of helpful information is with the help button on the top left corner. This contains instructions on how to use the UI in concise bullet point format. The help section may also provide a tutorial of the verification tool which is provided by the developer. Apart from the top-level help function, the UI also displays information for the parameters when the mouse has hovered over. A help message is displayed on the main screen when a graph is not rendered to help the user know that they need to input some source or select an example to start using the UI. Node/edge selection tip is also displayed in place of the component data table on the left when a node/edge is not selected to aid the user in selecting a component to view its information. The presence of these tutorials and documentation allows some form of welcoming features for novice users who are not familiar with formal verification and the tools but would like to experiment with them.

In summary, the UI satisfies most of the heuristics defined above to some extent providing good enough usability for the first version of the UI despite a lot of improvement can be made. Referring back to the requirements in Table 3, the UI has satisfied the requirements defined previously for the usability of the UI and its functionalities. Functional requirement (FR) 1 has been satisfied by the tool selection. FR2 has been satisfied by the input box and the visualization rendered after receiving the output DOT file. FR3 has been satisfied with d3-Graphviz. FR4 has been satisfied with the display of attribute data in the component information table, the information here however can be optimized more to selectively hide information that provides little or no value as described by the aesthetic and minimalist design section. FR5 was satisfied with the implementation of the text box and the removal of file upload from earlier designs.

FR7 was satisfied by the display of both required and optional parameter options defined by the backend configuration. Non-functional requirement (NR) 1 is satisfied by the use of a web interface to allow the user to access the tools without needing to download them or worry about the user environment. The addition of tutorials, helpful information, and tooltips guides the user to the usage of the tool and the UI. This in turns satisfies accessibility as it allows novice users to also have an easier time using it. Although it is satisfied, it can still be improved, for example, to optimize accessibility, even more, descriptions to the example files could be added and guidance on how to interpret the graph could also be added as a new feature. NR 2 was satisfied with the buttons hiding and showing labels. NR 3 was satisfied with a selection of example sources, currently, it only shows a name, it may be useful to show some description for the files too. NR 4 was satisfied by changing the border color of the nodes/edges to bright red upon selection by the check box. The color is chosen to make the path visible when zoomed out in larger graphs. Its efficacy is yet to be established because there are various types of color blindness that might not be able to identify bright red effectively. NR 5 was satisfied through the help documentation displayed after clicking on the help button. It has a description of how to use each of the UI components in bullet point format. NR 6 was satisfied with the optional tutorial of the tool also displayed on the help screen. The choice of display is using Markdown, this is the typical format used in documentation for GitHub which should be well known for most users in the field, creating a sense of consistency. NR 7 was satisfied by showing a tooltip when the parameter label has hovered over.

5.1.2 Backend

The evaluation for the backend is done for the user-developer user class. Some of the heuristics such as user control and freedom, recognition rather than recall, and aesthetic and minimalist design do not fit very well with the use cases therefore they will not be contained in this section.

Visibility of system status:

The visibility of backend runtime is not very visible, the execution states are only visible through stdout of the console, also, not all information is printed therefore it can be very difficult for verification developers monitoring the backend to visibility understand what is happening and debugging. A logging system is needed in place to help the developer understand what is happening in the system.

Error prevention:

There are a couple of error prevention mechanisms within the backend. First of all, the inputs are sanitized as previously described to minimize the possibility of execution error

of the verification tool. Secondly, access to example files, parameters, tutorials, etc. is key-value mapped. This means, only defined values will be used in the backend to prevent unrecognized variables. Lastly, in the case of Pifra, the output files are preprocessed to remove unrecognized tokens that will break the DOT parser.

Consistency and standards:

The backend uses Node.js which is commonly used by servers, the configuration file is implemented in JSON which is also a commonly used format for configuration files. This gives some level of consistency for the developers. The locations of the local files such as examples, tutorials, and temporary files created are in a directory defined by this dissertation. Therefore, it is not following any specific standards. This could raise some confusion as to where files should be and the structure of the project directories.

Flexibility and efficiency of use:

Flexibility is a key feature of the backend which is most evident in the definition of configuration files. As explained in previous sections, the configuration contains mostly optional attributes that the developer can use to define the level of information they provide to the frontend users. For example, parameters, examples, tutorials, homepage link, etc. The configuration can also have optional input sanitizations in the form of regex, blacklists, datatypes, and max/min. There is also a level of freedom in defining the types of command-line arguments supported by the backend, including the most common three types, positional, flag, and flagged input arguments. In the context of the visualization, the framework allows flexibility of verification tool-defined attributes with the use of DOT language, hence the tool itself is capable of defining how the graph should look like and what information is contained within. The backend however does not provide flexibility of arbitrary input and output modes. The only one supported currently is the use of files.

Help and documentation:

Help and documentation are currently not provided but should be soon provided in the form of the README file attached to the project.

In summary, the backend is lacking usability especially in the area of visibility and error prevention. However, it excels at providing flexibility to the developers. Again, referring back to the requirements in Table 3, the backend has successfully satisfied the requirements of being accessible to the developers and easy integration of verification tools using a flexible configuration file. FR 6 has been satisfied by the introduction of a flexible configuration file; a developer can easily make use of the framework by supplying the configuration file. However, it should be noted that optional functions such as example files, tutorials require the developer to put the files into a predefined directory. This can be cumbersome for the developer. It might be beneficial to implement the flexibility to use any directory for the files and define them in the configuration file. NR 1 is satisfied by the backend to an extent with the adoption of Docker to provide an isolated environment for the developers. However, the visibility of system status is not up

to par for easy debugging and logging of the backend. This is inconvenient for the developer to use.

5.2 Security Evaluation

Security considerations have been sparingly considered during the course of development. With the use of command-line file execution, the most recognizable form of attack is command-line injection. Since this could lead to serious consequences such as running arbitrary commands on the host machine, this problem is addressed using the execFile function provided by Node. Other security risks are not actively handled but should be considered in future developments. The main risk stated in this dissertation should not be taken as a comprehensive list of the potential security risks. First of all, the famous DDOS attack is possible unless the host network is capable of preventing this. Secondly, the server is currently not identifying the origin of the requests, this means the API could be called from any arbitrary sources. This could potentially break the server whether it is intentional or unintentional by calling the API with arbitrary request payloads. Lastly, the server currently does not handle capacity management as they expect usage in the current stage is very low. This poses a problem of potential attacks to the server by concurrently sending multiple execution requests to the server which could potentially overload the host CPU. Concerning data security, it should be noted that any information that could be sent to the UI should not contain confidential information these include example files, tutorials, public parameters, homepage and output files, etc. Considering the use cases of the framework, it is unlikely that private information will be leaked. However, it is reasonable to be cautious with the network data that are transferred over the internet.

5.3 Performance Evaluation

For the initial artefact of the project, performance is not considered to be as important as usability. The framework is a demonstration of the potential architectural setup that provides the services. There are however a number of performance concerns that have been identified. The main performance bottleneck is the execution of the verification tool and the processing of graph output. The execution of the verification tool can be dependent on the host machine's processing power. It is reasonable to take a longer time to execute larger input files. However, as mentioned previously there are no capacity management mechanisms in place such that the performance of the entire host could be hindered if many processes are executed as a result of uncontrolled executions. Performance issues also exist with large graphs. Currently, in the examples used in the project, most of the graphs are small enough that can be processed in a relatively short time. Where very large graphs are outputted. It may be dependent on the user machine

whether the graph could be rendered within a reasonable amount of time. The main issue arises from the parsing of the DOT file into Ngraph objects and back. Although this step is costly, it is unavoidable currently as explained previously. Therefore, it may be advised that the framework should limit usage to small demonstrative graphs rather than production-level graphs with a large number of nodes and edges. Server-side rendering could also be implemented to alleviate computation load from the user machine. [33]

Section 6 Future Work

As identified in the evaluation, a large amount of work is still needed to optimize the usability, performance, and security of the framework. In terms of usability, for the front end, the most important work that needs to be done is making system status more visible. This may include the adding of loading transition when waiting for responses from the server and when the UI is finding paths of the graph. This should also include showing error messages and potentially showing execution logs of the underlying verification tool to the user to help them understand the workings of the verification tool. Apart from making system status more visible, the UI also needs work on providing the user the freedom to cancel executions, this may be the case where the user has executed the tool with incorrect input or the execution is taking too long and the user no longer wants to wait. Of course, in general, the usability of the UI can always be optimized further. The usability of the UI should also be evaluated in more depth preferably with the help of user testing to find a more complete view of the usability needs. This should ideally consist of users from different areas of expertise including students, novice users, and expert users to capture a variety of viewpoints.

From the backend point of view, there are a couple of important optimizations that are needed. On the top of the list is resource management. As requests come in, a tool process that takes longer to execute can cause problems by hogging system resources causing CPU and memory overload. As mentioned before this is currently not managed. Measures should be placed such that only a certain number of processes can be running at the same time, during this period, the server should not receive any more requests employing a semi-blocking model. This does however hinder the scalability of the backend but it could be scaled by adding more host machines to handle the requests. As mentioned in the evaluation, the usability for verification developers is can be improved upon, including logging of system states and adding detailed documentations. The configuration file proposed by this dissertation can also be improved by adding more flexibility features such as supporting more input and output modes, customizable logging showed to the user, DOT attribute definitions, etc.

For the Rendering of large graphs, currently, it will be dependent on the user machine as it operates using client-side rendering with JavaScript. This should work nicely if the framework is only intended to showcase examples that output relatively small graphs. For large graphs, it could be beneficial to employ React server-side rendering so that the user machine is put high pressure for rendering the graphs. This should be traded off with the addition of network travel time to determine the suitable threshold for employing server-side rendering.

Section 7 Conclusion

In conclusion, the main contributions of this dissertation involve the making of a web framework using a React.js frontend and Node.js backend for showcasing graph visualization of verification tool outputs. The frontend employee client-side rendering using D3.js and Graphviz to render DOT files into standardized visualizations. Interactivity is provided with visualization to allow for the exploration of graph data. The UI provides the user with the ability to input source code and parameter values to the backend for execution. The UI also provides features to display tutorials, tooltips, and predefined examples to help novice and expert experiment with verification tools. The backend executes verification tools from request payloads and with the help of a flexible configuration file, the verification tool developer can easily integrate their tool. They use Docker to containerize the web app also contributes to the portability. It allows the verification tool developer to integrate tools in an isolated environment that can be deployed to any host. The framework still needs many improvements as described previously whereby the most important ones include system status visibility, backend resource management, and support for large graphs. However, hopefully, this framework can evolve into a useful tool for formal verification researchers.

References

[1] T. Munzner, Visualization Analysis and Design, AK Peter, 2014.

[2] V. Kasyanov and E. Kasyanova, "Information visualisation based on graph models," *Enterprise Information Systems*, vol. Vol. 7, no. No. 2, p. 187–197, 2013.

[3] I. Rodhe and M. Karresand, "Overview of formal methods in software engineering," Overview of formal methods in software engineering, 2015.

[4] S. Leung, "Modelling Concurrent Systems: Generation of Labelled Transition Systems of Pi-Calculus Models through the Use of Fresh-Register Automata," Trinity College Dublin, Dublin, 2020.

[5] J. Laird, "A Fully Abstract Trace Semantics for General References," *Automata, Languages and Programming*, pp. 667-679, 2007.

[6] C. Broadbent and N. Kobayashi, "Saturation-Based Model Checking of Higher-Order Recursion Schemes," *Computer Science Logic 2013*, vol. 23, pp. 129--148, 2013.

[7] M. Ulbrich, V. Klebanov and M. Kiefer, "LLRÊVE," [Online]. Available: https://formal.iti.kit.edu/projects/improve/reve/. [Accessed 26 02 2021].

[8] K. G. Larsen, W. Yi, P. Pettersson and e. al., "UPPAAL," Department of Information Technology Uppsala University (UPP), Department of Computer Science at Aalborg University (AAL), [Online]. Available: https://uppaal.org/.

[9] Microsoft, "Rise4Fun," Microsoft, [Online]. Available: https://rise4fun.com/. [Accessed 15 02 2021].

[10] E. Mnkandla, "About Software Engineering Frameworks and Methodologies," *AFRICON 2009*, pp. 1-5, 2009.

[11] C. Joy, "Rapid application development (RAD)," *Salem Press Encyclopedia. 2019*, 2019.

[12] B. Nuseibeh and S. Easterbrook, "Requirements Engineering: A Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, Limerick, 2000.

[13] M. Jacobsson, "d3-graphviz," [Online]. Available: https://github.com/magjac/d3-graphviz. [Accessed 06 02 2021].

[14] A. Kashcha, "ngraph," [Online]. Available: https://github.com/anvaka/ngraph. [Accessed 12 03 2021].

[15] V. Sharma, H. K. Saxena and A. K. Singh, "Docker for Multi-containers Web Application," in *2020 2nd International Conference on Innovative Mechanisms for Industry Applications* , Bangalore, 2020.

[16] "What is a Container?," Docker, [Online]. Available: https://www.docker.com/resources/what-container. [Accessed 15 02 2021].

[17] A. Modi, "React.js and Front-End Development," [Online]. Available: https://dzone.com/articles/why-choose-react-for-front-end-development. [Accessed 01 02 2021].

[18] "Introducing Hooks," React, [Online]. Available: https://reactjs.org/docs/hooks-intro.html. [Accessed 01 02 2021].

[19] "About Node.js," OpenJS Foundation, [Online]. Available: https://nodejs.org/en/about/. [Accessed 02 02 2021].

[20] "Express," [Online]. Available: https://expressjs.com/. [Accessed 02 02 2021].

[21] "Node.js v16.0.0 documentation," OpenJS Foundation, [Online]. Available: https://nodejs.org/api/child_process.html#child_process_child_process_execfile_file_args_options_callback. [Accessed 10 02 2021].

[22] C. Ho, "Understanding execFile, spawn, exec, and fork in Node.js," [Online]. Available: https://dzone.com/articles/understanding-execfile-spawn-exec-and-fork-in-node. [Accessed 10 02 2021].

[23] M. Hoppe, "Preventing Command Injection Attacks in Node.js Apps," 28 October 2020. [Online]. Available: https://auth0.com/blog/preventing-command-injection-attacks-in-node-js-apps/. [Accessed 02 03 2021].

[24] E. R. Gansner, E. Koutsofios, S. C. North and K.-P. Vo, "A Technique for Drawing Directed Graphs," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 19, pp. 214--230, 1993.

[25] "The DOT Language," [Online]. Available: https://graphviz.org/doc/info/lang.html. [Accessed 04 02 2021].

[26] P. E. Black, "all simple paths," 2 September 2008. [Online]. Available: https://xlinux.nist.gov/dads/HTML/allSimplePaths.html. [Accessed 03 03 2021].

[27] validatorjs, "validator.js," [Online]. Available: https://github.com/validatorjs/validator.js. [Accessed 12 03 2021].

[28] M. Jackson, S. Crouch and R. Baxter, "Software Evaluation Guide," Software Sustainability Institute, [Online]. Available: https://www.software.ac.uk/resources/guides-everything/software-evaluation-guide. [Accessed 15 04 2021].

[29] B. Kontsevoi, "The ultimate way to effective software evaluation," CIO, [Online]. Available: https://www.cio.com/article/3268647/the-ultimate-way-to-effective-software-evaluation.html. [Accessed 15 04 2021].

[30] G. Cockton, "Usability Evaluation," in *The Encyclopedia of Human-Computer Interaction, 2nd Ed.*, The Interaction Design Foundation.

[31] J. Nielsen, "10 Usability Heuristics for User Interface Design," Nielsen Norman Group, 15 11 2020. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/. [Accessed 16 04 2021].

[32] M. Jacobsson, "Graphviz Visual Editor," [Online]. Available: http://magjac.com/graphviz-visual-editor/. [Accessed 22 02 2021].

[33] R. Kumar, "How to implement server-side rendering in your React app in three simple steps," 29 FEBRUARY 2020. [Online]. Available: https://www.freecodecamp.org/news/server-side-rendering-your-react-app-in-three-simple-steps-7a82b95db82e/#:~:text=In%20Client%2Dside%20rendering%2C%20your,The%20output%20is%20HTML%20content.. [Accessed 04 04 2021].