Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Front-end part of the process testing data management system

Maximilián Herczeg

Supervisor: Ing. Matěj Klíma, Ph.D.
May 2024

BACHELOR'S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name: **Herczeg Maximilián** Personal ID number: **510641**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Front-end part of the process testing data management system

Bachelor's thesis title in Czech:

Front-end ást k systému pro správu dat pro procesní testování

Guidelines:

Create a design and implementation of front-end part of graph-like test data management system for system process testing.
User should be able to create account, to visualize and edit graphs, to visualize generated test sets, and to export the graphs and test data into JSON format. User also should be able to define attributes in the visualized graphs that add some specific properties to the graph's nodes, edges, or group of nodes or edges. Those attributes should be visually emphasized in the graph.
Using background research identify a set of 10 test data samples, which then model using the test data management system and store them to the constituted test data repository.
Test the implementation using a set of automated end-to-end tests.

Bibliography / sources:

Ammann, Paul, and Jeff, Offutt. Introduction to software testing. Cambridge University Press, 2016.
Bures, Miroslav, Tomas Cerny, and Matej Klima. "Prioritized process test: More efficiency in testing of business processes and workflows." Information Science and Applications 2017: ICISA 2017 8. Springer Singapore, 2017.
Mardan, Azat. React quickly: painless web apps with React, JSX, Redux, and GraphQL. Simon and Schuster, 2017.

Name and workplace of bachelor's thesis supervisor:

Ing. Mat j Klíma, Ph.D. System Testing IntelLigent Lab FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **07.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Mat j Klíma, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Ing. Matěj Klíma, Ph.D., for his expertise, patience, and guidance, which were invaluable throughout the process of writing this thesis. I would also like to thank doc. Ing. Miroslav Bureš, Ph.D. for giving me the opportunity to work on this project and his guidance and encouragement during the development. Finally, I would like to thank my friend, Daniel Holotík, for his work on the back-end module of the application, without which this project would not be possible.

Declaration

I declare that this work is my own work and that I have cited all sources I have used in the bibliography according to the methodical instructions for observing the ethical principles in the preparation of a university thesis.

In Prague 23.5.2024

Abstract

This bachelor thesis aims to design and develop a front-end module for a test data management system in the form of a web application. The application allows users to view, create, and edit system models based on directed graphs. In addition, the applications provides a user interface for interacting with the features implemented by the back-end module of the system. These features include storing, sharing, and accessing graphs on the server, exporting/importing graphs into files, and generating test cases for given graphs.

The text of the thesis describes the necessary terminology related to model-based testing and focusses on directed graphs as models of tested systems along with the test cases created using these models and coverage criteria.

The implementation of the web application is carried out using JavaScript and the React framework together with the JointJs library for diagramming and Mantine component library to build the UI.

Keywords: Model-based Testing, Path-based Testing, Directed Graph Visualisation, React Front-end Web Application Development

Supervisor: Ing. Matěj Klíma, Ph.D.
System Testing IntelLigent Lab FEE, Department of Computer Science, Karlovo náměstí 13,
121 35 Praha 2

Abstrakt

Tato bakalářská práce se věnuje návrhu a vývoji front-endového modulu pro systém managementu testovacích dat ve formě webové aplikace. Tato aplikace dovoluje uživatelům prohlížení, vytváření a upravování systémových modelů založených na orientovaných grafech. Dále aplikace poskytuje uživatelské rozhraní pro interakci s funkcemi implementovanými backendovým modulem systému. Mezi tyto funkce patří ukládání, sdílení a přístup ke grafům na serveru, importování/exportování grafů do souborů a generování testovacích scénářů pro dané grafy.

Text této práce také popisuje potřebnou terminologii související s model-based testingem a soustředí se na vysvětlení orientovaných grafů jako modelů testovaných systémů spolu s testovacími scénáři a kritérii pokrytí.

Implementace webové aplikace je realizována pomocí JavaScript a React framworku spolu s knihovnou JointJs pro vytváření diagramů a komponentovou knihovnou Mantine pro stavbu uživatelského rozhraní.

Klíčová slova: Model-based Testing, Path-based Testing, Vizualizace Orientovaného Grafu, Vývoj Front-end aplikace v React

Překlad názvu: Front-end část k systému pro správu dat pro procesní testování

Contents

1 Introduction 1

2 Terminology 3
2.1 Model-Based testing 3
2.2 Test case 5
2.3 Coverage criteria 5

3 Analysis 7
3.1 Application requirements 7
3.2 Use-case diagram 10
3.3 Chosen technologies 11
3.4 Deployment 15
3.5 User interface design 16

4 Implementation 19
4.1 Local environment 19
4.2 Project setup 20
4.3 React component tree 24
4.4 App component 25
4.5 Graph model 29
4.6 Editor component 33
4.7 Communication with back-end . 40
4.8 Header components 41
4.9 Left column components 46
4.10 Righ column components 49
4.11 Benchmark creation 60

5 Quality assurance 61
5.1 End-To-End tests 61
5.2 Exploratory testing 62
5.3 Performance testing 62

6 Conclusion 65

Bibliography 67

A Basic user manual 69
A.1 Register and login 69
A.2 Graph creation and saving 70
A.3 Generating test cases 72

B Test scenarios 75

C Created benchmarks 83

Figures

2.1 Login process based on the developed system . . . 4
2.2 The graph model representing the process from Figure 2.1 modelled using the Oxygen platform[9] . . . 4

3.1 Use case diagram visualizing functions available to the users . . . 10
3.2 NPM - Number of framework downloads in past 5 years[6] . . . 12
3.3 Website deployment diagram . . . 15
3.4 UI mockup . . . 16

4.1 Project structure . . . 21
4.2 Application's component tree . . . 24
4.3 App component diagram . . . 25
4.4 UI defined by the *App* component 28
4.5 Graph model link diagram . . . 29
4.6 Node visual attributes application example . . . 33
4.7 JointJs architecture [10] . . . 34
4.8 Create node sequence diagram . . 37
4.9 Node movement event sequence diagram . . . 37
4.10 Edge creation sequence diagram 38
4.11 UI defined by the *Changelog* component . . . 42
4.12 Menu defined by the *OptionsButton* component . . . 43
4.13 Form for generating graphs defined by the *OptionsButton* component . . . 44
4.14 *LoginBox* modal tabs . . . 45
4.15 Left column UI . . . 46
4.16 GraphSelector component diagram . . . 47
4.17 *FolderGroup* component diagram 48
4.18 InfoBox component diagram . . 50
4.19 GraphInfo component diagram 52
4.20 UI defined by the *GraphInfo* component . . . 53
4.21 UI defined by the *ShareModal* . 54
4.22 NodeInfo and EdgeInfo component diagram . . . 55
4.23 UI defined by *NodeInfo*/*EdgeInfo* 56
4.24 UI defined by *CategoriesList* . . 57
4.25 UI defined by *AttributeInfo* . . . 58
4.26 Example diagram created inside application based on Figure 1.14 from paper `https://www.sciencedirect.com/science/article/pii/B9780128183731000019`. . . . 60

A.1 Registration menu . . . 69
A.2 Application UI with blank graph 70
A.3 Hovered over node . . . 70
A.4 Custom tab of the left application column with the New Folder button highlighted . . . 71
A.5 The Graph Info UI . . . 72
A.6 Generated test case . . . 73

C.1 System model based on `https://ieeexplore.ieee.org/abstract/document/9079344`. . . . 83
C.2 System model based on `https://ieeexplore.ieee.org/abstract/document/9079344`. . . . 84
C.3 System model based on `https://eprints.unmer.ac.id/id/eprint/2843/1/1.%20Jurnal.pdf`. 85
C.4 System model based on `https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/`. 86
C.5 System model based on `https://www.sciencedirect.com/science/article/pii/B9780128183731000019`. . . . 87
C.6 System model based on `https://link.springer.com/article/10.1007/s10586-021-03291-7#Sec10`. 88
C.7 System model based on `https://ieeexplore.ieee.org/abstract/document/9137867`. . . . 89
C.8 System model based on `https://www.hindawi.com/journals/scn/2021/9928254/`. . . 90
C.9 System model based on `https://koreascience.kr/article/JAKO202010163509620.pdf`. . . . 91

C.10 System model based on https://onlinelibrary.wiley.com/doi/full/10.1002/ett.4112. 92

Tables

Chapter 1

Introduction

In the modern world, software plays a critical role in almost all aspects of human society and consequently mistakes in software have the potential to lead to catastrophic results from financial losses for businesses to human fatalities. For this reason, software testing is an essential part of the software development process, and the software industry and researchers constantly strive to improve and optimise the methods and tools used for testing. Model-based testing is one approach to system testing in which the tested system is abstracted into a graphical representation of its behaviour, enabling the generation of test cases through automated tools. With the development of additional tools, there is a requirement for a system that enables the comparison of these tools using publicly accessible data to assess their relative effectiveness and to facilitate the creation of customised test models for these objectives.

The main objective of this thesis is the creation of a front-end module in the form of a web application for such a system. The module should enable the creation, viewing, and editing of system models based on directed graphs, and should provide a user interface for interacting and using the features provided by the back-end module of the system, which was implemented as part of another student's thesis. These features include account management, import/export of graph files, saving graphs on the server, generating graphs based on user-defined parameters, and test case generation for system models, which should also be able to be highlighted on graph of the system to which they belong. The additional objectives for this thesis are the creation of automated end-to-end tests to ensure the correct functionality of the application and the creation of 10 public models based on 10 test data samples identified by background research.

This thesis is divided into three chapters that describe the module development process and the creation of public models. The first chapter covers the necessary terminology related to model-based and path-based testing. The second chapter focusses on the analysis of the application requirements and the design of the application. The third chapter describes the implementation of the module and documents the implemented applications. The fourth chapter describes the methods used to test the correct behaviour of the application, and finally, the fifth chapter covers the example public system

models created as part of this thesis.

Chapter 2

Terminology

2.1 Model-Based testing

Model-based testing, also known as MBT, is the practice of designing software tests from an abstract model that represent some aspects of the tested system.[4]

2.1.1 Directed Graphs

Most of the time, a model of the tested software is defined as a directed graph $G = (N, E)$, where N is a set of nodes, $N \neq \emptyset$ and E is a set of edges and a subset of $N \times N$. A single start node belonging to N is defined along with a non-empty set of end nodes.[24]

Figure 2.1 illustrates a simple log-in process based on the one of the system developed as part of this thesis. This process is then modelled by a directed graph in Figure 2.2. The nodes in the graph represent the decision points inside the system and the points where the branches converge. The start node is distinguished from other nodes, and the end node is represented by the one that has no outgoing edges.

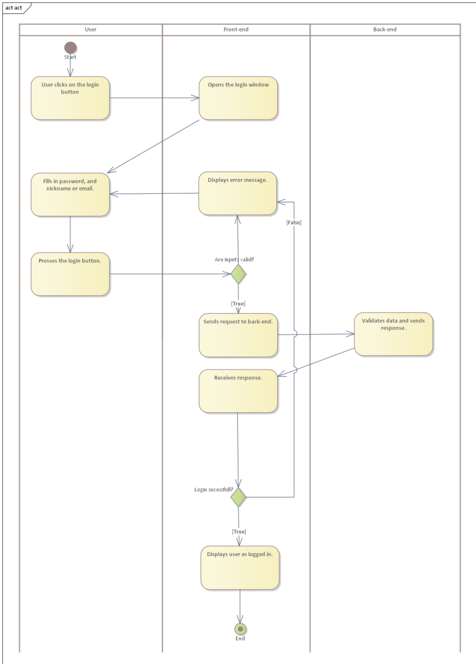

Figure 2.1: Login process based on the developed system

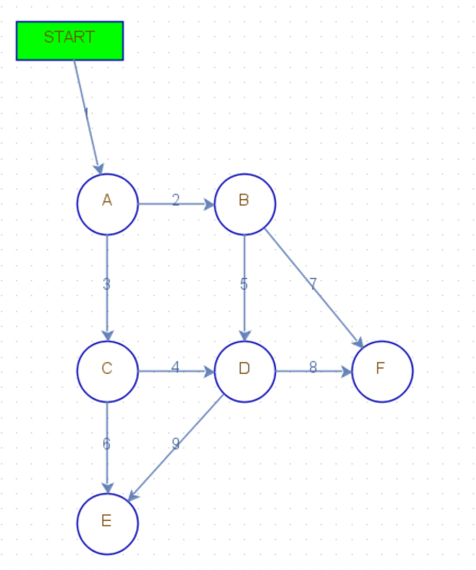

Figure 2.2: The graph model representing the process from Figure 2.1 modelled using the Oxygen platform[9]

2.2 Test case

A path-based test case can be defined as a sequence of nodes $n_1, n_2, ..., n_n$, along with a sequence of edges $e_1, e_2, ..., e_{n-1}$, where $e_i = (n_i, n_{i+1})$, n_1 is the start node of the graph and n_n is one of its end nodes.

2.3 Coverage criteria

An essential problem with software testing is the large number of possible inputs even for a small program. It is impossible to test all inputs and states that a program could enter, as to all practical purposes the input space is infinite. Thus, a tester's goal could be to find the fewest number of tests that will reveal the most problems. Coverage criteria provide a structured and practical way to search the input space, and satisfying a coverage criterion gives a certain amount of confidence that the input space is covered effectively.[4]

Some of the most common coverage criteria for path-based testing are *Edge coverage*, where in a set of test cases, each edge must be covered at least once or *Node coverage*, where the same applies but for nodes. These criteria are suitable for low-intensity tests. For higher intensity tests, the *Edge-Pair coverage* criterion is used, where a set of test cases must contain each possible pair of edges.[24]

Alternatively, the *Test Depth Level* coverage criterion can be used. A *Test Depth Level* equal to x is satisfied when for all nodes n a set of test cases contains all possible paths starting with an edge incoming to a node n, followed by a sequence of $x - 1$ edges outgoing from node n.[24]

Chapter 3
Analysis

Before the implementation of the application, it is crucial to analyse the requirements, choose the appropriate technologies, and design the application and its user interface.

3.1 Application requirements

The main requirement of the project is to create a front-end module for a test data management system. The module shall have the form of a web application accessible via a Web browser over the Internet.

The established name for this application is CPT Manager, which is also used to reference it through this thesis. Here is a comprehensive list of the functional requirements of the application.

Model visualisation

The core feature of the web application is the ability to visualise the models of systems using directed graphs.

The application shall be able to render the nodes of the directed graphs along with the edges connecting them and visually distinguish the start node from the regular ones. In addition to visually displaying the graph, the application shall also offer information pertaining to the model, such as its name, description, and owner.

Model editing

In addition to displaying current models, the application shall enable users to create and modify their system models through an interactive editor.

The users shall be able to:

- Add new nodes including a start node.
- Drag nodes across the graph to change their positions.
- Edit nodes and edges properties.
- Create edges by connecting existing nodes.

- Delete nodes along with connected edges from the graph.
- Delete edges from the graph.
- Customize the data of elements and graph itself.

The application shall also restrict the user from creating more than one start node and creating edges with the start node as the target.
When creating a new node, it shall be given a unique name according to the rule of the lowest alphabetically available string. For example: If nodes "A" and "B" exist, then the next node will be "C". If "A" and "C" exist, then the next node will be "B". If all single-letter strings are taken, the next one will be "AA".

Data and visual attributes for graph elements

Users shall be able to assign data attributes to nodes and edges of the graph. These attributes shall function as key-value pairs tied to these elements and be used by test case generating algorithms.

Graph elements also have a set of predefined visual attributes that change how the elements look inside the editor.

It shall be possible to configure these attributes for each element individually or by including an element in a group. These groups shall apply attributes defined inside them to the assigned elements and a mechanism shall be defined to resolve conflicts of overlapping attributes based on user-defined priorities.

Test case generation and visualisation

Users shall be able to generate test cases for their created models using algorithms provided by the back-end module. The requirements for generating test cases shall change depending on the user login status. An anonymous user shall be able to generate test cases for the currently loaded graph, but the test cases shall only exist locally until a change in the graph occurs. For logged-in users, the requirement of generating test cases shall be that the graph needs to be saved on the server, as during generation the test case is automatically stored on the server and can be recalled later.

The application shall also offer a method to visually emphasise specific scenario paths within the displayed graph.

Publicly accessible models

The application interface shall offer an option to make a stored model publicly available, allowing access for other users. A list of public models shall be presented to all users who shall have the ability to view but not modify these models.

User specific access

Along with the option to share graphs publicly, the application shall allow sharing of the stored graphs with specific users, who shall be able to view the graph.

User-specific workspace

The application shall provide a user with the ability to register and then log in and out of their account. The purpose of this log-in is to allow users to store and access their graphs on the server.

The features provided to logged in users shall be:

- Create, rename and delete folders.
- Save the currently active model.
- Load back saved models.
- Set models as public, allowing other users to also see and view them, or set public models back to private.
- Delete saved models.
- View previously generated test cases.

Anonymous mode

When not logged in, the user shall still have access to a limited set of features. They shall be able to browse and open public graphs, create and edit their graphs, and generate non-permanent test cases. They shall not be able to save graphs on the server, but shall have the ability to export them to a file and import exported files back into the editor.

Importing and exporting

The application shall provide a way for the user to export the currently opened graph into a file and then import the exported file back into the editor. An imported file shall become a newly created graph with the same contents as the graph inside the file.

Generating new graph

Users shall have the ability to generate a new graph using a generator that is supplied by the back-end module, by filling out a form with the necessary parameters for the generator.

3.2 Use-case diagram

The use-case diagram visualises and describes the application's requirements from the point of view of external users. One use-case element represents a single interaction available to the user of the application based on their permissions.[3]
Different types of users are represented by two actors depicted in Figure 3.1. The first actor is an anonymous user. The second actor extends the first actor and represents a signed-in user, who is granted additional permissions related to workspaces and storage of the graphs.

Figure 3.1: Use case diagram visualizing functions available to the users

3.3 Chosen technologies

An essential step during the design process of the application and before its implementation is to decide on the right technologies on which it will be built.

3.3.1 Framework

In the past, websites and web applications were mostly rendered from the server. A user would visit a URL in a browser and request all associated HTML, CSS and JavaScript files from a web server. The Web applications were mostly structured by the returned HTML and CSS files, and only a small amount of JavaScript code was used to make interactions possible. All crucial functionality was performed by the server, while the client only rendered the returned page.[1]

In modern JavaScript web applications, the focus is shifted from the server to the client, and single-paged applications have become increasingly popular. In this approach, only a minimal HTML file and an associated JavaScript file are downloaded. All rendering and interaction are handled by the JavaScript file locally.[1]

Creating a modern single page application with HTML, CSS and JavaScript alone would be challenging. With a standard website, every time the data of the application changes, the DOM needs to be updated. Updating the DOM manually would make the application quite verbose and difficult to manage, slowing down development, and making it difficult to build complex web applications. JavaScript UI frameworks aim to solve this problem, usually by allowing developers to describe the UI inside the code and updating the DOM behind the scenes[5].

Framework choice

There are many different front-end frameworks, each with its advantages and disadvantages. The three most popular frameworks on the market are Angular, Vue.js, and React. Of these three frameworks, Angular is the one with the steepest learning curve and is suited for bigger projects. That is why the final choice was between Vue.js and React. Although Vue.js is the most modern out of these frameworks and is easiest to learn, it lacks the community support and popularity which developed around React. In the end, React's third-party library availability and its wide use in the job market made it the preferred choice to develop this project.[2]

The difference between the popularity of different frameworks can be seen in Figure 3.2.

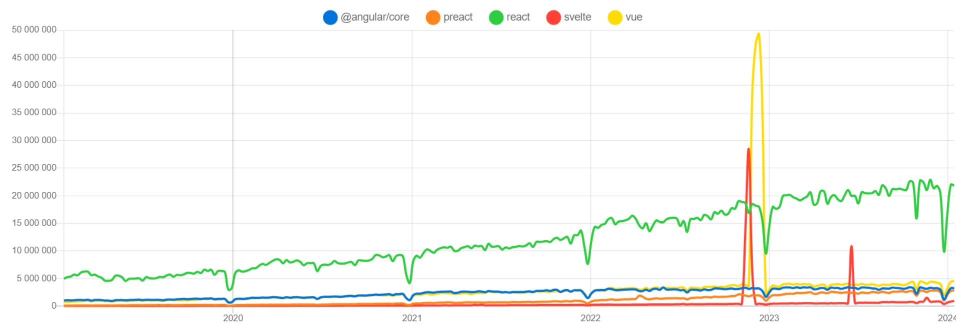

Figure 3.2: NPM - Number of framework downloads in past 5 years[6]

React

Officially React is not a framework as it does not force developers to structure their project in a certain way, but only a library and is not exclusive to web applications. What is referred to as the React "framework" in web development is React in combination with ReactDOM. React can also be used to develop mobile applications with React Native.[7]

React describes the UI using reusable components. The most common types of component in React are functional components, which are defined as a function that is run every time the component is rendered.

The component functions return an HTML-like markup called JSX, which allows one to describe the site's structure inside the JavaScript code and can also include other defined components to build the component's UI.

Components can also have information passed down through their JSX using props.[1]

To add functionality and interactivity to React components, functions called hooks that control the components behaviour. Some of the most commonly used hooks are the following:

- *State hook* - The state hook holds a value inside the component that is persistent across renders, its value affects the components appearance. When this value is updated, a component re-render is automatically triggered.[1]
- *Context look* - The context Hook lets a component subscribe to information from a parent component without the information needed to be passed as a prop. When the information changes, a re-render is triggered inside the subscribed component.[7]
- *Reference hook* - The reference hook allows mutable data to be added to a component, which is shared between renders and does not affect the appearance of the component.[1]
- *Effect hook* - The effect hook allows control of the life cycle of the components. The use effect hook is used to trigger code that interacts with parts of the application and components outside of the Reacts domain or

to execute actions based on specific changes inside the component. An effect can be set to run on every render, when a component is mounted or dismounted, and when a prop or state value changes.[1]

3.3.2 Other libraries

Using the framework itself is not enough to satisfy the application requirements, as developing other necessary technologies from scratch would prove unrealistic. This section lists the other libraries needed to implement the project successfully.

JointJs

The main purpose of the application is to view and edit models in the form of directed graphs. Implementing this functionality from scratch would be impractical, so choosing a good diagramming library is vital. The library chosen for this project is JointJs[8]. It is a very robust modern JavaScript library offering a free open-source version, which perfectly serves the needs of this project.

Vite

Vite is a development tool for modern web applications. It consists of two main parts. The first is a dev server, which offers the developer a way to immediately see the changes to the application during coding without the need to even restart the server after a change in the code thanks to its "Hot Module Replacement". The second is a build command that outputs an optimised static webpage ready to be deployed in production.[13] Vite also offers templates to quickly create a web application project using a wide variety of frameworks.

Mantine

Mantine[11] is an open source React component library. This library was chosen because using predefined components instead of custom ones significantly speeds up development and reduces the risk of bugs caused by unexpected behaviour. Another reason to choose a component library such as Mantine is that it offers a way to build a modern-looking UI out of the box without the need to define CSS styles. Mantine also offers extensions for managing notifications, modals, and forms making it the perfect library to create a functional app quickly.

Tabler Icons

Tabler Icons[12] is an open-source icon library that offers a huge collection of modern-looking SVG icons. It also offers a React plugin, which enables the use of these icons as React components when building the application UI.

Cypress

Cypress[15] is a front-end testing tool that allows one to write end-to-end tests using JavaScript. It is easy to integrate into a web application running locally inside a development environment, provides a simple way to code complex tests, and ships with a user-friendly UI.

3.4 Deployment

The web application will be deployed on an HTTP Web server. When a user wants to access the website from a browser, a request is sent to the server, which will then respond with the static website content. Once running in the browser, the application will request data and communicate with a separately developed back-end module of the application through a REST API. The application deployment diagram can be seen in Figure 3.3

At the time of this thesis submission, the mutually confirmed URL on which the application shall be accessible is `https://cpt.fel.cvut.cz/manager/`.

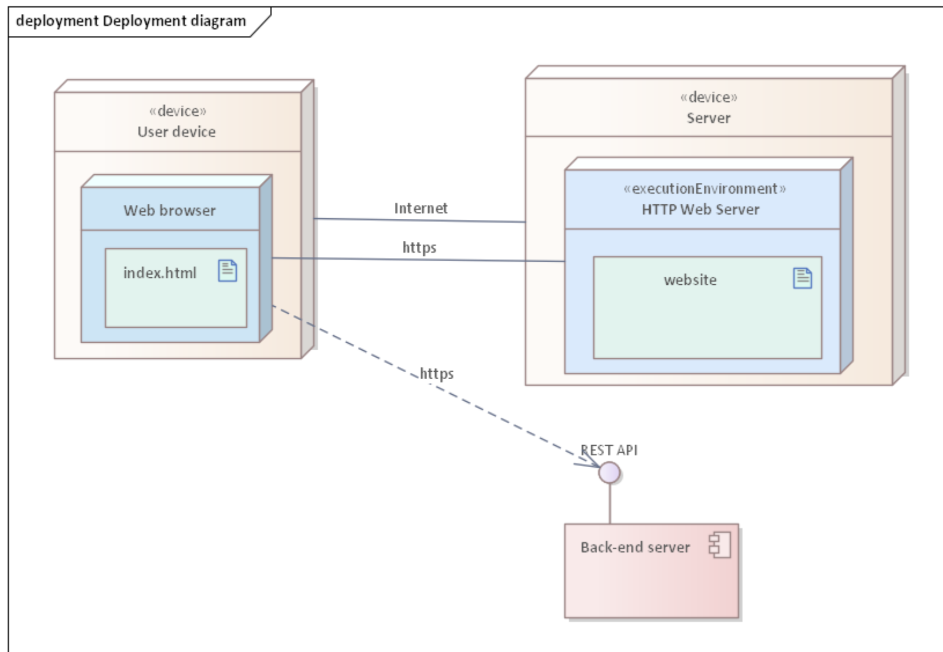

Figure 3.3: Website deployment diagram

3.5 User interface design

Designing the user interface is an essential prerequisite for starting the development of a front-end application. Based on the application requirements and research conducted on existing similar graph editors, a standard three-column layout with a header was chosen for the application.

The left side of the application header will contain access to basic features, such as switching light modes, accessing information about the app itself, and working with files. The right side of the header will be dedicated to account management.

The centre column of the application will contain the core functionality, a canvas on which the current graph is displayed, edited, and individual elements can be selected by the user.

The right column of the application will be used to manage and load the graphs stored on the server. The column itself will present three categories of graphs to the user that are stored within folders. Within the custom category, the user will be able to manage his own workspace and manage his own folders and graphs stored inside of them.

The left column will serve to display and edit additional information about the graph, selected elements, groups, and test cases, which cannot be shown in the middle of the application.

The figure of the described UI is illustrated in Figure 3.4.

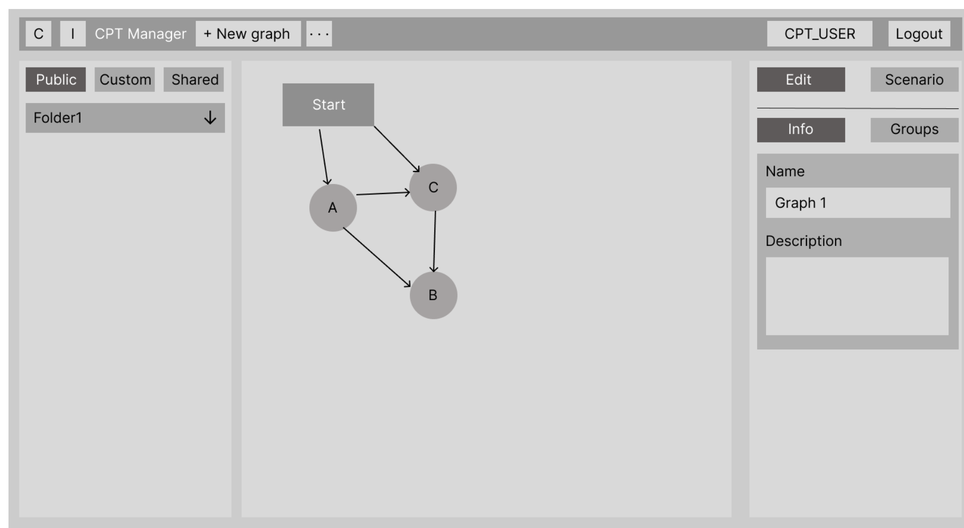

Figure 3.4: UI mockup

User interface modes

Based on the application requirements, there are two important features that the application needs to provide graph editing and test case generation/visualisation. Based on the analysis of these two features, test case viewing and generation should only be allowed when a graph is saved on the server,

otherwise, the graph and test cases could become inconsistent with each other. For example, when a test is generated for a graph whose current state is not saved on the server, a test case would be saved on the server upon generation which does not match the saved graph. If the local graph would never be saved and the graph would be viewed in the future, then the user would get a test case that does not match the graph and cannot be displayed on it.

That is why the application will be divided into two modes. An *Edit* mode and a *Scenario* mode. When in edit mode, the application will allow the user to make changes to the graph. The transition to scenario mode will be locked out until the user saves the graph to the server. Once the graph is saved on the server, the user can transition to the scenario mode. This will ensure that the scenarios are consistent with the graph that is saved on the server and server can properly delete old scenarios upon saving of the graph. When the user transitions to scenario mode, the saved test cases are loaded from the server. In this mode, the user is locked out from editing the graph and can only view or generate test cases. If the user performs any action that changes the current graph, such as loading the new graph or logging out, the application will automatically transition him back into the edit mode.

The described locks will not be present when the user is not logged in, as in this state the application does not allow graphs to be saved on the server and tests are only saved locally. When in this state, once the user leaves the scenario mode, all of the test cases will be deleted, to ensure data consistency.

Chapter 4
Implementation

This chapter focusses on describing how the designed application is implemented and serves as a documentation for the application. A basic manual for how to use the implemented application is attached in the Appendix A.

4.1 Local environment

The first step in implementation is setting up the local environment in which the application can be developed. This section lists the used and required programmes for development.

IDE

The IDE[1] chosen to develop this project is Webstorm, a specialised IDE for JavaScript and web applications.[14] The decision to use WebStorm was made based on previous experience with tools developed by JetBrains.

Version control

During the project's development, a version control system was deployed alongside an online repository to monitor code changes and organise code into distinct branches. Git, the most popular and widely used version control system[16], and a Bitbucket repository were used for this purpose.

Node.js and NPM

NPM and Node.js programmes need to be installed for this project. NPM servers as a package manager for installing dependencies, while Node.js provides the environment necessary for running JavaScript code outside the browser needed by the development tools used in the project. Both of these programmes are bundled together. [17]

[1] Integrated Developer Environment

4.2 Project setup

Creating project

Once all dependencies have been installed, the initial project can be created using NPM and Vite. This is done by typing **npm create vite@latest** into a command line directed to the folder where the project should be created.

Afterward, a menu will appear asking to select a framework.

```
    ? Select a framework: >> - Use arrow-keys. Return to submit.
>   Vanilla
    Vue
    React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Others
```

Then the script will ask for a variant.

```
? Select a variant:>> - Use arrow-keys. Return to submit.
    TypeScript
    TypeScript + SWC
    JavaScript
>   JavaScript + SWC
    Remix
```

The options chosen for this project are React and JavaScript + SWC.

After the setup is completed, a Vite project should appear in the project folder, as seen in Figure 4.1. After creating the project, the command **npm install** needs to be run inside the project folder to install all the necessary dependencies. Finally, a dev server can be started by running the command **npm run dev**. The server can then be accessed through a web browser at the address localhost:5173, where the developed application is hosted, and changes are reflected in real time without the need to refresh the page. By default, Vite creates the project with a simple React app template, which can be used as a starting point for the development of an application.[13]

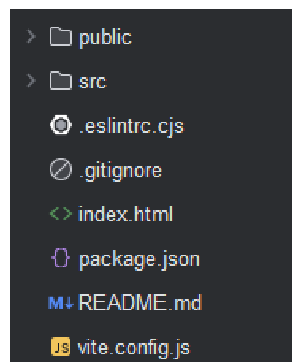

Figure 4.1: Project structure

4.2.1 Installing and setting up dependencies

Mantine UI Library

After creating the project, the dependencies for Mantine need to be added. This is done by running the command **npm install** followed by the list of dependencies to be installed.

List of dependencies for the Mantine UI library:

- @mantine/core
- @mantine/hooks
- @mantine/form
- @mantine/notifications
- @mantine/dropzone
- @mantine/modals

Mantine also requires installing PostCSS plugins and a PostCss Mantine preset by running the command **npm install –save-dev postcss postcss-preset-mantine postcss-simple-vars**.

After this, a *postcss.config.cjs* file needs to be created inside the project root folder to configure postcss.

postcss.config.cjs content:

```
module.exports = {
  plugins: {
    'postcss-preset-mantine': {},
    'postcss-simple-vars': {
      variables: {
        'mantine-breakpoint-xs': '36em',
        'mantine-breakpoint-sm': '48em',
        'mantine-breakpoint-md': '62em',
        'mantine-breakpoint-lg': '75em',
        'mantine-breakpoint-xl': '88em',
      },
    },
  },
};
```

Finally, within the file App.jsx, which is the application's root component, all of the jsx needs to be wrapped inside of the required Mantine provider components and the styles.css file must be imported. [11]
App.jsx content:

```
import '@mantine/core/styles.css';

...
export default function App() {

...

return (
        <MantineProvider defaultColorScheme="light">
             <ModalsProvider>
                  <Notifications position="top-center"/>
                   ...content
             </ModalsProvider>
        </MantineProvider>
    );
}
```

JointJs

JointJs is added to the application by adding the necessary script tag to the body of the index.html file.

```
<script src="https://cdn.jsdelivr.net/npm/@joint/core@4.0.1/dist/joint.js"></script>
```

This will allow the JointJs library to be accessed within the JavaScript code, as it will be added to the global scope of the application.[10]

Other dependencies

The rest of the dependencies that need to be installed are underscore and Tabler icons. Underscore is a library that provides useful utility functions for JavaScript.[23] These packages are installed by running the command **npm install @tabler/icons-react underscore**.

4.3 React component tree

In React, components are constructed by combining other React components, forming a hierarchical structure where the entire user interface of a single page application is encapsulated within a single top-level component. This hierarchical structure can be described as a tree with the root node being the top-level component of the application. The components nested within another one are considered children of the parent component.[1]

Moreover, this tree structure also defines how data flows within a standard react application, as parent components can pass down their data to children components. Additionally, React optimises rendering by re-rendering only the component and its children affected by a state change. This enhances performance by minimising unnecessary re-renders.[18]

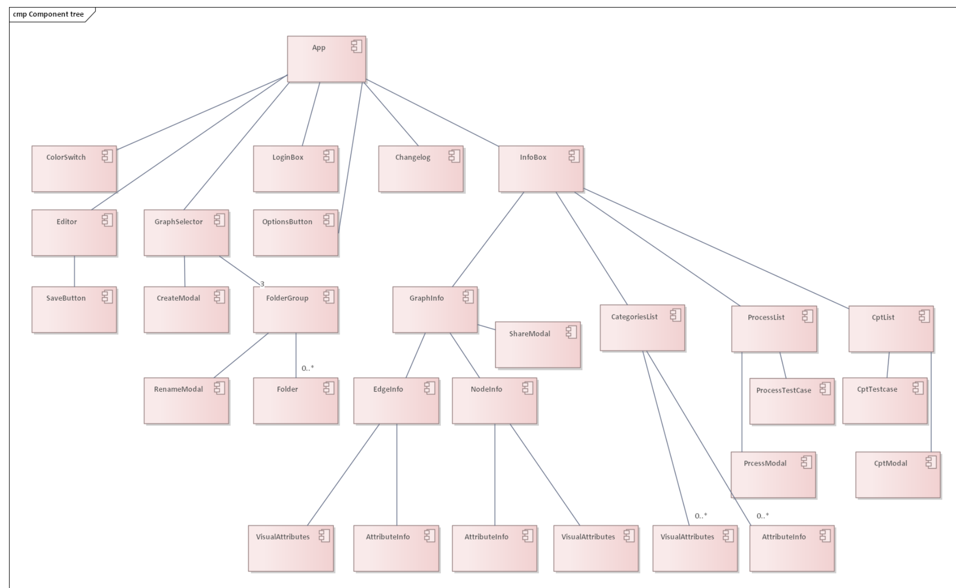

Figure 4.2: Application's component tree

Figure 4.2 illustrates the CPT Manager component tree, with the App being the root component. Certain relationships between components carry multiplicities, indicating that a parent component may contain multiple instances of the child component.

Furthermore, the diagram focusses only on components defined within the project, as displaying components from UI libraries would unnecessarily clutter the diagram.

4.4 App component

The app component serves as the root of the CPT Manager and encapsulates the entire UI of the application. As the root component, it holds most of the application state and implements critical functions for its modification. Figure 4.3 illustrates the structure, dependencies, state, and properties of the component that are inherited by its child components.

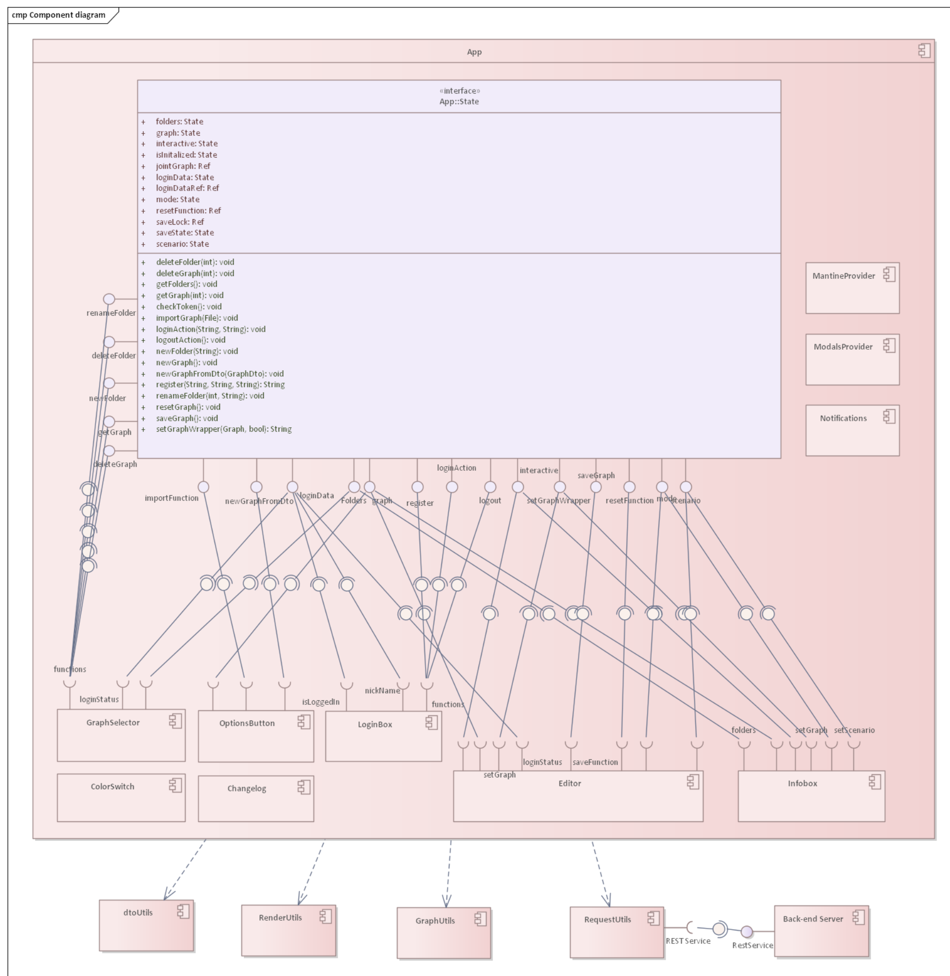

Figure 4.3: App component diagram

States

List of useState hooks within the *App* component:

- *folders* - Holds three lists of folders retrieved from the server, which are available to be viewed by the user, *publicFolders*, *privateFolders* and *sharedFolders*. The folders themselves hold a list of graphs saved inside the folder on the server.
- *graph* - Value containing the custom graph model loaded inside the application.

- *interactive* - Boolean representing whether the user should be allowed to edit the current graph.
- *isInitalized* - Boolean indicating the completion of the initialisation process. If set to false, the application displays only a loading circle.
- *logindata* - An object that holds data about the current logged-in user or indicates that no user is currently logged-in.
- *mode* - Defines which of the two modes the application is in. *Edit* mode, which allows the user to modify the graph or *scenario* mode, where a user can view and generate new test cases.
- *saveState* - Represents the status of the current graph inside the database. Can be *saved*, *saving* or *unsaved*.
- *scenario* - Object representing the current path on the graph to be displayed.

Refs

List of useRef hooks within the *App* component:

- *jointGraph* - Graph object from the JointJs library, which depicts the graph displayed on the screen. This object is connected to a paper, another JointJs object that represents the canvas area where the graph is shown.
- *loginDataRef* - Mutable object that stores the same values as the loginData state and updates through an effect. This is done to solve state staleness, which can occur inside a closure function defined in React components. State staleness occurs when a function uses a state value from the one it was defined instead of the current one. The solution is to use a reference hook, to create a value which is shared across renders and access to needed data trough it.[20]
- *resetFunction* - Mutable value holding the function resetting the paper to 0,0 position and reset scaling. Upon initialisation, this reference is passed to the editor, which inserts the function into the reference, allowing it to be called from the App component.
- *saveLock* - Boolean that prevents certain actions from happening while the graph is being saved, as the app could enter an inconsistent state and corrupt data.

Effects

List of useEffect hooks used inside the *App* component:

- Startup effect - Effect activating when the application is first rendered, getting the login data from *localStorage* and if a user is supposed to be logged in, checking with the back-end whether the session is still valid. After starting the app, it sets the *isInitiated* state to *true* allowing the application to render the full UI.
- *graph* effect - Effect activates whenever the graph state is updated, switching the mode to back to edit.
- *loginData* effect - Effect activated when the *loginData* state is set, updating the *LoginDataRef* to the current value of *loginData*.

Functions

List of functions implemented inside the *App* component:

- *deleteFolder* - Function responsible for deleting the requested folder.
- *deleteGraph* - Function to delete the requested graph.
- *getFolders* - Function that updates the shown folders to the current state on the back-end.
- *requestGraph* - Used to update the loaded graph to a graph with a specific id stored on the server.
- *checkToken* - Checks whether the login token is still active, attempts to refresh the token ID if it is still possible, or otherwise logs out and returns the result.
- *importGraph* - Function to set the loaded graph from the given file.
- *loginAction* - Function to log in the user with given data returns the result.
- *logoutAction* - Function to log out the currently logged-in user.
- *newFolder* - Function to create a new folder for the user.
- *newGraph* - Function that sets the graph as a new blank graph.
- *newGraphFromDto* - Function that sets a new graph based on the graph of a given DTO (Data transfer object).
- register - Function to register a new user with given values, returns the result.
- *renameFolder* - Renames the given folder to the new name.
- *resetGraph* - Sets a new blank graph as a loaded graph and resets the position inside the editor.
- *saveGraph* - Saves the currently loaded graph on the server.
- *setGraphWrapper* - Function that wraps the *setGraph* function provided by React's state hook. Enhances the function to modify the save state of the graph when a change occurs, to reflect that the graph model no longer corresponds to the one stored on the server.

Defined UI

Figure 4.4 illustrates the user interface as defined by the *App* component, which sets the layout for the entire page and the placement of its child components. The upper part features the header. The left side of the header hosts the *ColorSwitch* and *Changelog* components. Adjacent to these is the title of the page, to the right of which lies the new graph button and the *OptionsButton* component. The *LoginBox* is located on the left side of the header.

The primary section of the page is divided into three columns, with the largest column located in the centre. The left column houses the *GraphSelector* component, the central column holds the *Editor*, and the right column contains the *InfoBox* above which is the mode switch defined by the *App* component.

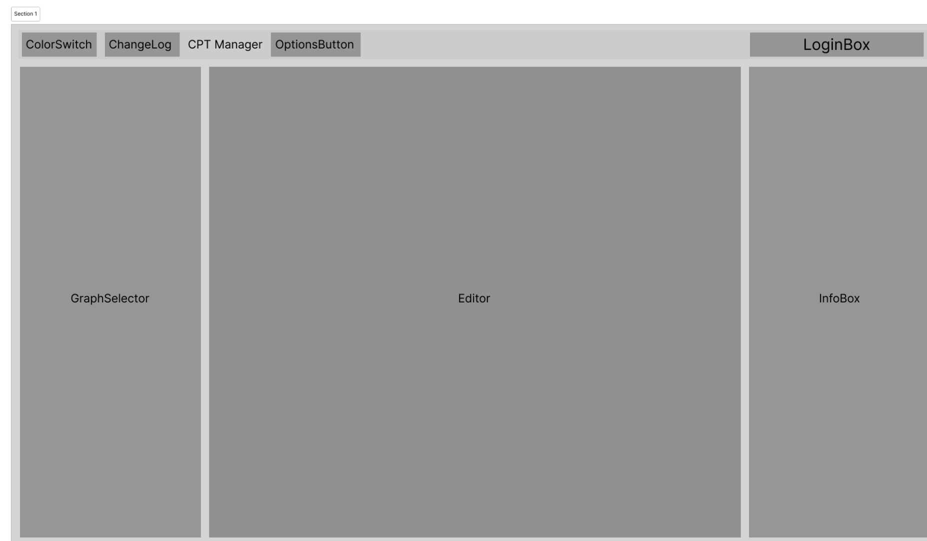

Figure 4.4: UI defined by the *App* component

4.5 Graph model

The graph model of the application is constructed using the DTO (Data Transfer Object) for back-end communication, supplemented with additional values needed by the front-end. Initially, the model incorporates the graph owner's name, a requirement for the UI display. Upon fetching a graph, the server provides only the owner's ID, necessitating a separate request for the owner's name. Further enhancements include features for selecting elements within the model, allowing users to choose and modify elements. The model tracks the selected element through two attributes: *selectedElement* and *selectedType*, which identify the chosen element and its type, either a node or an edge. Additionally, the model integrates a reference to the JointJs graph model, which dictates what is rendered on the screen.

4.5.1 JointJs Graph

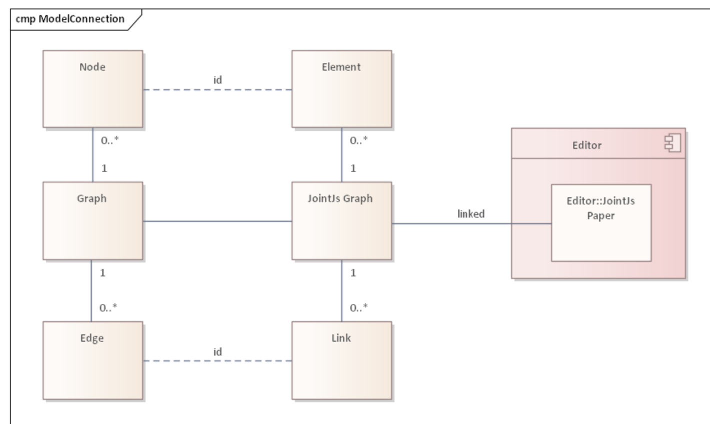

Figure 4.5: Graph model link diagram

Figure 4.5 demonstrates the connection between the JointJs graph and the custom graph models. Using two graph models within the application to represent the same graph for distinct purposes might not seem ideal, but is required. This necessity arises because react components must read data from the graph, and any modifications to the graph require a re-render of these components. This process is done by defining the graph as a React state within the main *App* component and passing it to other components. A challenge arises because react states should work with immutable objects and be updated by creating a new object and setting it as the new state value, whereas the JointJs graph updates its state internally and works as a mutable object. Consequently, there is a need for a custom graph model that coexists with the JointJs graph, managing the graph data, while the JointJs model is used for displaying the visual elements on screen. It is also important to note that only one JointJs graph is maintained throughout the application's life, as once it is linked to a paper it cannot be detached. Therefore, the lifecycle of the JointJs graph is independent of the custom graph model's lifecycle. When a new graph is initiated, the JointJs model is transferred from the old to the new one and updated accordingly.

JointJs terminology

In Section 2.1.1, it was established that the elements of directed graphs are called nodes and edges, and this terminology is also employed within the application. However, it is important to note that within JointJs, the terminology differs. In JointJs, nodes fall into the element category, while the lines connecting the elements are called links. Together, they form the cells of the graph within the JointJs library.

It is crucial to be aware of this distinction, as both sets of terminology are utilised in this paper based on the context in which they are discussed.

Custom shapes

JointJs comes with a variety of built-in shapes for links and elements, but it is advantageous to define custom shapes for the application's purposes, as they will be easier to work with. That is because the shapes will automatically possess the desired visual attributes and it will be easier to determine which element inside our custom model they represent.

The shapes for the application are defined within the function contained in *defineShapes* JavaScript file. This file is run by the editor on its mount, adding the custom shapes to the default namespace under the editorShapes category.

The file defines four custom shapes in total:

- Start - Custom shape based on the JointJs rectangle. The shape represents the graph's start node. Visually by default, it is a blue rectangle with the text **Start**.
- Node - Shape based on the JointJs circle, representing the normal nodes of the graph. By default, it appears as a grey circle with the name of the node it represents in the centre.
- Edge - A custom JointJs link representing the edges of the graph. It appears as a white arrow, pointing from its source to its target and has a white label displaying its ID.
- Label - A custom JointJs element, which serves as a child element for nodes displaying the label defined inside the element's visual attributes.

As illustrated in Figure 4.5, the shapes are linked to their counterparts from the custom graph by the IDs of the elements, which are set as custom props inside the JointJs shapes when they are created.

Modifying the graph

Given the interconnected nature of the two graph models, any alteration in one requires adjustments in the other to maintain their consistency. Consequently, it is advantageous to develop utility objects to manage these modifications rather than embed them directly within the components that interact with the graph. This approach prevents components that only handle the custom model's data from engaging with the JointJs graph, preserving the system's modularity for potential future enhancements.

DtoUtils

The first utility object of the application is *DtoUtils*, which handles the transition from the application's model to and from the graph DTO used for communication with the back-end.

The functions defined inside the object are:

- *fromDto(jointGraph, graphDTO, ownerName)* - The function takes a received DTO and returns a custom graph model to be set as the application's new graph. Inside the function, a new shallow copy of the received DTO is created. Then the missing values, *ownerName*, *selectedElement*, and *selectedType* are added. Finally, the provided JointJs graph is cleared and updated to reflect the provided graph.
- *toDTO(graph)* - This function returns a copy of the graph with the additional values possessed by the application's graph model removed so that it can be used in a call to the back-end.

GraphUtils

The *GraphUtils* object is the main object used to modify and work with graph models. Most of its functions return a shallow copy of the graph, which can then be set as the new graph state. The functions defined inside the object are:

- *newGraph(ownerid, ownerName, jointGraph)* - Creates new blank graph with given parameters and attached JointJs graph.
- *addStart(x, y, graph)* - Adds the start node to the graph and calls the *renderStart* function from *RenderUtils*.
- *addNode(x, y, graph)* - Same as the previous function, but for regular nodes.
- *moveNode(x, y, graph, id)* - Updates the position of the node with the given ID and calls *moveNode* from *RenderUtils*.
- *deleteNode(id, graph)* - Removes the given node from the graph and calls *deleteNode* from *RenderUtils*.
- *deleteEdge(id, graph)* - Similar to the previous function, but for edges.
- *edgeFromLink(link, graph)* - Function that creates a new edge from the already existing JointJs link and adds the required data to the existing ling. This function is required because when two nodes are connected inside the editor, the link is created before the edge.
- *setSelectedNode(nodeId, graph)* - Sets the *selectedElement* to the node with given ID and *selectedType* to node.
- *setSelectedEdge(edgeId, graph)* - Same as the previous function but with *selectedType* set as edge.
- *nextNodeName(names)* - Function that takes a list of already existing names and returns the alphabetically lowest possible new name.
- *nextId(ids)* - Function that takes a list of existing IDs and returns the lowest unoccupied ID[2].

[2] IDs for each graph element or category are independent of other elements and unique only for a given graph.

- *newViewModifier()* - Returns a default configured *viewModifier*, an object holding an element's visual attributes.
- *isEmpty(graph)* - Returns true if there are nodes, edges, or categories present inside the graph.

RenderUtils

The final utility object for graph modification is *RenderUtils*. Its purpose is to provide functions that modify and work with the JointJs graph. The functions provided by this object are:

- *createJointGraph()* - Creates a new JointJs graph. It is used only at the start-up of the application.
- *renderStart(graph)* - Creates a new start element in the JointJs graph based on the start from the custom model.
- *renderNode(graph, node)* - Creates a new node element in the JointJs graph based on the one passed as an argument.
- *renderEdge(graph, edge)* - Same as the previous function, but for edges.
- *refreshNode(graph, node)* - Redraws the node element with current visual attributes. Used after an element's visual attributes have been modified.
- *refreshEdge(graph, edge)* - Same as the previous function, but for edges.
- *moveNode(graph, node)* - Sets a new position for the node element in the JointJs graph based on its custom graph counterpart.
- *deleteNode(graph, id)* - Deletes the node element from the JointJs graph.
- *deleteEdge(graph, id)* - Deletes the edge element from the JointJs graph.

4.5.2 Visual attributes customisation and groups

The application offers a feature for visually customising graph elements. This customisation can be applied on individual elements or collectively on several elements via groups to ensure uniform visuals.

These visual attributes are stored inside the *visualAttributes* object attached to each element. The values contained by this object are:

- *priority* - The number determining how the visual attributes should render if there is an overlap of attributes for a single element.
- *nodeSize* - Determines the size of the node. can be *small*, *default*, *large*.
- *nodeFillColor* - Determines the colour of a node.
- *nodeOutlineColor* - Determines the colour of the node's border.
- *edgeColor* - Determines the colour of the edge.
- *label* - Text that should be displayed next to the element on the graph.

priority	nodeSize	nodeFillColour	nodeOutlineColour	Label
5	default	red	default	
3	small	default	default	primary node
1	large	default	blue	

Figure 4.6: Node visual attributes application example

Applying visuals

A single element may belong to multiple groups that apply their visuals and can also possess its own visual attributes. To address this issue, a logic is established to decide the final visual attributes applied to the element. Figure 4.6 illustrates the visuals that appear in a node affected by three *visual attributes* entities. The visuals displayed are based on the priority value that the objects carry. For each attribute, a non-default value with the highest priority is chosen. A default value is chosen only if no other value is set for a given attribute.

Labels

The *label* visual attribute presents itself as a floating text box moving relatively with its element. Adding a label to a link is straightforward, as JointJs already has the functionality built-in. For JointJs elements, labels are implemented using the Label shape defined in the section 4.5.1 and the embedding functionality of JointJs, which allows linking the position of the element to another. These elements also have an additional prop called ignore, which signifies that they do not represent nodes, and events triggered by them should be ignored.

4.6 Editor component

The primary role of the application is to offer a user interface for viewing and modifying directed graphs. The *Editor* component manages this role, serving as the central component of the application and facilitating interactions between the user and the displayed graph.

JointJs Paper

The diagramming library chosen for this project, JointJs, splits its graph into two parts, a graph model and a paper view. The graph model contains element and link models and is tied to the paper. The paper object represents the onscreen element and renders the graph by generating views from element models.[10] The architecture is illustrated in Figure 4.7.

Out of the box, the JointJs architecture does not work seamlessly with React. The editor component encompasses the paper and manages its interactions with the rest of the application, incorporating the specialised graph model utilised by CPT Manager, which relies on the React state hook. The actual rendering of the graph is not performed by the component itself; instead, this task is carried out by the components *graphUtils* and *renderUtils*, which facilitate the interaction between the two models.

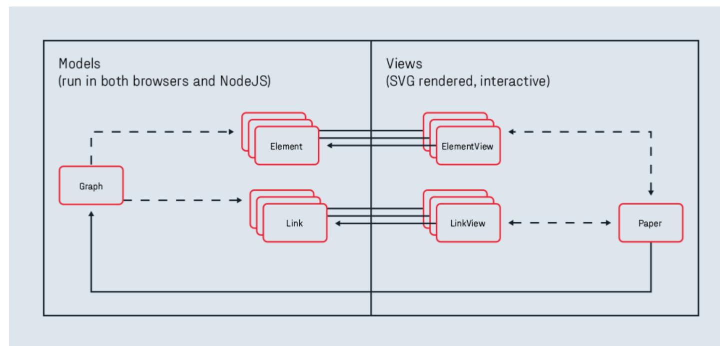

Figure 4.7: JointJs architecture [10]

Props

The props accepted by the editor are:

- *interactive* - Boolean determining if the user can edit or only view the current graph.
- *graph* - The custom model of the currently loaded graph.
- *setGraph* - Function to update the graph model.
- *resetFunction* - A value containing a React reference hook that belongs to the application component. When the paper is set up, the editor component returns a function for resetting the pan and zoom of the paper.
- *loginStatus* - Boolean representing whether the user is logged in.
- *saveFunction* - Function that saves the currently loaded graph to the back-end.
- *saveState* - Value representing whether the current state of the graph is *saved*, *unsaved* or *saving*.
- *mode* - Indicates whether the application is currently in the *edit* or *scenario* mode.
- *scenario* - Path to be displayed on the graph if the application is in *scenario* mode.

Hooks

The hooks used by the component are:

- *colorScheme*: useMantineColorScheme - Hook to change the colour of the paper based on the application's colour scheme.
- *JoinData*: useRef - Mutable object that stores data related to the paper.
- *dragStartPosition*: useRef - Hook holding the position where the paper drag event started.
- *paperElRef*: useRef - Reference to the HTML element of the paper.
- *currentPathRef*: useRef - Reference holding the scenario path currently highlighted on the graph.
- *modeRef* and *graphModelref*: useRef - References holding the same data as the *mode* and *graph* prop for the reason described in *loginDataRef*.4.4

4.6.1 Paper set up and destruction

After the Editor component is created, the JointJs paper must be set inside the component. For interacting with non-React components, the UseEffect hooks are used, which can be set to trigger when a component is updated in a specific way. To set up the paper, an effect is triggered when the *Editor* is first mounted.

A series of actions are performed in this order:

1. *defineShapes()* function is called.
2. The *JointData* reference is updated with an object. The object contains these values: *namespace*, *paper*, *scale*, *toolsView*, *nodeToolsView*, *startToolsView* and *paperIteractivity*. For now, all of these values but the scale are set to null.
3. A div element is created inside the *dia-window* div element defined by the *Editor* component and stored in the *paperElRef*. This will be the element to which the paper will be linked.
4. The *namespace* value of *JoinData* is set to joint.shapes, where custom shapes are defined.
5. The paper object is created. The options for the created paper are:
   - *model*: The JointJs graph model linked to the graph model.
   - *gridSize*: 10
   - *background*: The background colour chosen based on the *colorScheme* hook.
   - *width*: Set to be the same as the width of the *dia-window* div element.
   - *height*: Set to be the same as the height of the *dia-window* div element.
   - *drawGrid*: true
   - *el*: *paperElRef* - Reference to the element, where the paper should be created.
   - *linkPinning*: false - Forbids the creation of links not connected to nodes.
   - *defaultLink*: The custom-defined edge. This will set the custom-defined edge to be used when two nodes are linked together.
6. Translates and scales the paper so that the scale value matches the one in *JointData* and the 0, 0 position of the paper is in the middle of the screen.
7. Sets the reset function.
8. Creation of event listeners interacting with the graph and the paper.

Finally, in the end, the effect returns a function, which is called when the component is to be unmounted. This function will destroy the created paper so that it does not remain on the page when the component is no longer present. Currently, the editor component is always present as part of the application, but it is good practice to destroy anything set up by the component to avoid possible problems and unintended side effects.

4.6.2 Changing the background colour

The editor defines an effect that activates when the *colorScheme* value is updated and calls *paper.drawBackround* with the colour option changed based on the current colour scheme.

4.6.3 Panning and zooming

An ability to pan across the paper is implemented inside the *Editor*. This is done by three event handlers. The first listens for when the pointer grabs a blank space on the paper and records the initial position inside the *dragStartPosition* reference. Then, by moving around, a mouse move event is triggered, translating the paper by the distance between the current and the starting point, but only if the starting coordinate is set. When the mouse is released, a final event is triggered, erasing the starting coordinate.

Zooming is implemented by a scroll wheel event listener. When the event is activated, the scale of the paper is changed by the delta of the event * 0,1. After changing the scale, a translation occurs on the paper to keep the centre point of the graph the same.

Calculating centre

Some operations inside the *Editor* require the coordinates on the paper that correspond to the current centre of the *Editor*. For this a helper function, calcCenter() is defined. The function returns the centre coordinates based on this equation:

$$x = -paper.translate().tx/scale + canvasWidth/2/scale$$

$$y = -paper.translate().ty/scale + canvasWidth/2/scale$$

When called without an argument, the paper translate function returns the distance by which the 0,0 coordinate was moved from its original position (the upper left corner of the paper).

4.6.4 Resizing the paper

By default, the JointJs paper does not change size when the browser window does. This is not the intended behaviour, as the editor should always occupy the same proportions in the middle of the screen. To solve this problem, a function *resizePaper* is defined inside the editor, which sets the dimension of the paper the same as the div element in which it resides. This function is linked to an event listener that triggers when the size of the window changes.

4.6.5 Creating new nodes

Two functions are defined within the editor component to add regular or start nodes to the graph model. These functions are linked to buttons in the editor's upper-right corner. The process of creating a new regular node is illustrated in Figure 4.8. Creating a new start node is almost identical to this process, but with the added constraint that a start node does not exist.

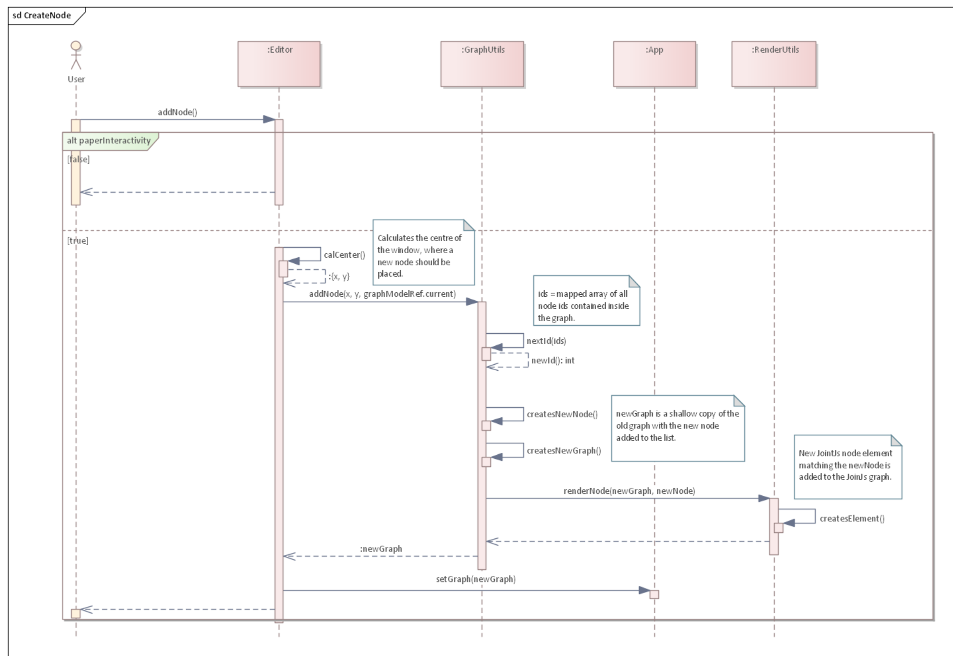

Figure 4.8: Create node sequence diagram

4.6.6 Moving nodes

When a node is moved on the paper, the node's position needs to be updated inside the custom graph model. This is achieved by setting up an event handler on the paper, which listens for when a pointer releases an element on the screen. The process triggered by the vent listener is illustrated in Figure 4.9.

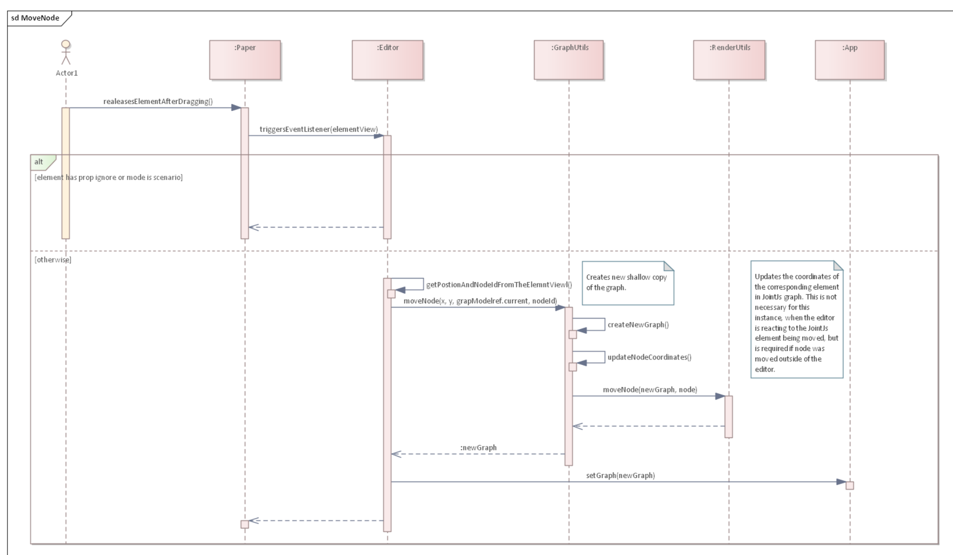

Figure 4.9: Node movement event sequence diagram

4.6.7 Creating edges, deleting edges and deleting nodes

JointJs Tools

JointJs provides a way to add functional UI to elements and links with tools. The *Editor* creates three tools when setting up the paper, a remove and connect tool for elements (nodes), and a remove tool for links (edges). After that, a *ToolsView* needs to be created for each type of element that is responsible for rendering the tools over the element. These views are then stored inside the *JointData* reference. After this step, the tools have not yet been attached to their elements and will not appear on the paper. It is intended for them to show up only when the element is hovered over. This is accomplished by adding two event listeners to links and elements for mouse entering and leaving over them. These event listeners add and remove the tools view from the triggering cell, but only if the graph is interactive (can be edited).

Creating edges

New links are created using the JointJs connect tool by pressing the connect button on a node and dragging a link over to the target node. This creates a link inside the JointJs graph model, and the custom graph model needs to be updated to reflect this change. This is done with an event listener that is activated when a link is connected to a node. The update process is illustrated in Figure 4.10.

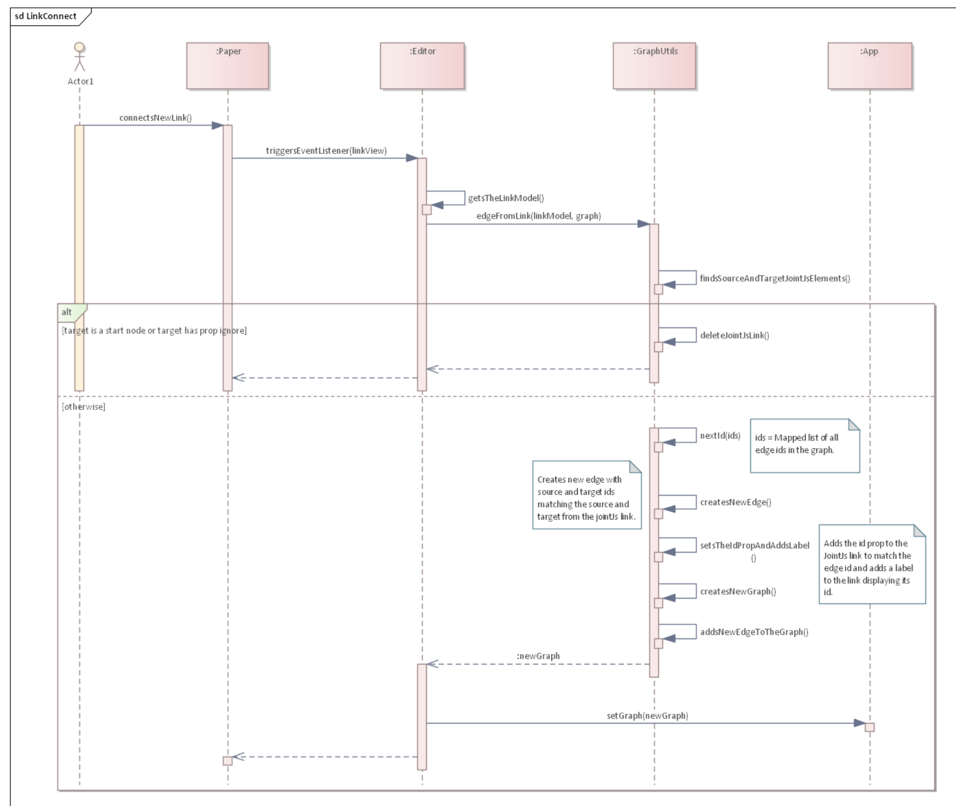

Figure 4.10: Edge creation sequence diagram

Removing nodes and edges

Nodes and edges are removed using the JointJs remove tool, which activates the *deleteNode* or *deletelink* functions defined by the editor. These then request an updated graph from *GraphUtils* by calling the appropriate delete function and setting it as the new graph model. The *GraphUtils* functions delete the element with the given ID from the custom graph model and JointJs model.

4.6.8 Selecting elements

An element inside of the application can be selected, so that its inner data can be edited by the user. This selection is done through the editor. A user can select an element by clicking on it on the paper, and deselect it by clicking on an empty space.

JointJs highlighters

The JointJs library provides a tool for visually highlighting elements called highlighters. These highlighters can be added to an *elementView* with a carry an assigned ID so that it is possible to remove them. Out of the box, JointJs offers a variety of highlighters, with the mask highlighter being used int the *Editor*, which applies a stroke around the highlighted element.

Select implementation

The select functionality is implemented by three event listeners and a helper function. The helper function *removeHighlight* removes the mask from the view of an currently selected element, if such an element exists. An element selection is triggered by either a pointer-click event on a JointJs link or a JointJs element. When the event is triggered, the *GraphUtils* *setSelectedEdge* or *setSelectedNode* function is called, returns a new changed graph, and a highlighter is added to the JointJs link's or element's view and removed from the previously selected one. Because highlighters work with the view and not the JointJs graph, no change is required. For deselecting an element, an event listener is setup for pointer-click on blank space of the paper, which calls the *deselectElement* function from *Graphutils* and removes the highlight from the element's view. It is important to note that the selection only works when the mode is set to *edit*.

4.6.9 Interactivity and application modes

When talking about the interactivity of the application and mainly the *Editor*, what is meant is the ability of the user to make changes to the graph. The interactivity inside the *Editor* depends on two props, *interactive*, which determines whether the user has permission to edit the loaded graph, and *mode*, which should lock the interactivity when set to *scenario*.

Changing interactivity

The interactivity of the *Editor* can be changed by two effects, reacting to the change in the interactive prop or the mode prop. When the props change their value, its value is determined based on the conditions stated before, and it needs to be updated at two places. The first is the paper itself, where it is updated through

the *setInteractivity* function and then inside the *JoinData* ref, which is used by the *Editor* event handlers and functions.

Mode change and path highlighting

When the *mode* prop changes, the *Editor* triggers an effect that makes the necessary changes depending on the mode in which the application is entering. When transitioning from *edit* to *scenario* mode, the effect deselects the selected element, as element selection is not allowed in this mode. If the transition is in the opposite direction, then the effect removes the highlighted path of the scenario prop.

When the editor is in the *scenario* mode, it should highlight the path of the scenario passed as the prop to the component. This is implemented with an effect triggered by the change of the scenario prop, which uses the same JointJs highlighter functionality used for selecting elements.

4.6.10 Saving the graph

The user interface to save the graph to the server is a part of the Editor component. The functionality itself is delegated to a child component called *SaveButton* that serves not only to save the graph, but also as an indicator of the graph's save status. The component accepts three props, *state*, indicating what is the save status of the graph, *editable*, indicating if the graph can be edited by the user and *saveFunction* that the button activates.

The component is visible only when *editable* is set to true, and its appearance is dependent on the *state*. If the graph has not yet been saved to the server, it manifests as a red button which triggers the save function. During the server saving process, a loader is displayed over the button. Once the graph is securely saved on the server, the component is displayed as a non-clickable green icon.

4.7 Communication with back-end

As CPT Manager is divided into a front-end and back-end module, it is necessary to communicate between the modules. This communication is facilitated through HTTP calls to the endpoints provided by the back-end module. Since 2017, most browsers have supported the Fetch API technology, which provides an interface for fetching resources across the network.[19] This technology is used in the front-end module.

4.7.1 Request implementation

The request functions of the application are implemented within the *requestUtils* folders. This folder itself contains multiple JavaScript files, each representing a category of requests and with:

- FileRequests
- FolderRequests
- GraphRequests
- LoginRequests
- TestCaseRequests

- OtherRequests

These files define and export functions, which encapsulate a request to a specific endpoint provided by the back-end. Only the results and requested data are returned by the functions themselves, while all of the error logic is contained inside the functions.

4.8 Header components

This section describes the components used to compose the header of the application.

4.8.1 Mantine modals

Some header components use modals, a window that overlays the page's UI, to provide menus for features not intended to be part of the default UI. The Mantine library provides a modal component in its core package.

To use the modal inside of a React component, Mantine provides a special hook, *useDisclosure*. This hook returns a list of two values. The first one is a Boolean indicating whether the modal is open. The second is an object holding three functions, *open*, *close* and *toggle*. These functions are used to control the modal from code. To implement a modal, a Modal component must be included in the current component's JSX. For the modal to function correctly, two props must be passed to the modal, *opened*, controlling the visibility of the modal and *onClose*, providing a function to be called when the user attempts to close the modal from the user interface. Any JSX inserted between the modal component's tags will be displayed inside the opened modal.

4.8.2 Changelog component

The purpose of the *Changelog* component is to offer a menu that allows users to view the updates and functionality introduced in various versions of CPT Manager. In its default state, the component provides a button that opens a modal containing descriptions of the application's versions. Figure 4.11 illustrates the UI defined by the *Changelog* component when it is open.

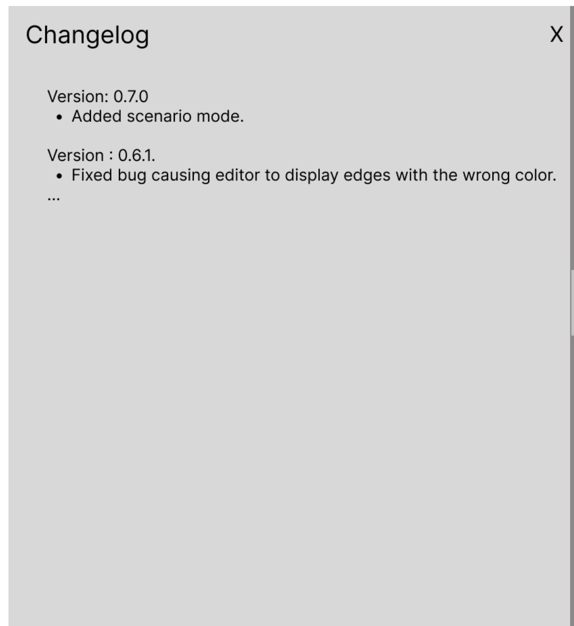

Figure 4.11: UI defined by the *Changelog* component

4.8.3 ColorSwitch component

The *ColorSwitch* component provides a button to switch between Mantine's dark and light colour schemes. It uses two hooks:

- *useMantineColorScheme* to get the function needed to change the global colour scheme of the library.
- *useComputedColorScheme* to get the current colour scheme.

When the button is clicked, it sets the colour scheme to the opposite of what is currently used. The button icon also changes according to the currently active colour scheme.

4.8.4 OptionsButton component

The *OptionsButton* component provides users with additional options to create, load, and save graphs. The current implementation of the CPT Manager allows one to load a graph from a file, export the loaded graph to a file, and generate a graph based on given parameters using this button. After clicking the button, a floating menu will expand from the button, as shown in Figure 4.12, presenting the three available options.

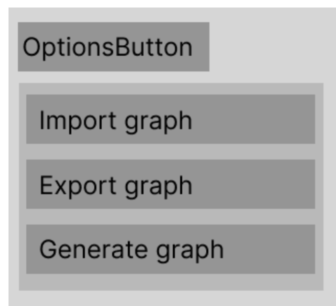

Figure 4.12: Menu defined by the *OptionsButton* component

The component requires three props:

- *graph* - The currently loaded graph passed down by the *App* component.
- *importFunction* - Function that accepts a JSON graph file, which loads the graph into the application.
- *newGraphFromDto* - Function that loads a graph into the application from the provided DTO of the graph.

Hooks

Hooks used inside the *OptionsButton* are:

- *importModal*: useDisclosure - Mantine modal controls for a modal containing file dropzone.
- *generateModal*: useDisclosure - Mantine modal controls for a modal containing the generator form.
- *message*: useState - State containing a message to be displayed inside the generate modal.
- *loadingGen*: useState - Boolean indicating whether a graph is currently being generated and a loading indicator should be displayed.
- *genForm*: useForm - A hook that defines a form provided by the Mantine forms library. This form is used for graph generator parameters.

Functions

OptionsButton has only two defined functions. The first is *generateGraph*, which is called when a graph is to be generated and sets the new graph from the DTO object returned by the back-end call or sets the message to be displayed if an error is returned. The second is *exportGraph*, which is called when the user wants to export the loaded graph to a JSON file.

Export

As mentioned above, the export function is called when the user clicks on the export menu option. This function in turn calls the *toDTO* function from the *dtoUtils* component. Then the function calls the export function from *requestUtils*, which handles the call to the back-end and triggers a browser download for the requested file.

Import

When the user clicks on the import menu option, a modal containing a Mantine dropzone component is opened. The dropzone is set to only accept a singular JSON file and displays a notification when an invalid file is inserted. When a file is provided, the function inside the *importFunction* prop is activated with the file.

Generate

When the user clicks on the generate menu option, a modal containing a form with the generator parameters is opened. The UI illustration of the form defined by the component for generating graphs is in Figure 4.13. After the form is submitted, the *generateGraph* function is called with the form values, and either a new graph is set or a message is displayed inside the modal.

Figure 4.13: Form for generating graphs defined by the *OptionsButton* component

4.8.5 LoginBox component

The *LoginBox* component displays the current log-in status of the user and offers options to log in or register.

The props required by the component are:

- *isLoggedIn* - Boolean representing if an account is logged into the application.
- *nickName* - The name of the logged-in user.
- *functions* - Object containing three functions, *login*, *register*, and *logout*.

When a user is logged in, the component appears as a badge with the user's nickname and a logout button next to it. Otherwise, the component only shows the log-in button, which opens a modal menu. The modal is divided into two tabs using the Tabs Mantine component. One tab contains the login form, and the other one is a register form. The forms are created using the useForm hook from Mantine Forms Library. Both tabs are illustrated in Figure 4.14.

The component has two defined functions, one for each form submission. These functions call their respective callback functions from the functions prop and set errors in appropriate fields if necessary.

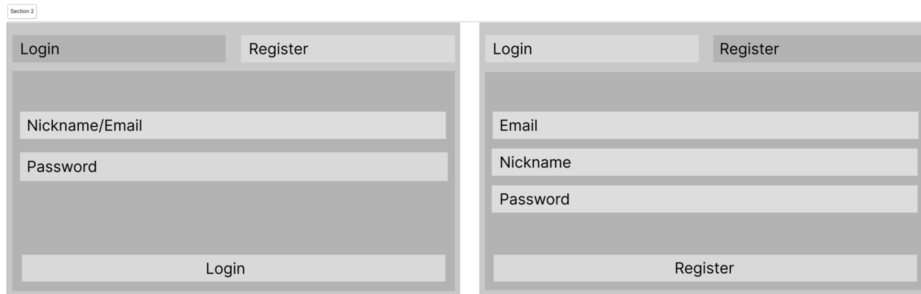

Figure 4.14: *LoginBox* modal tabs

4.9 Left column components

The function of the application's left column is to provide the user with a UI to display, select, and manipulate graphs saved on the server and sorted into user-defined folders. The UI is divided into three tabs, each containing a list of folders from one of three categories, public, custom, and shared. Custom and shared folders are only enabled when a user is logged in. Each folder can be expanded and provides a list of graphs stored inside, which the user can load into the application. In addition, the custom folders tab, which contains folders belonging to the current user, has a button to create new folders, and the folders inside it provide options to delete or rename them. Finally, graphs inside the custom folder can be deleted by hovering over them and clicking the delete button, which appears after a brief delay. The UI defined by the components of the left column is illustrated in Figure 4.15.

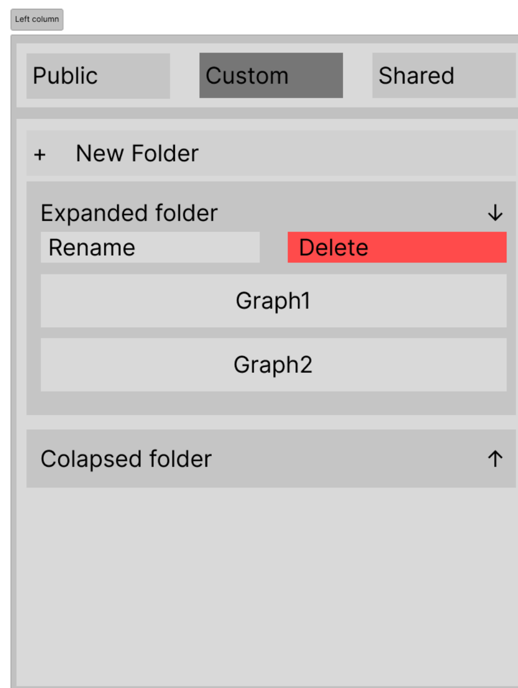

Figure 4.15: Left column UI

4.9.1 GraphSelector component

The *GraphSelector* is the root component of the left application column. Its component diagram can be seen in Figure 4.16.

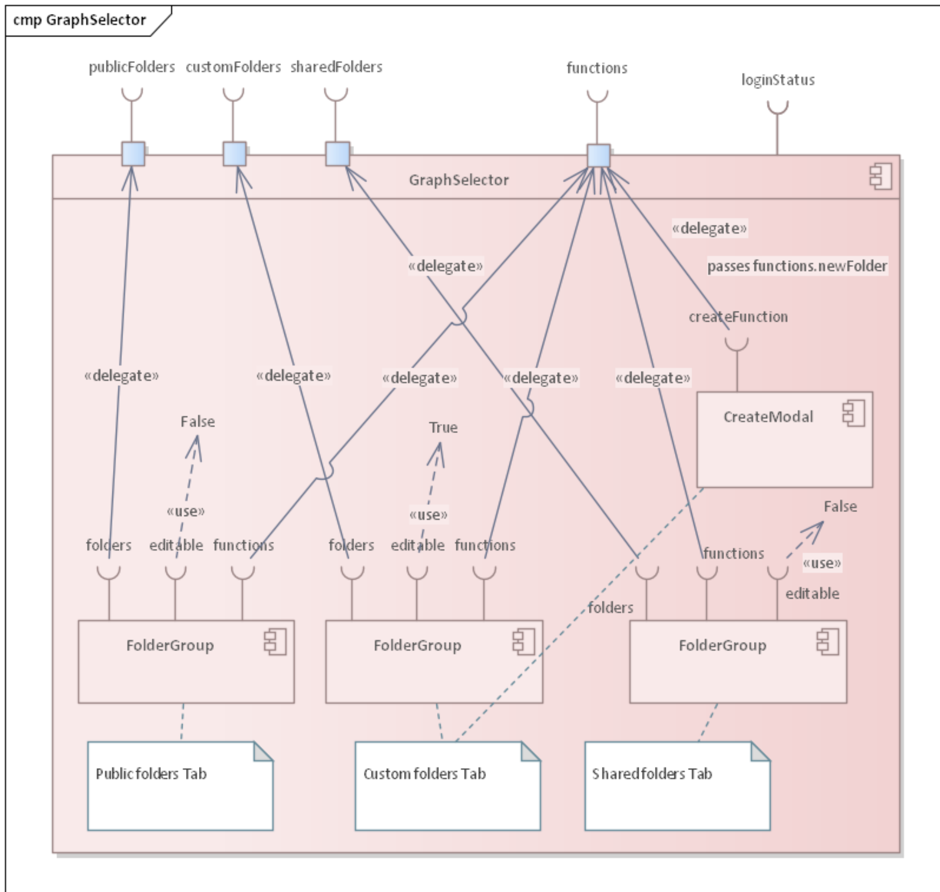

Figure 4.16: GraphSelector component diagram

The component accepts five props, *publicFolders*, *customFolders*, *sharedFolders*, *functions*, and *loginStatus*. The first three props are lists of folders from the three described categories. The functions prop expects an object containing five functions, *newFolder*, *deleteFolder*, *renameFolder*, *getGraph* and *deleteGraph*. The last prop, *loginStatus* is a boolean indicating whether a user is logged in.

The structure of the component is divided into three tabs using the Tabs component from the Mantine library. Each tab contains a *ScrollArea* component from the same library, and inside the scroll area is a custom component *FolderGroup*, which renders the vertical list of folders. For the custom tab, the *FolderGroup* is set to editable, allowing modification of the inside folders and allowing the option of deleting the graph. The scroll read of the custom tab also contains the component *CreateModal*, which presents itself as a button to create new folders. The scroll area is used to limit the height of the list so that it does not exceed the height of the web page if the list is too big to fit inside. If the value of *loginStatus* is false, the custom and shared tabs are set to disabled and the user can only access the public tab.

The component uses only one state hook to control which tab is currently selected, for when the user logs out the component needs to force select the public tab.

4.9.2 FolderGroup component

The *FolderGroup* component displays a list of folders, where a folder can be expanded to a list of graphs contained inside. This is achieved by using the accordion component from the Mantine library. Each item in the list passed through the folder prop is mapped to an accordion item that encapsulates a folder component. The other two props, editable and functions, are passed to Folder components. A diagram of the component is shown in Figure 4.17.

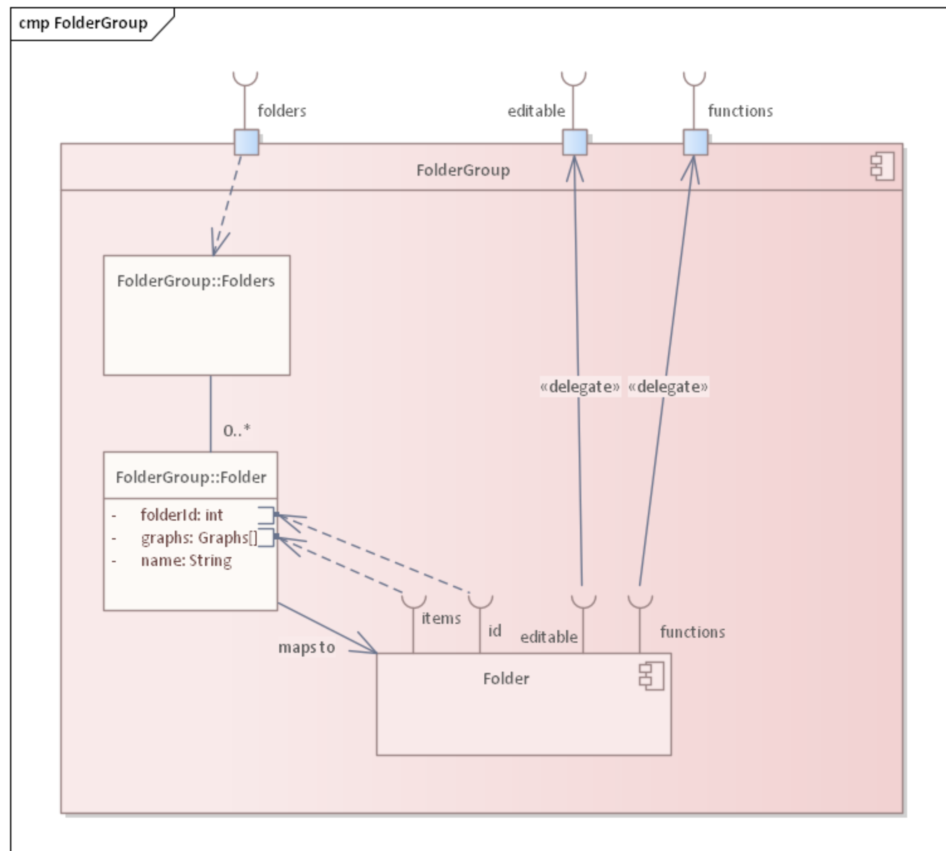

Figure 4.17: *FolderGroup* component diagram

4.9.3 Folder component

The *Folder* renders a list of buttons, each representing a graph contained within the list passed through the *item* prop. When the button is pressed, a function is called to load the graph into the application. If the folder is set as editable, the rename and delete buttons are also displayed on top of the list, and a hover menu over individual graph buttons is enabled, which contains the delete graph option.

The props passed to the component are:

- *items* - List of graph names and their IDs inside the folder.
- *functions* - Object containing the necessary functions for the component.
- *id* - Database ID of the folder.

- *editable* - Boolean representing whether the user can edit the contents of the folder.

The component has no hooks and implements only one function, *deleteFolder*. This function uses the predefined modal from the Mantine modal library to ask for confirmation and passes a lambda function that calls *deleteFolder(id)* when activated. It is assigned to the delete button. The component maps each graph to a button calling *getGraph(graphId)*, which is wrapped in a Mantine menu component with one menu button to delete the graph, which calls *deleteGraph(graphId)*. This menu button is only enabled when the *editable* prop is set. The rename button is provided by the *RenameModal* component, to which *renameFolder* is passed along with the folder ID.

4.9.4 RenameModal and CreateModal components

The *RenameModal* and *CreateModal* components are very similar to each other and are used to create or rename folders. The only functional difference between them is that *RenameModal* also needs an *id* prop, as it works with an existing folder. Both components use the Modals Mantine component to display a form with a text field, a confirmation button, and a possible error message. When the confirm button is called, it calls the function passed through a prop which should perform the required action.

4.10 Righ column components

The purpose of the application's right column changes based on which mode it is currently in. In *edit* mode, the right column provides an interface to interact with the non-visual elements and data of the loaded graph and modify them. It is divided into two tabs, Info and Groups. The Info tab provides expandable windows showing information about the current graph and the currently selected node or edge. The Groups tab allows users to bundle elements of the graph into groups and to set attributes to the entire group instead of individually to each element.

In the *scenario* mode, the right column becomes a menu to browse and see details of generated test cases or to generate new ones. It is divided into three tabs, Process, LNCT, and CPT, each containing a list of test cases generated with the given type of algorithm.

4.10.1 InfoBox component

The *Infobox* component implements the UI for the right column of the application. The component diagram is illustrated in Figure 4.18.

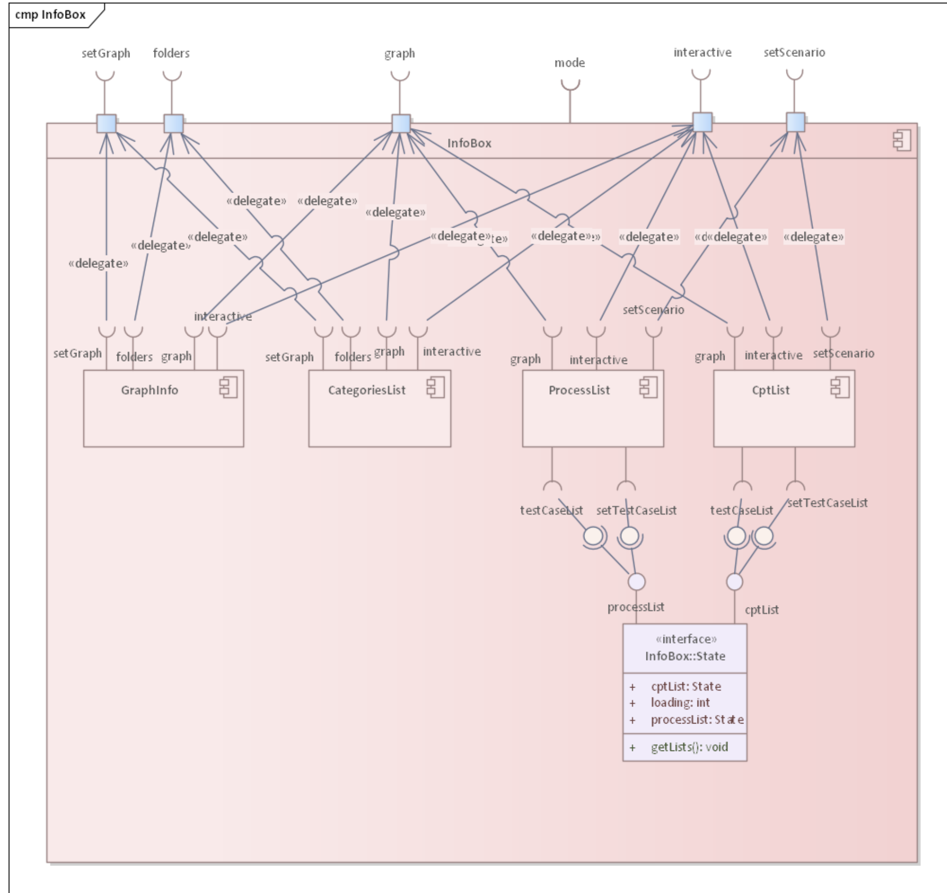

Figure 4.18: InfoBox component diagram

The props accepted by the component are:

- *graph* - Graph loaded inside of the application.
- *setGraph* - Function to set the graph state.
- *interactive* - Boolean indicating whether the graph can be edited by the user.
- *folders* - List of folders owned by the current user.
- *mode* - Mode, *edit* or *scenario*, in which the application is currently in.
- *setScenario* - Function to set the scenario displayed by the editor on the graph.

The component uses three state hooks:

- *processList* - List of process test cases belonging to the current graph.
- *cptList* - List of CPT test cases that belong to the current graph.
- *loading* - Boolean indicating whether the component should render a loading overlay.

Only one function is implemented inside the component, *getLists*. When activated, it retrieves the lists of test cases stored on the server for the current graph. While the function is running, the *loading* prop is active.

The component has one effect hook, which is activated when the *mode* is changed. If the *mode* is changed to *edit*, it resets *processList* and *cptList* back to empty lists. If the mode is changed to *scenario* and the graph ID is not 0, which means that it should be stored on the server, the *getLists* function is called.

The component itself renders its user interface based on the mode in which it is currently in. If the *mode* is edit, it returns a Mantine tab component with two defined tabs, Info and Groups. Inside both tabs is a *ScrollArea* component to prevent the tab contents from spilling from the window. The scroll area for the Info tab contains the *GraphInfo* component, and the other contains the *CategoriesList* component. If the *mode* is scenario, a *Box* component with the position set to relative is returned. This is needed because of the loading overlay component located inside the box. The box also has a tabs component, which compared to the edit mode has three tabs corresponding to the three different types of test case algorithm. The Process and CPT tabs contain a scroll area with the respective test case list component inside. The LCNT tab currently has no content as the LCNT algorithms are unavailable and their functionality has not been implemented yet.

4.10.2 GraphInfo component

Using the *GraphInfo* component, users can view and edit the global graph data and the currently selected node/edge. Figure 4.19 shows the diagram of the component.

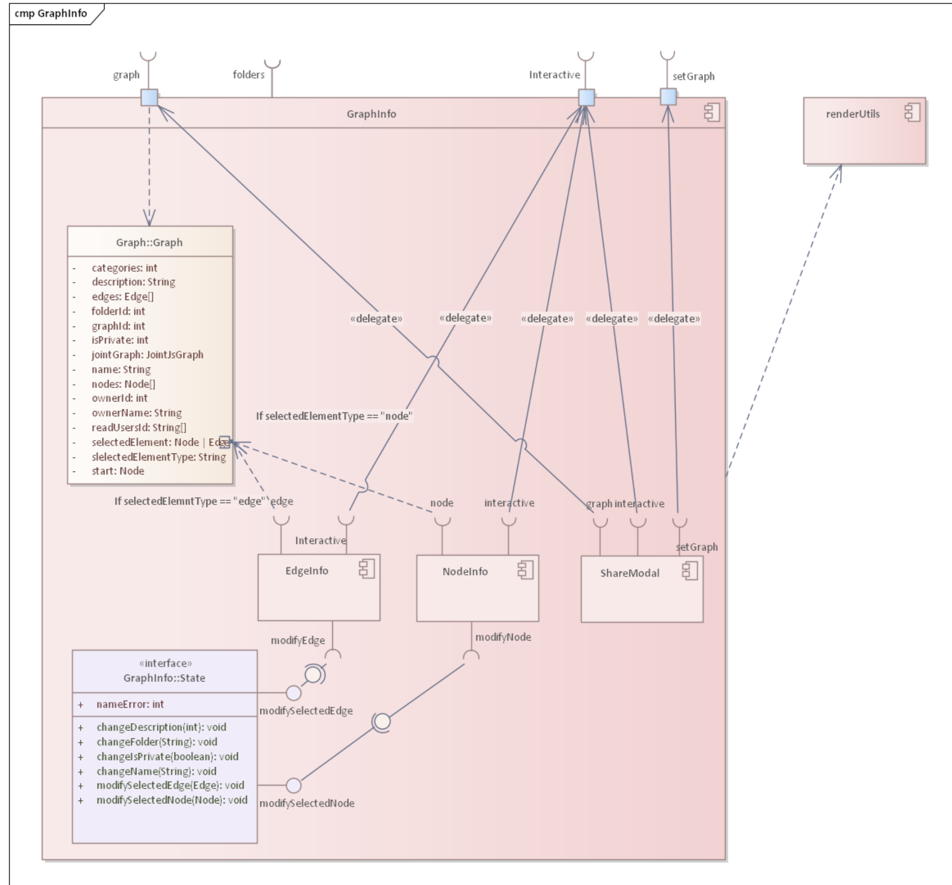

Figure 4.19: GraphInfo component diagram

The component is implemented using an *Accordion* component from the Mantine library and has two expandable sections. The first is Graph info, which contains the information about the graph itself. The second one is Node/Edge info, based on the selected element. If no element is selected, it becomes hidden.

The props required by the component are passed down from the *InfoBox* component and represent the same values. The component has only one state hook, *nameError*, an error message displayed by the graph name input.

The functions implemented inside of the component are:

- *changeName(name)* - Sets a new graph with a changed name. If the name is an empty string, it sets the error message as "Name cannot be empty".
- *changeDescription(description)* - Sets a new graph with a changed description.
- *changeFolder(stringId)* - Sets a new graph with the ID of the folder in which the graph should resign. Due to how the Mantine select component works, the function accepts the ID as a string and converts it to a number.
- *changeIsPrivate(checked)* - Sets a new graph with changed value *isPrivate*.

- *modifySelectedNode(node)* - Sets a new graph where the selected node is replaced with the provided one and calls a *renderUtils.refreshNode* to update its view.
- *modifySelectedEdge(edge)* - Same as *modifySelectedNode* but for the selected edge.

Figure 4.20 illustrates the UI defined by the component.

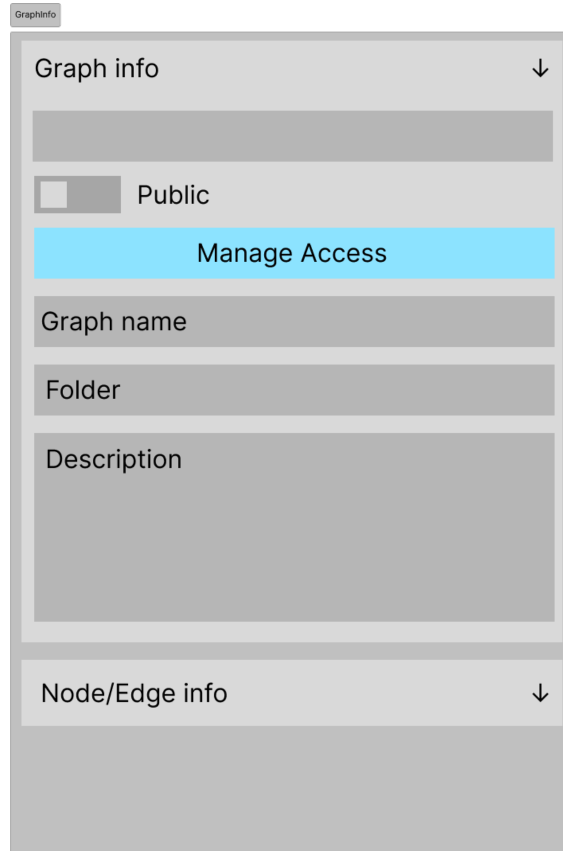

Figure 4.20: UI defined by the *GraphInfo* component

4.10.3 ShareModal component

The *ShareModal* implements a window to share the loaded graph with specific users. The component renders a button that opens a Mantine modal on activation. In this modal, all users with whom the graph is shared are displayed as a group of pills with a remove option. Adding new users to the group is facilitated through text input, where the person's nickname needs to be entered.

Three props are required for the component to work, *interactive*, *graph*, and *setGraph*. When the *interactive* prop is set to false, the modal cannot be opened.

The component uses three state hooks, *pills*, *message*, and *nickname*. The *pill* hook represents the list of users to be displayed, the *message* hook contains information to be shown to the user by the UI, and the *nickname* represents the value of the text input. Additionally, a disclosure hook is used for controlling the modal's state.

The component has three functions, *openWrapper*, *addUser* and *removeUser*. The *openWrapper* function adds additional functionality over the modal's open function. As the users with whom the graph is shared are stored inside the graph only by IDs, a request needs to be sent to the server to acquire their nicknames. This is done by the *openWrapper* function before opening the modal, and the list of nicknames with the corresponding IDs is stored inside the *pills* state. The *addUser* function requests the ID of the user entered in the text input, and if the user exists it is added to the graph or the current user is informed of the error. Finally, the *removeUser* function removes the user with the given ID from the graph.

An effect is defined inside of the component, triggering when the graph value is changed. This effect calls for the *openWrapper* function, but only if the modal is already open. This is done to show the correct information after an update is made by another function of the component.

The UI defined by the *ShareModal* component is illustrated in Figure 4.21.

Figure 4.21: UI defined by the *ShareModal*

4.10.4 NodeInfo and EdgeInfo components

The *NodeInfo* and *EdgeInfo* components are a variation of the same component whose purpose is to display and edit the data of their graph element. The difference between them is in the displayed information and props passed to their child elements. The diagrams of both components are illustrated in Figure 4.22.

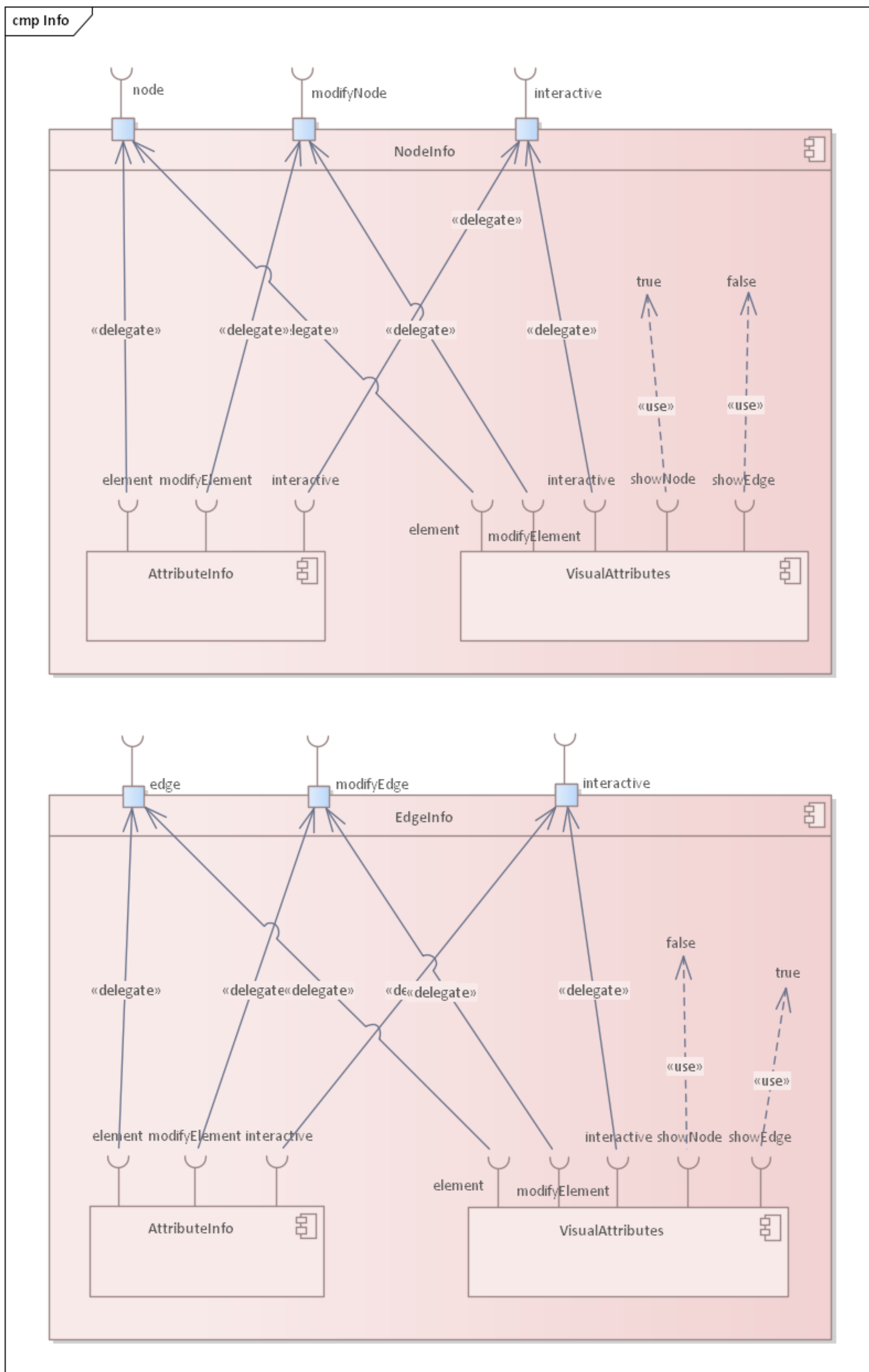

Figure 4.22: NodeInfo and EdgeInfo component diagram

Both components feature a description and number input to set the priority of their attributes and implement two functions to change these data inside the element.

The difference in their UI is in the identifier of the element. For nodes, the identifier is their name, while for edges, the identifier is their ID inside of the graph. Both components use *AttributeInfo* and *VisualAttributes* to display additional attributes related to the element. The UI defined by the components is illustrated in Figure 4.23.

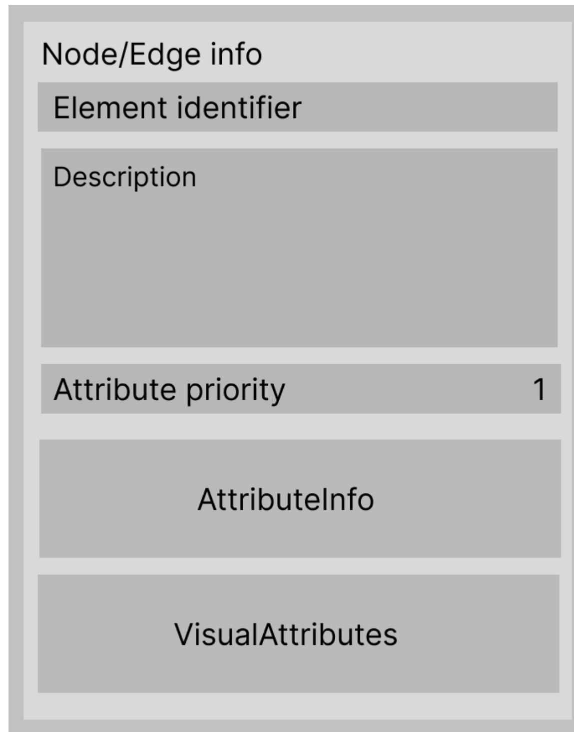

Figure 4.23: UI defined by *NodeInfo*/*EdgeInfo*

4.10.5 CategoriesList component

The *CategoriesList* component renders a list of groups assigned to the current graph. When the groups functionality was first thought up, it was named categories and was changed later to groups. Due to this, the functionality is still referred to inside the application's code as categories.

The component accepts three props passed down from the *InfoBox* component, *graph*, *setGraph*, and *interactive*.

The functions implemented inside the component are:

- *modifyCat(oldCat, newCat)* - Replaces the *oldCat* inside the graph with the *newCat*. This function is passed down to the *AttributeInfo* and *VisualAttributes* components as their *modifyElement* prop.
- *createCat()* - Creates a new category (group).
- *renameCat(name, id)* - Changes the (group) name.
- *getCategoryNodes(cat)* - Returns a list of node IDs assigned to the group in string form. This is necessary for a multi-select UI component.
- *getCategoryEdges(cat)* - Similar to *getCategoryNodes* but for edges.

- *changeCatNodes(cat, selectedValues)* - Modifies the graph nodes, so that the category assignment matches the selected values.
- *changeCatEdges(cat, selectedValues)* - Similar to *changeCatNodes*, but for edges.
- *deleteCat(cat)* - Deletes the group from the graph.
- *changePriority(value, oldcat)* - Calls *modifyCat()* to change the attribute priority value to the given one.

Most of these functions also call for a refresh on the view of modified elements assigned to the affected group.

The UI list is implemented using the Mantine *Accordion* component, and each graph group (category) is mapped to an accordion item. When a group is opened, the user can change its data, delete it, assign nodes and edges to the group through a multi-select component, and change the priority of the group's attributes. The data and visual attributes of the group can be edited through the *AttributeInfo* and *VisualAttribute* components. On top of this list, there is also a button for creating new groups. The UI defined by the component is shown in Figure 4.24.

Figure 4.24: UI defined by *CategoriesList*

4.10.6 AttributeInfo component

The *AttributeInfo* component allows the assignment of data attributes as key-value pairs to an element, displays already assigned values, and provides the option to change or delete them. The component is designed to be reused for different elements and takes three props, *element*, *modifyElement*, and *interactive*. The *element* prop requires an object that has a list of attributes assigned to it. The attributes themselves should be objects with name and value strings. The element is updated by calling the function passed through the *modifyElement* prop. The last prop, *interactive*, determines whether the user can modify the element.

The component has three state hooks:

- *message* - Contains text to be displayed inside the component to the user. It is used if a user tries to create an existing attribute or an attribute without a name.
- *nameField* - Linked to the value of the *name* text to create a new attribute.
- *valueField* - Linked to the value of the input of *value* text to create a new attribute.

Three functions for element modification are defined inside the component, *deleteAttribute*, *modifyAttribute* and *addNewAttribute*.

A new attribute is created by filling in the *name* and *value* fields and pressing the plus button. The created attributes are presented as a vertical list, where each column contains an attribute name, value, and a delete button. The attribute value can be modified by clicking on it and rewriting it. The UI defined by the component is illustrated in Figure 4.25.

+	Name	Value
X	Attribute1	Value1
X	Attribute2	Value2

Figure 4.25: UI defined by *AttributeInfo*

4.10.7 VisualAttributes component

The *VisualAttributes* component serves to edit predefined values that affect the visuals of the elements. Similarly to the AttributeInfo component, it requires the *element*, *modifyElement*, and *interactive* props, which serve the same purpose. In addition, it requires the props *showNode* and *showEdge*, which accept Boolean values to

determine whether attributes relating to the given elements should be shown. When *showNode* is active, the UI allows to change the node outline, node fill colours and node size. For *showEdge* set to true, the UI displays an option to change the colour of an edge. Regardless of the props passed, the UI offers an option to change the element's label. Five functions are defined inside the component; each one updates one of the listed attributes.

4.10.8 Scenario mode components

As stated above, when the application is in the *scenario* mode, the *InfoBox* component renders tabs, each containing a list of test cases generated with a different type of algorithm. In the latest version of CPT Manager, only the tabs for process and PCT group of test cases are implemented, since LCNT algorithms are not yet available.

List components

The insides of the scenario tabs is implemented by the list components, *ProcessList* and *PctList*. These components accept a list of test cases generated with their respective algorithm and a function to update this list as one of their props. In addition, they accept the currently loaded graph as a prop, a *setScenario* function, which accepts a path from the test case and highlights it on the displayed graph, and the interactive prop, which determines if the user has permission to edit the scenarios. Each test case from the list is mapped to a test case component matching the algorithm type, which is wrapped inside an accordion item making it expandable. On top of the list is the button supplied by the test case modal components, which are responsible for creating new test cases. The list components implement three functions, *addNewTestCase*, *exportTestCase* and *deleteTestCase*, which are passed to the child components.

TestCase components

The test case components display the data of the test case passed to them through their props. Currently, there are two test case components, *CptTestCase* and *ProcessTestCase*, each corresponding to the type of algorithm with which the passed test case has been generated. This distinction is necessary because the algorithms require different parameters to be filled when called. The test case component displays these parameters along with the different paths generated by the test case. The component provides a button to display a path on the loaded graph by calling the *setScenario* function passed to a prop. A test case can also be deleted with a passed *deleteTestCase* function and exported to a JSON file with an *exportTestCase* function also passed through a prop.

Test case modal components

A modal component is implemented for each algorithm type, which contains a form with the required parameters to generate a new test case with the given algorithm type. Two modals are currently implemented, *ProcessModal* and *CptModal*. The components accept a *addNewCase* function and the graph for which the test case is generated as props. The modals are opened with a button provided by the component, and once the parameters are set, a new test case can be generated by submitting the form. After submission, the test case is generated by the API call and stored by calling the *addNewcase* function passed as a prop.

4.11 Benchmark creation

One of the requirements for the thesis was a creation of ten public models based on real-life systems that could be used as benchmarks. These systems were created by researching papers and finding control-flow graphs mainly on Google Scholar and other internet sources. The researched models are available to access from the deployed application `https://cpt.fel.cvut.cz/manager`, with the sources they were based on inside the graph's description.

Figure 4.26 presents one of the created graphs. All created graphs are attached in the Appendix C.

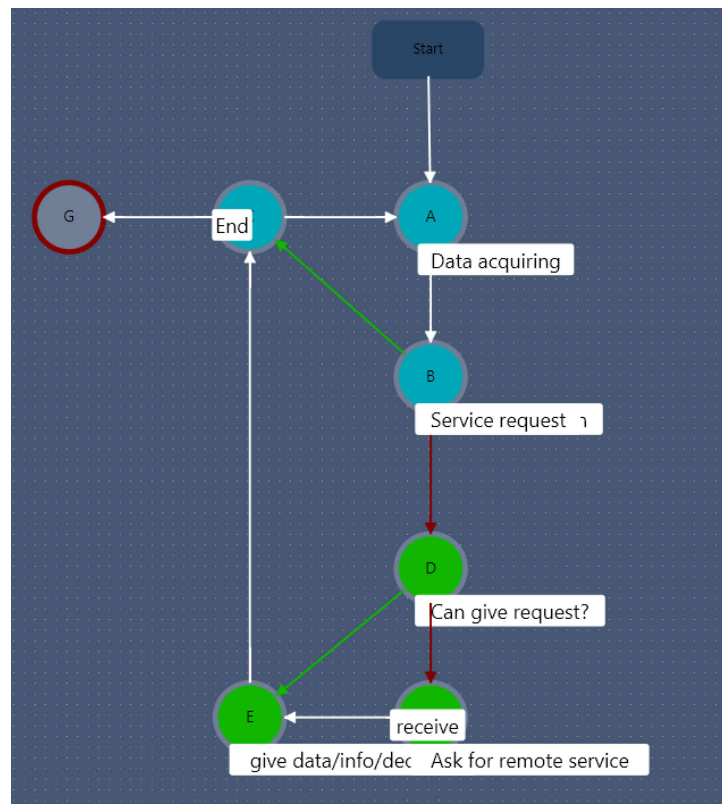

Figure 4.26: Example diagram created inside application based on Figure 1.14 from paper `https://www.sciencedirect.com/science/article/pii/B9780128183731000019`.

Chapter 5
Quality assurance

This chapter of the thesis focusses on describing the methods used to test the application and ensuring that its core functionality behaves as expected.

5.1 End-To-End tests

End-to-End testing, also referred to as E2E testing, is a software testing technique that tests the entire application from the perspective of a user by simulating real-world scenarios.[21] This type of testing was chosen to ensure that the front-end module interacts correctly with the back-end module of the application and that all features of the application behave according to their requirements.

The E2E tests are implemented using a JavaScript testing framework called Cypress. The framework divides the tests into files with the *cy.js* extension. The tests inside these files are described as a list of commands to be run in the sequence, and all the tests inside the file are run together. Cypress then provides a simple user interface that visually shows the steps in the test as it is running and after it has been completed.[15]

Test environment

The implemented tests are run with the **npm run test** command, which implements the cypress UI. For the tests to run correctly, these conditions need to be met inside the local environment:

- A vite dev server needs to be running on port 5173.
- A local instance of the back-end module needs to be running. To configure the web application to make requests to the local running instance of the back-end module, the URL must be set inside the *restService.js* file.
- An account needs to exist on the server with email test@test.test, nickname autotest, password 12345678 and the account should own no folders and graphs.
- An account needs to exist on the server with email test2@test.test, nickname autotest2, password 12345678 and the account should own no folders and graphs.

5.1.1 Test scenarios

The test files implemented for the application divide the test scenarios into groups based on the functionality they test:

- Login tests
- Export/Import tests
- Folder tests
- Basic save tests
- Scenario tests
- Share tests

The complete list of all scenarios implemented within these files is included as an appendix B to the thesis.

Results

Overall, the tests revealed only minor bugs in the system, mostly caused by an integration of a new feature, a mistake in the written code, or changes in the back-end API that were overlooked during the development of the front-end module.

5.2 Exploratory testing

Instead of defining structured test cases, exploratory testing focusses on free discovery and relies on the experience of individual testers to uncover defects and edge cases that are not easily found by other tests.[22]

5.2.1 Results

Exploratory testing was used during the implementation of the application after addition after each new feature. It helped uncover small defects such as the owner name of the local graph not being displayed correctly after a login action was performed, or parallel edges not showing correctly in very specific scenarios. Overall, exploratory testing discovered small bugs that were not detrimental to the application's functionality, which would not be uncovered by the bigger automated tests testing the major functionalities of the application.

5.3 Performance testing

During development performance testing was used to validate that the application could handle graphs with higher element counts without making the UI unresponsive. It was essential that this type of testing would be done after each major feature implementation as building the application on top of non-optimal code would make it costly to redesign the application in the future. The tests themselves were performed by generating graphs with high element counts and measuring the time it took the desired actions to execute.

5.3.1 Results

These tests revealed major flaws in the initial design of the application architecture in the early stages of development that would have been almost impossible to change at a later stage. Initially, the application did not use the two-graph model architecture, but relied only on the custom model. The JointJs graph was fully encompassed

inside the editor component, which would erase and redraw the whole graph of every element upon the custom model change. This method proved to be non-ideal, as the time to redraw the graph increased exponentially with every element added, which made the application unresponsive for noticeable period of time. That is why the architecture was redesigned to the less elegant, but much better performing one used today.

Chapter 6
Conclusion

The result of this thesis is the successful creation of a front-end module for a test data management system. The module can view, create, and edit system models and provides the user interface to interact with the back-end module of the system. These interactions include importing/exporting graphs into a file, generating artificial system models based on user-defined parameters, storing and accessing models on the server, and generating test-cases for system models, which can be highlighted and displayed graph. Furthermore, the module was covered with end-to-end tests to ensure the correct functionality of its features. Finally, a public set of ten system models was created based on background research, which is accessible inside the application.

The CPT Manager platform, whose front-end module development started as a subject of this thesis, constitutes a powerful tool for software testers as it combines the ability to create, share, store and edit system models, which can be customised to a high degree, and allows the generation and storing of test cases for the given models. The development is expected to continue as part of the students thesis under the CTU Faculty of Electrical Engineering.

The thesis was also beneficial to the author, as it provided practical experience in the process of developing web applications including analysis, design, implementation, testing, and collaboration with back-end developers. Additional experience gained was in the field of academic writing and research.

Based on the feedback received and the tests conducted, it can be said that the requirements for this thesis were fully met. The implemented application meets all functional requirements and is ready for public deployment.

Bibliography

[1] Wieruch, Robin. The road to react: Your journey to master plain yet pragmatic React. js. Robin Wieruch, 2017.

[2] Saks, Elar. JavaScript Frameworks: Angular vs React vs Vue, 2019.

[3] "Sparx systems: Use Case Diagram" May 2024 [Online] Available: https://sparxsystems.com/enterprise_architect_user_guide/14.0/model_domains/usecasediagram.html

[4] Ammann, Paul, and Jeff, Offutt. Introduction to software testing. Cambridge University Press, 2016

[5] "mdn web docs: Introduction to client-side frameworks" May 2024 [Online] Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction#things_to_consider_when_using_frameworks

[6] "npm trends: Front-end framework popularity" May 2024 [Online] Available: https://npmtrends.com/@angular/core-vs-preact-vs-react-vs-svelte-vs-vue

[7] "mdn web docs: Getting started with React" May 2024 [Online] Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started

[8] "JointJs" May 2024 [Online] Available: https://www.jointjs.com/

[9] "Oxygen project" May 2024 [Online] Available: http://still.felk.cvut.cz/oxygen/

[10] "JointJs Resources" May 2024 [Online] Available: https://resources.jointjs.com/tutorial

[11] "Mantine: Getting started" May 2024 [Online] Available: https://mantine.dev/getting-started

[12] "Tabler docs: Tabler Icons" May 2024 [Online] Available: https://tabler.io/docs/icons

[13] "Vite guide: Getting started" May 2024 [Online] Available: https://vitejs.dev/guide/

[14] "Jetbrains: Webstorm" May 2024 [Online] Available: https://www.jetbrains.com/webstorm/

[15] "Cypress: Why Cypress" May 2024 [Online] Available: https://docs.cypress.io/guides/overview/why-cypress

[16] "Atlassian: What is Git" May 2024 [Online] Available: https://www.atlassian.com/git/tutorials/what-is-git

[17] "Robin Vieruch: How to setup React.js on Windows" May 2024 [Online] Available: https://www.robinwieruch.de/react-js-windows-setup/

[18] "React dev: Learn React" May 2024 [Online] Available: https://react.dev/learn

[19] "MDN web docs: Fetch API" May 2024 [Online] Available: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[20] "Free code camp: What Every React Developer Should Know About State" May 2024 [Online] Available: https://www.freecodecamp.org/news/what-every-react-developer-should-know-about-state/

[21] "Katalon: What is End-to-End Testing" May 2024 [Online] Available: https://katalon.com/resources-center/blog/end-to-end-e2e-testing

[22] "Atlassian: Exploratory testing" May 2024 [Online] Available: https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing

[23] "Underscorejs" May 2024 [Online] Available: https://underscorejs.org/

[24] Bureš, Miroslav. Model-based Software Test Automation. 2018

Appendix A

Basic user manual

This user manual for the CPT Manager application covers the basic steps needed to control the application. A more comprehensive manual will be available to access from the application at a future date.

A.1 Register and login

To register for the first time, click on the button in the upper right corner of the screen. An overlay with to tabs should appear. Choose the Register tab and fill in all the required information. The password must be at least 8 characters long and the nickname at least 4 characters long and should not contain any special characters. The registration menu is shown in Figure A.1.

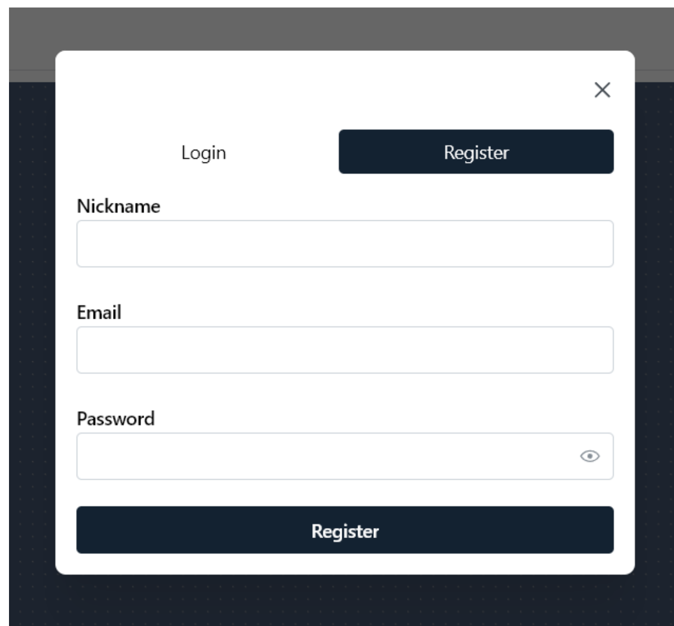

Figure A.1: Registration menu

After successful registration, login into the account by clicking on the login button and entering the information used for registration.

A.2 Graph creation and saving

This section of the manual goes through the process of creating a graph, adding elements to the graph, and saving it on the server.

A.2.1 Graph creation

When the application is started, an new blank graph is automatically created and loaded into the UI. If some other graph is currently loaded, a new graph can be created by clicking on the New graph button inside the header, which can be seen in Figure A.2.

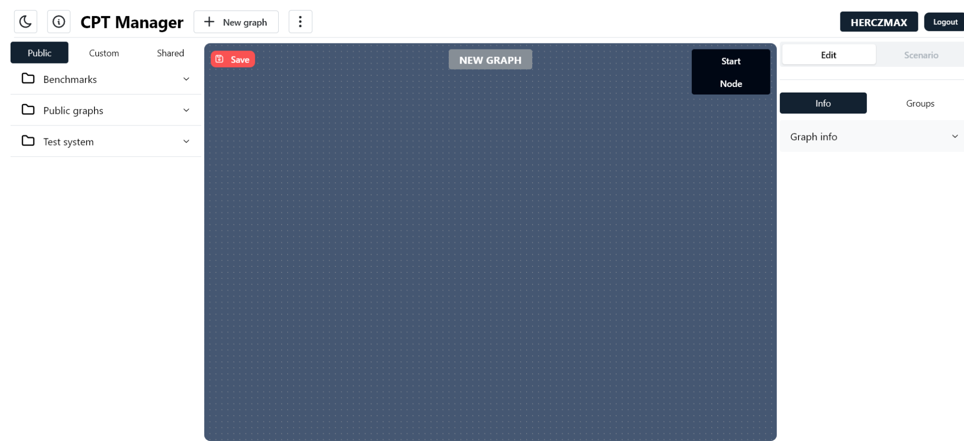

Figure A.2: Application UI with blank graph

A.2.2 Element creation

New nodes can be created by pressing the start and node buttons, which can be seen in Figure A.2, in the upper right corner of the editor.

A new edge can be created by hovering over a node and dragging from the arrow button to the target node. This button can be seen in Figure A.3.

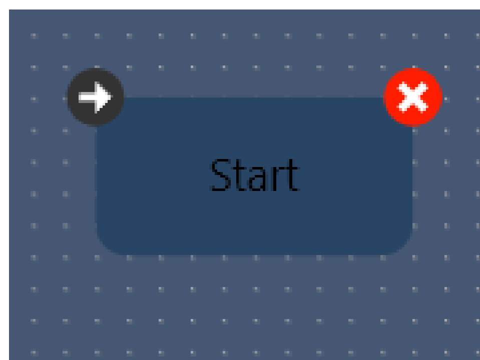

Figure A.3: Hovered over node

A.2.3 Saving the graph

Before a graph can be saved, at least one custom folder must be created. A custom folder can be created through the left column inside the custom tab by clicking on the New Folder button. This is pictured in Figure A.4.

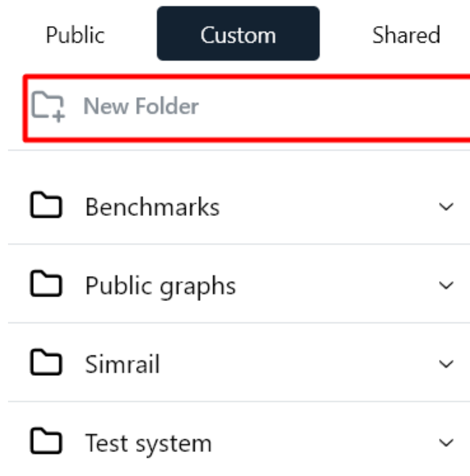

Figure A.4: Custom tab of the left application column with the New Folder button highlighted

To save the graph into a created folder, the folder needs to be selected, and the graph needs to have a name. This can be done through Graph Info located inside the right column. This is pictured in Figure A.5. The graph is then saved through the save button in the upper left corner of the editor, which can be seen in Figure A.2.

Figure A.5: The Graph Info UI

A.3 Generating test cases

Once the graph is saved, a test case can be generated by switching to the scenario mode through the switch seen in Figure A.5.

A simple process test case using the PCT algorithm can be generated by clicking on the Generate process test case button. The button opens a test case with TDL 1 is generated by clicking on the generate button.

Once the process test case is generated, it can be seen and expanded within the left column as seen in Figure A.6. A sequence can be displayed on the graph by clicking on the eye icon next to it.

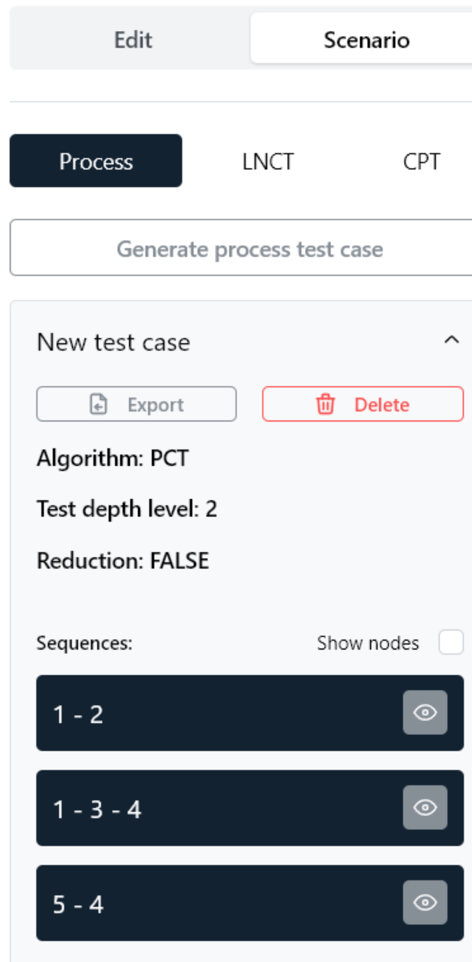

Figure A.6: Generated test case

Appendix B

Test scenarios

Login tests

Name errors test.

1. Click on the login button.
2. Enter the password *12345678*.
3. Click on the login submit button.
4. Expect the error message *Invalid login*.
5. Enter the nickname *nonexistentuser*.
6. Expect the error message *Username/email doesn't exist*.

Password errors.

1. Click on the login button.
2. Enter the nickname *autotest*.
3. Click on the login submit button.
4. Expect the error message *Password must have at least 8 characters and no whitespace characters*.
5. Enter the password *aaaaaaaaaaaaa*.
6. Expect the error message *Wrong password*

Email login and logout.

1. Click on the login button.
2. Enter the password *12345678*.
3. Enter the nickname *test@test.test*.
4. Click on the login submit button.
5. Verify the website has an login badge with text *autotest*.
6. Click the logout button.
7. Verify that the website shows the login button.

Nickname login and logout.

1. Click on the login button.
2. Enter the password *12345678*.
3. Enter the nickname *autotest*.
4. Click on the login submit button.
5. Verify the website has an login badge with text *autotest*.
6. Click the logout button.
7. Verify that the website shows the login button.

Export/Import tests

Export and import graph.

1. Click on the Start button.
2. Click on the Node button.
3. Verify that the canvas contains a start node.
4. Verify that the canvas contains a node with the name *A*.
5. Clear the graph name input and type *Test graph*.
6. Click on the options button and click on the *Export graph* menu option.
7. Refresh the page.
8. Click on the options button and click on the *Import graph* menu option.
9. Upload the downloaded file to the file input.
10. Verify that the canvas contains a start node.
11. Verify that the canvas contains a node with the name *A*.
12. Verify that the graph name input has value *Test graph*.

Folder tests

Before each test starts, the account *autotest* is automatically logged on to the website.

Logout - custom and shared folders are disabled.

1. Logout of the account.
2. Verify that custom folder tab is disabled.
3. Verify that shared folder tab is disabled.

Create folder, rename and delete folder.

1. Click on the custom folder tab.
2. Click on the new folder button.
3. Enter *Test folder* into the folder name input field.
4. Click on the new folder submit button.
5. Verify that the custom folder tab contains a folder with the name *Test folder*.
6. Expand the folder.
7. Click on the rename button.
8. Enter *Test folder2* into the folder rename input field.
9. Click on the folder rename submit button.
10. Verify that the custom folder tab contains a folder with the name *Test folder2*.
11. Expand the folder.
12. Click on the delete button.
13. Click the folder delete confirmation button.
14. Refresh the page.
15. Click on the custom folder tab.
16. Verify that a folder with name *Test folder2* does not exist.

Cannot create existing folder.

1. Click on the custom folder tab.
2. Click on the new folder button.
3. Enter *Test folder* into the folder name input field.
4. Click on the new folder submit button.
5. Click on the custom folder tab.
6. Click on the new folder button.
7. Enter *Test folder* into the folder name input field.
8. Click on the new folder submit button.
9. Verify that the the error message *You already own a folder with this name* is displayed.
10. Click outside the create folder menu.
11. Expand the folder.
12. Click on the delete button.
13. Click the folder delete confirmation button.

Cannot rename to existing folder.

1. Click on the custom folder tab.
2. Click on the new folder button.
3. Enter *Test folder* into the folder name input field.
4. Click on the new folder submit button.
5. Click on the new folder button.
6. Enter *Test folder2* into the folder name input field.
7. Click on the new folder submit button.
8. Expand the folder with name *Test folder2*.
9. Click on the rename button.
10. Enter *Test folder* into the folder rename input field.
11. Click on the folder rename submit button.
12. Verify that the the error message *You already own a folder with this name* is displayed.
13. Click outside the rename folder menu.
14. Click on the delete button.
15. Click the folder delete confirmation button.
16. Expand the *Test folder*.
17. Click on the delete button.
18. Click the folder delete confirmation button.

Basic save tests. Before each test starts, the account *autotest* is automatically logged on to the website and the *Save Folder* is created.

After each test, the folder is deleted.

Save errors.

1. Click on the save button.
2. Verify that the *Graph is not assigned to any folder!* error message is displayed.
3. Clear the graph name input field.
4. Click on the save button.
5. Verify that the *Graph name cannot be empty!* error message is displayed.

Graph is saved to folder, can be retrieved, contents are correct and can be deleted.

1. Click on the Start button.
2. Click on the Node button.
3. Verify that the canvas contains a start node.
4. Verify that the canvas contains a node with the name *A*.
5. Clear the graph name input and type *Test graph*.
6. Click on the folder selection and select the option *Save Folder*.
7. Click on the save button.
8. Refresh the site.
9. Click on the custom folder tab.
10. Expand the *Save Folder*.
11. Click on the *Test Graph*.
12. Verify that the canvas contains a start node.
13. Verify that the canvas contains a node with the name *A*.
14. Verify that the graph name input has value *Test graph*.
15. Hover over the *Test graph* button.
16. Click the delete graph button.
17. Click on the confirm button.
18. Verify that the *Test graph* does not exist.
19. Close the folder.

Graph cannot be saved when not logged in.

1. Logout.
2. Verify that the save button does not exist.

Scenario tests

Before each test starts, the account *autotest* is automatically logged on to the website and the *Scenario Folder* is created.

After each test, the folder is deleted.

Scenario not persistent when logged out.

1. Logout.
2. Import the *test_graphs/graph.json* file.
3. Click on the scenario button.
4. Click on the *Generate process test case* button.
5. Set the scenario name as *Test scenario*.
6. Click on the generate button.

7. Verify that the test case exists.
8. Click on the edit button.
9. Click on the scenario button.
10. Verify that the test case does not exist.

Edit locked when unsaved.

1. Import the *test_graphs/graph.json* file.
2. Verify that the scenario button is disabled.

Scenario persistent when logged in and deleted on graph change.

1. Import the *test_graphs/graph.json* file.
2. Set graph name input to *Test graph*.
3. Click on the folder select input and choose Scenario folder.
4. Click on save button.
5. Click on the scenario button.
6. Click on the *Generate process test case* button.
7. Set the scenario name as *Test scenario*.
8. Click on the generate button.
9. Verify that the test case exists.
10. Click on the edit button.
11. Click on the scenario button.
12. Verify that the test case exists.
13. Click on the edit button.
14. Click on the node button.
15. Click on save button.
16. Click on the scenario button.
17. Verify that the test case does not exist.

Share tests

Before each test starts, the account *autotest* is automatically logged on to the website and the *TestFolder* folder is created.

After each test, the folder is deleted.

Set public and back.

1. Set graph name input to *Test graph*.
2. Click the public switch.
3. Click on the folder select input and choose *TestFolder* folder.
4. Click on the save button.
5. Logout.
6. Expand the *TestFolder* folder.
7. Click on the *Test graph* button.
8. Verify that the graph name input has the value *Test graph*.
9. Login into the *autotest* account.
10. Click on the custom tab.
11. Expand the *TestFolder* folder.
12. Click on the *Test graph* button.
13. Click on the public switch.
14. Click on the save button.
15. Logout.
16. Verify that a folder with name *TestFolder* does not exist.
17. Login into the *autotest* account.

Share with account and take access back.

1. Set graph name input to *Test graph*.
2. Click on the *Manage access* button.
3. Enter *autotest2* into the new user input.
4. Click on the add user button.
5. Close the menu.
6. Click on the folder select input and choose *TestFolder* folder.
7. Click on the save button.
8. Logout.
9. Login into the *autotest2* account.
10. Click on the shared folders tab.
11. Expand the *TestFolder* folder.
12. Click on the *Test graph* button.
13. Verify that the name input has the value *Test graph*.
14. Logout.
15. Login into the *autotest* account.
16. Click on the custom folders tab.
17. Expand the *TestFolder* folder.

18. Click on the *Test graph* button.
19. Click on the *Manage access* button.
20. Click on the remove button inside the element containing *autotest2*.
21. Close the menu.
22. Click on the save button.
23. Logout.
24. Login into the *autotest2* account.
25. Click on the shared folders tab.
26. Verify that the *TestFolder* folder does not exist.
27. Logout.
28. Login into the *autotest* account.

Cannot share with non-existent user.

1. Click on the *Manage access* button.
2. Enter *autotest3* into the new user input.
3. Click on the add user button.
4. Verify that the *User with this nickname does not exist.* error message is displayed.

Appendix C

Created benchmarks

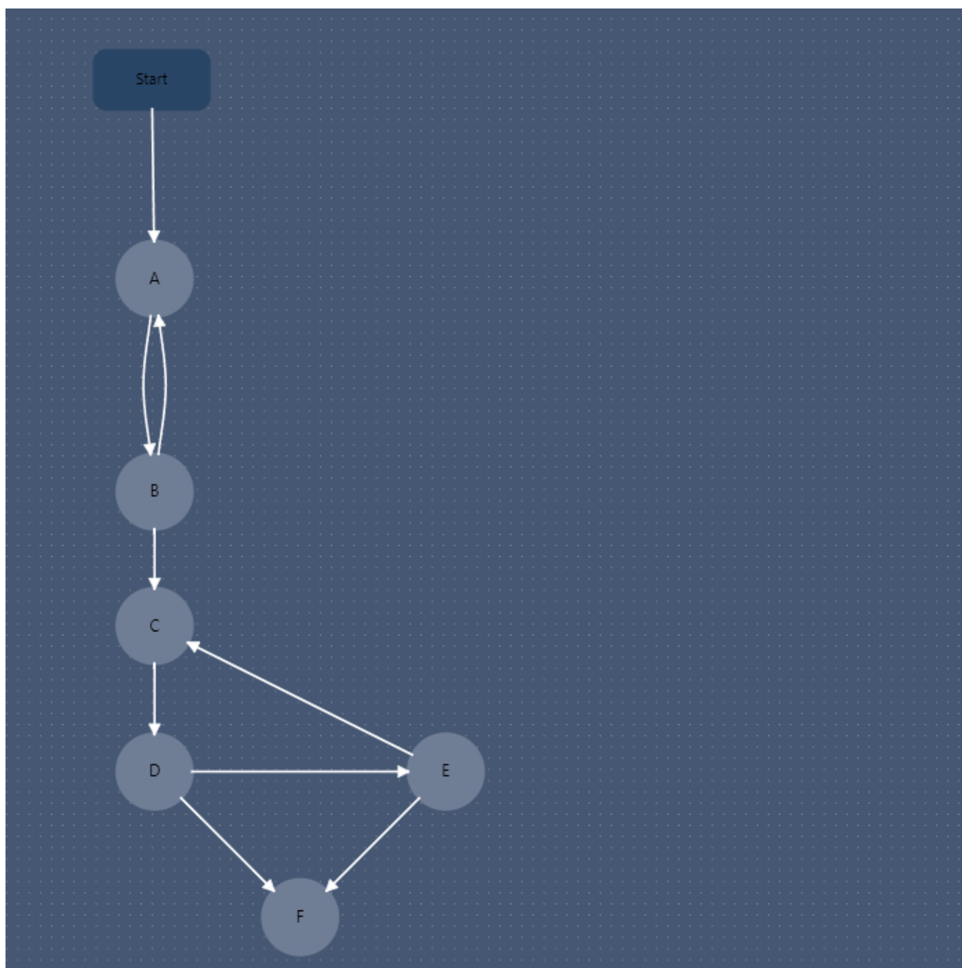

Figure C.1: System model based on `https://ieeexplore.ieee.org/abstract/document/9079344`.

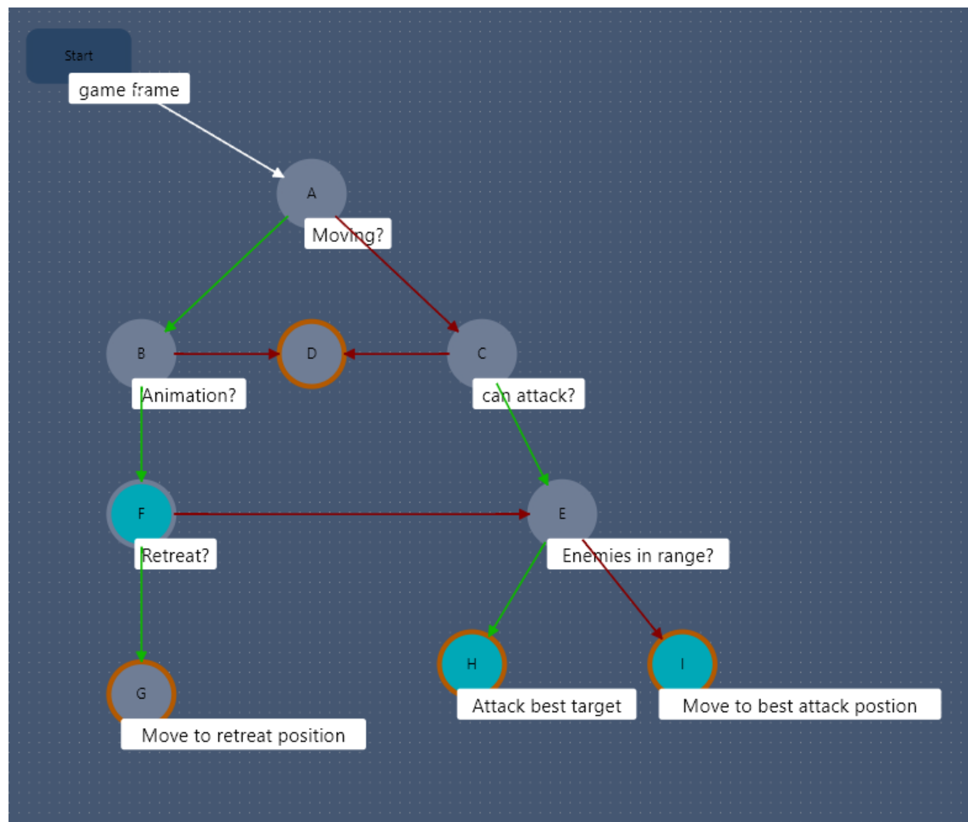

Figure C.2: System model based on `https://ieeexplore.ieee.org/abstract/document/9079344`.

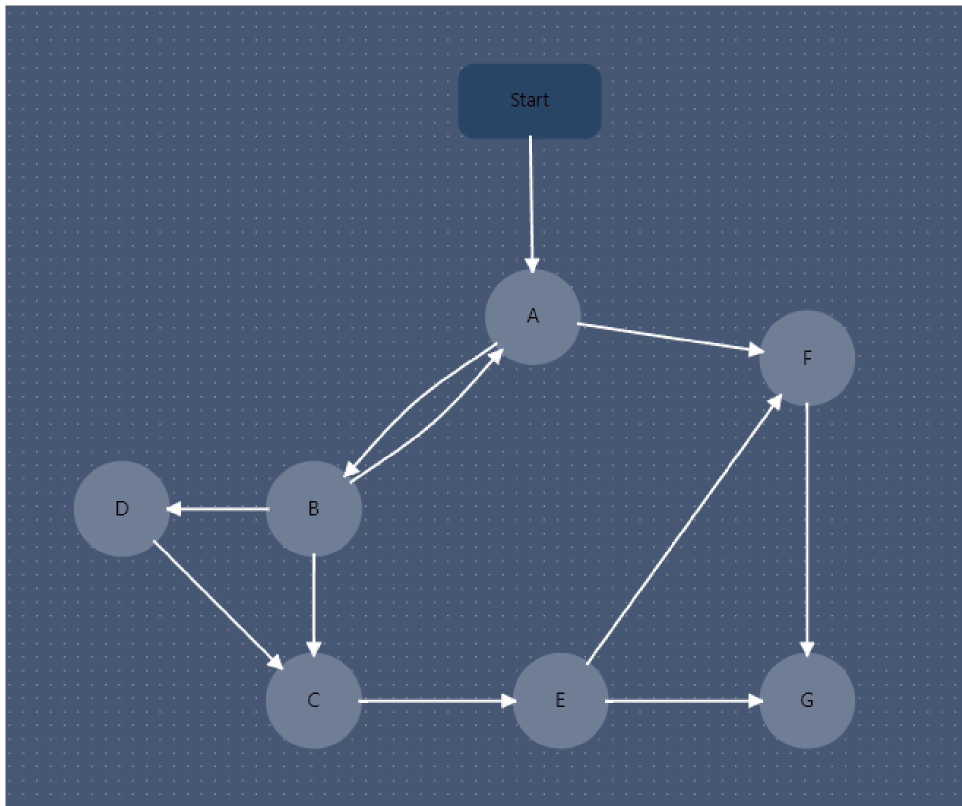

Figure C.3: System model based on https://eprints.unmer.ac.id/id/eprint/2843/1/1.%20Jurnal.pdf.

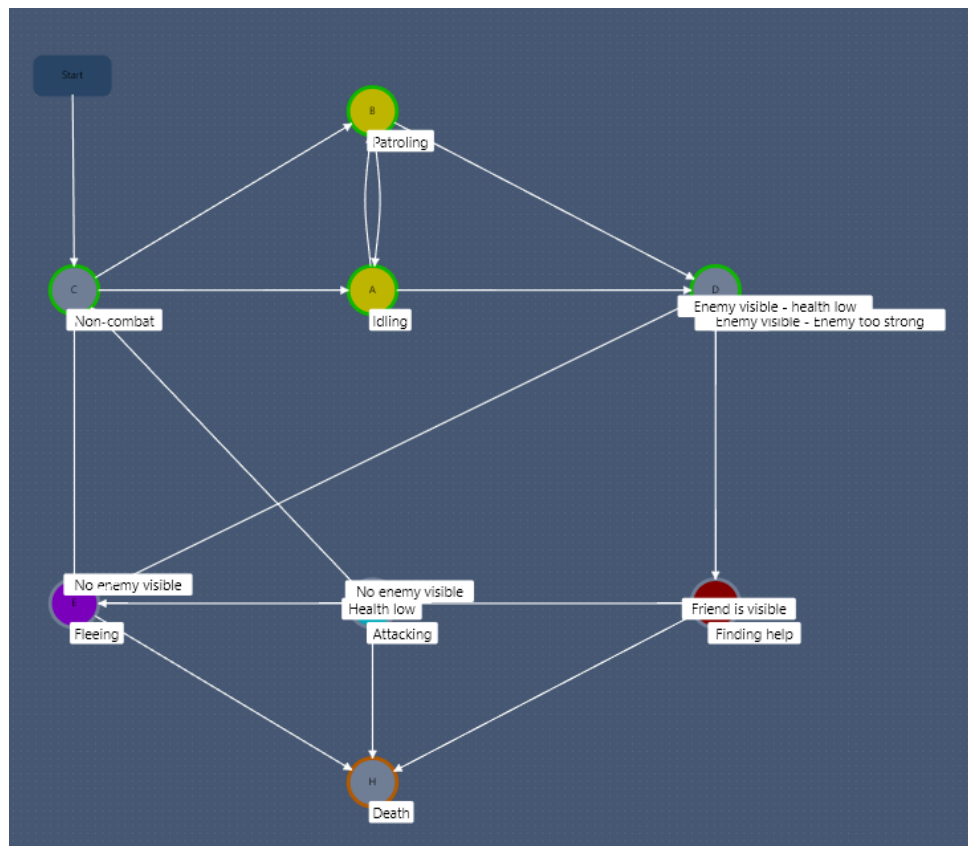

Figure C.4: System model based on `https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-ai-r4942/`.

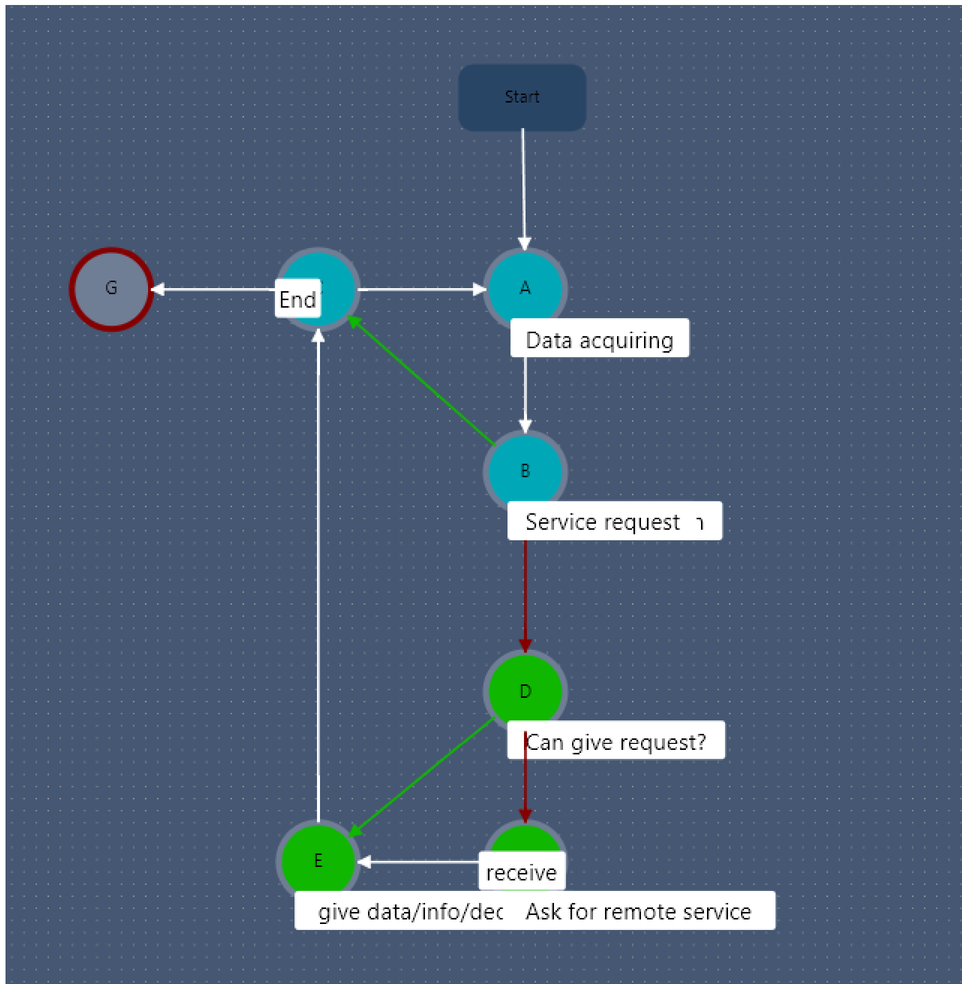

Figure C.5: System model based on `https://www.sciencedirect.com/science/article/pii/B9780128183731000019`.

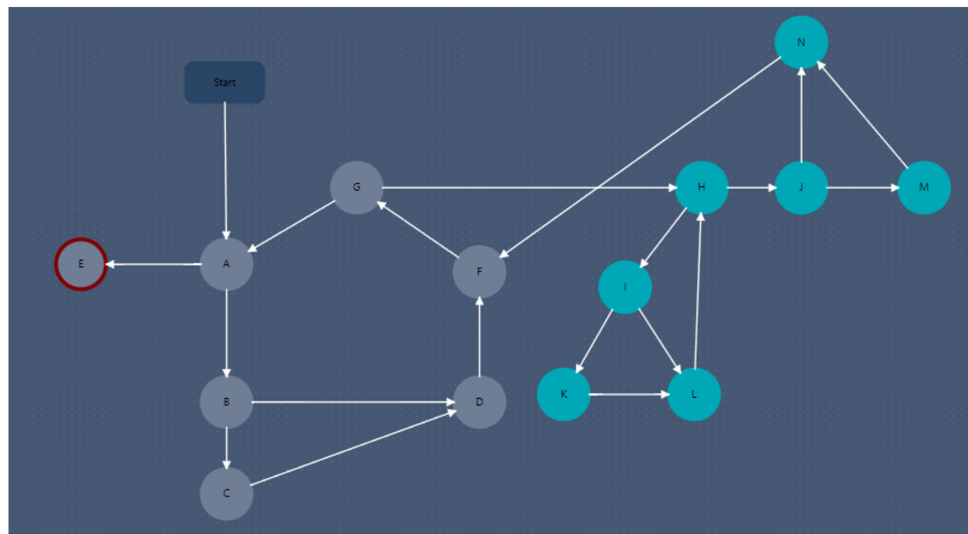

Figure C.6: System model based on `https://link.springer.com/article/10.1007/s10586-021-03291-7#Sec10`.

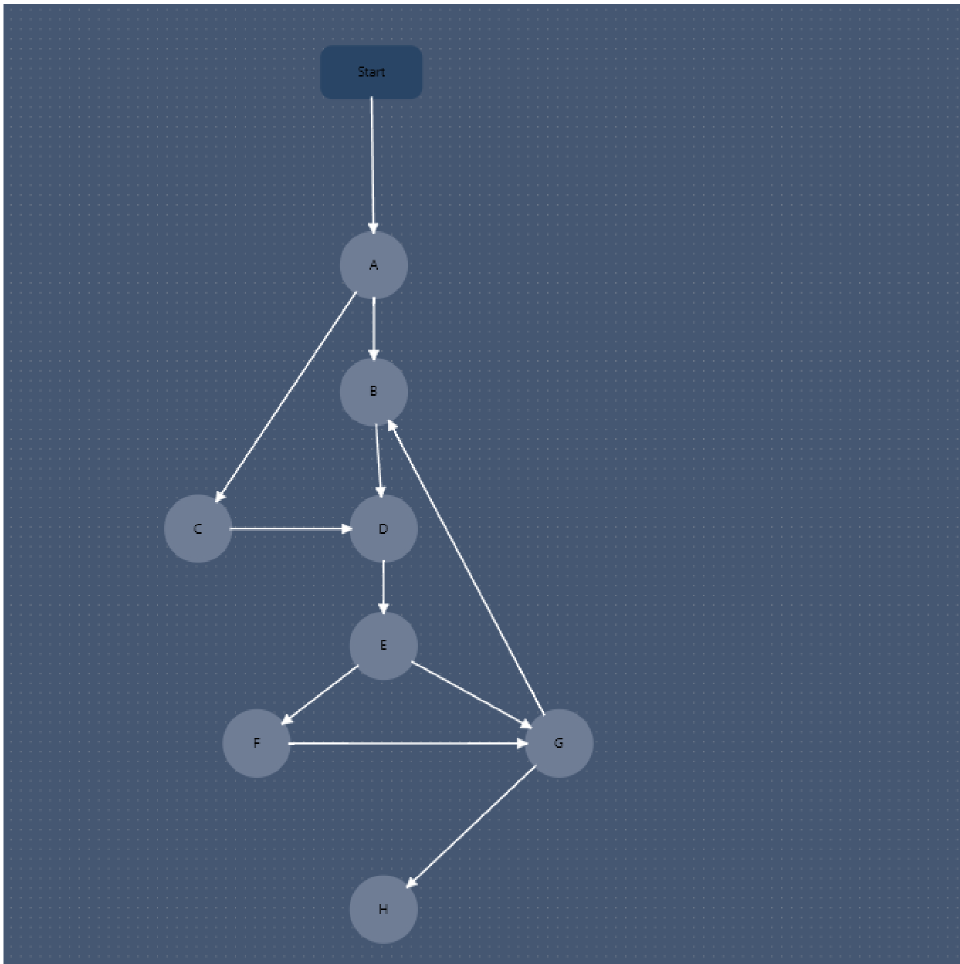

Figure C.7: System model based on `https://ieeexplore.ieee.org/abstract/document/9137867`.

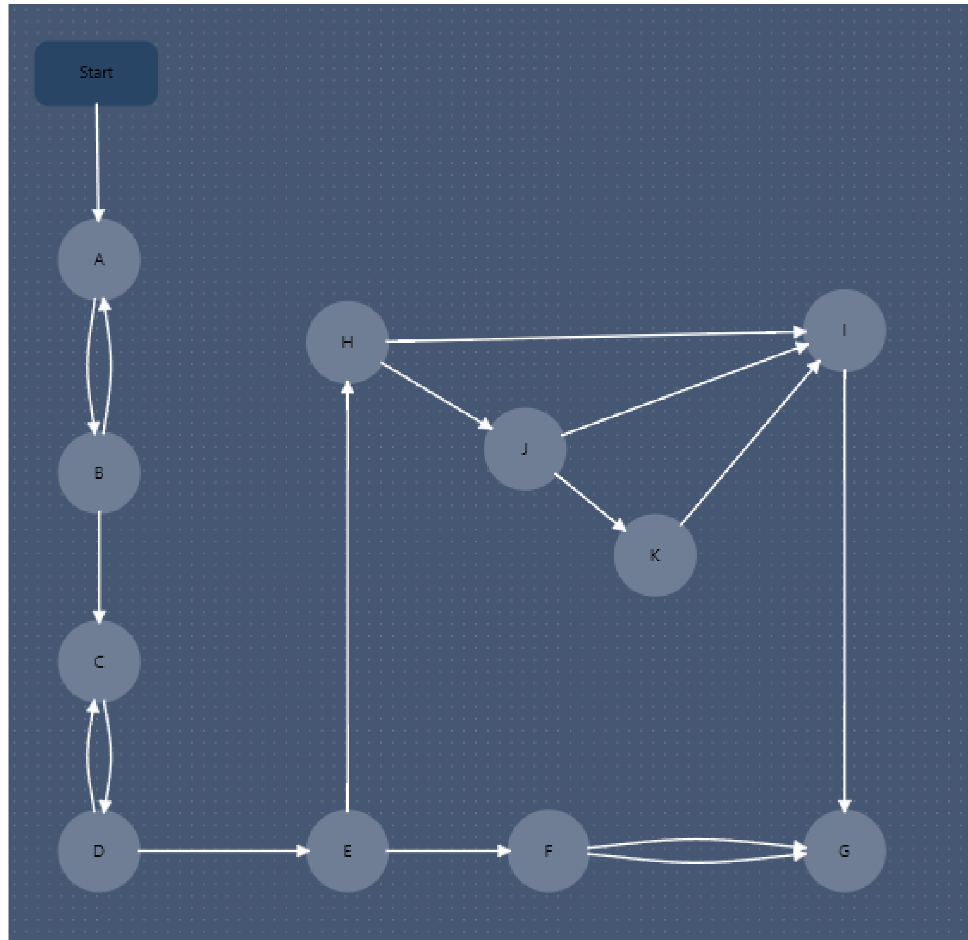

Figure C.8: System model based on `https://www.hindawi.com/journals/scn/2021/9928254/`.

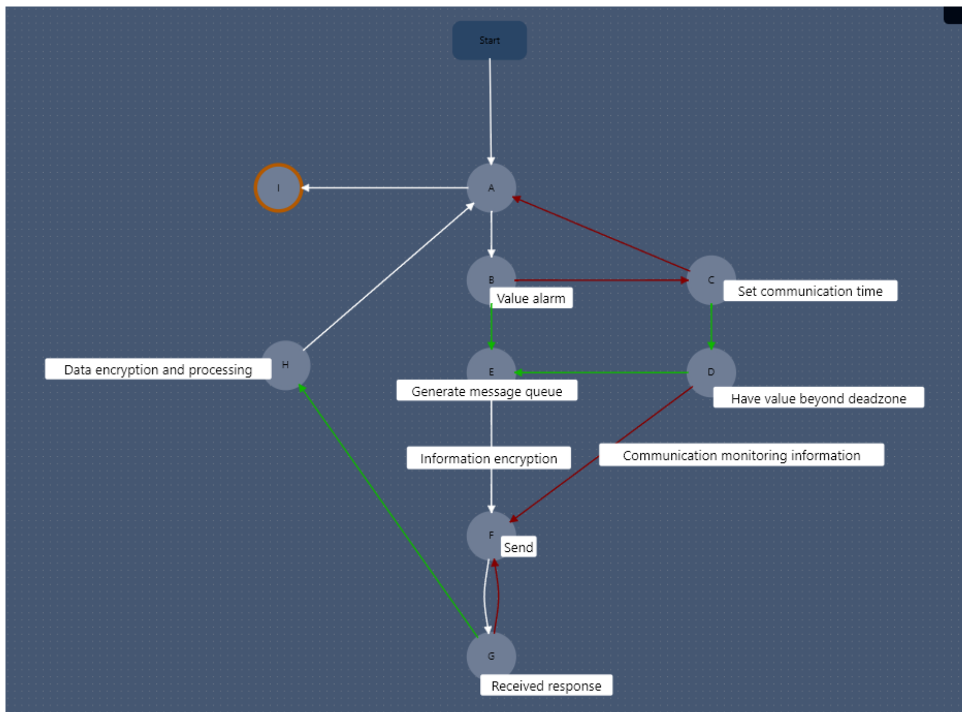

Figure C.9: System model based on `https://koreascience.kr/article/JAKO202010163509620.pdf`.

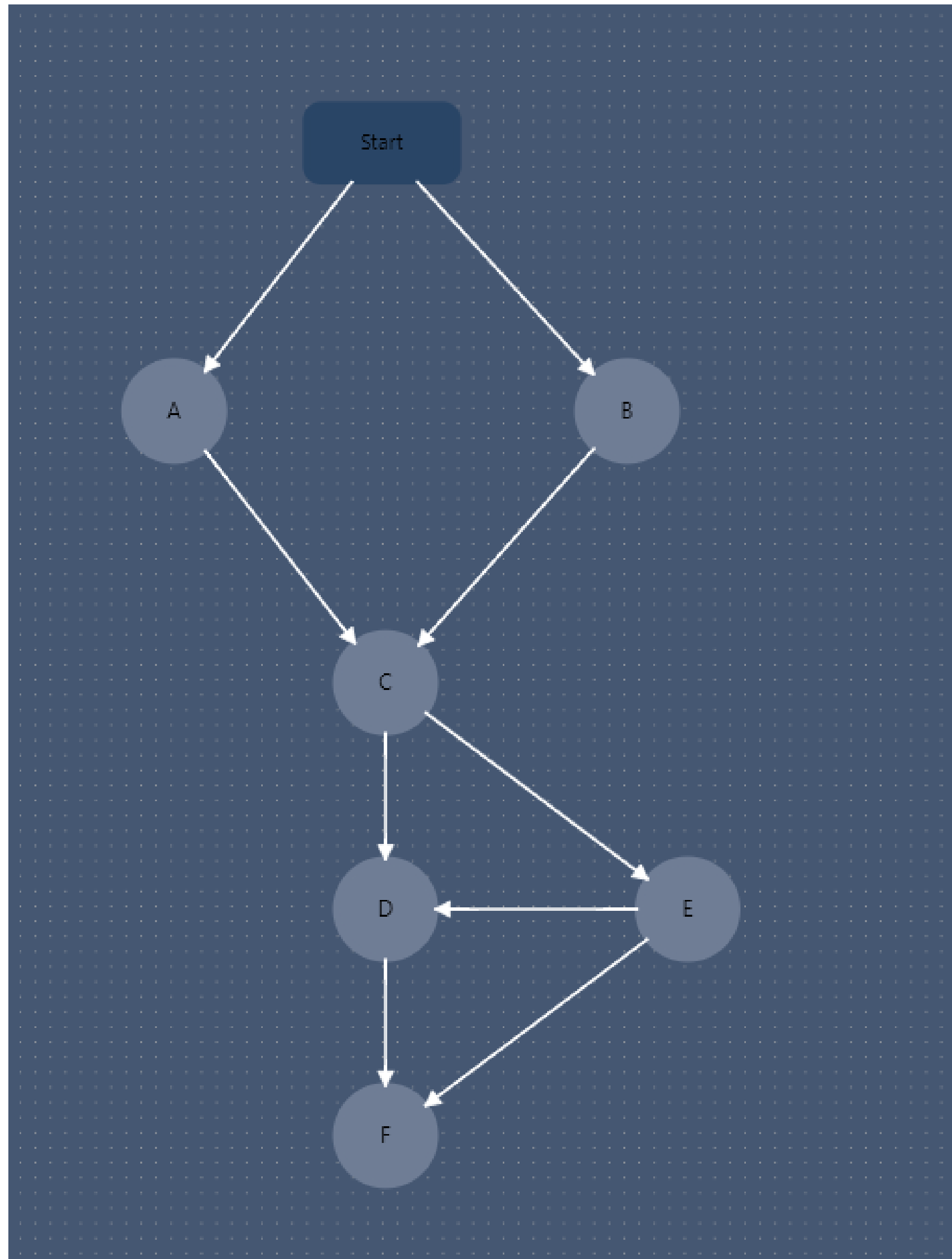

Figure C.10: System model based on `https://onlinelibrary.wiley.com/doi/full/10.1002/ett.4112`.