tuProlog Web IDE
[Intelligent System Engineering]

Alberto Donati
alberto.donati6@studio.unibo.it

Fabio Muratori
fabio.muratori2@studio.unibo.it

May 2023

tuProlog is a language and environment designed for logic programming, enabling knowledge representation, reasoning, and rule-based programming for artificial intelligence applications. These are essential components of numerous AI applications.

The objective of this project is to develop a web-based IDE[1] for the tuProlog platform. This IDE will offer users a user-friendly web environment for conducting tuProlog experiments conveniently. The project builds upon an existing implementation of tuProlog in Kotlin[1], leveraging Kotlin-based web native technologies for research and development purposes.

[1] IDE stands for Integrated Development Environment, which refers to a software application that provides comprehensive tools and features to assist programmers in developing, testing, and debugging software.

Contents

1 Requirements

We decided to follow the existing Kotlin IDE solution provided in the 2p-kt GitHub repository[1] to define the main functionalities of our solution.

Briefly, the final application should provide the following features integrated in a web page:

- theory and query editing;
- theory and query execution;
- display one, many or all solutions from given theory and query;
- save theory in a file;
- load theory from a file;
- an editor with advanced functionalities such as keyword highlighting and search, undo and redo, etc;
- management of multiple editor tabs.

Optionally, we will implement the following features on top of the ones previously specified:

- theory editor linting integration for the tuProlog language;
- messages in case of theory or query syntax errors;
- Docker containerization.

1.1 Scenarios

The main users of this application are students who wish to experiment with the Prolog logical programming language. Our goal is to offer an easy-to-use and accessible web-based solution for tuProlog, enabling users to get started by simply opening a browser tab.

Into the diagram visible at Figure 1, we can see the main users of the final system and some of the features they can access. We can note that the project doesn't need any back-end services typical of a web application. Our solution is in fact a static web page and after the initial download of dependencies, no other network calls are required.

1.2 Self assestment policy

In order to ensure that our requirements are met, we have devised a comprehensive approach that involves two key strategies to validate and verify the functionality of our solution.

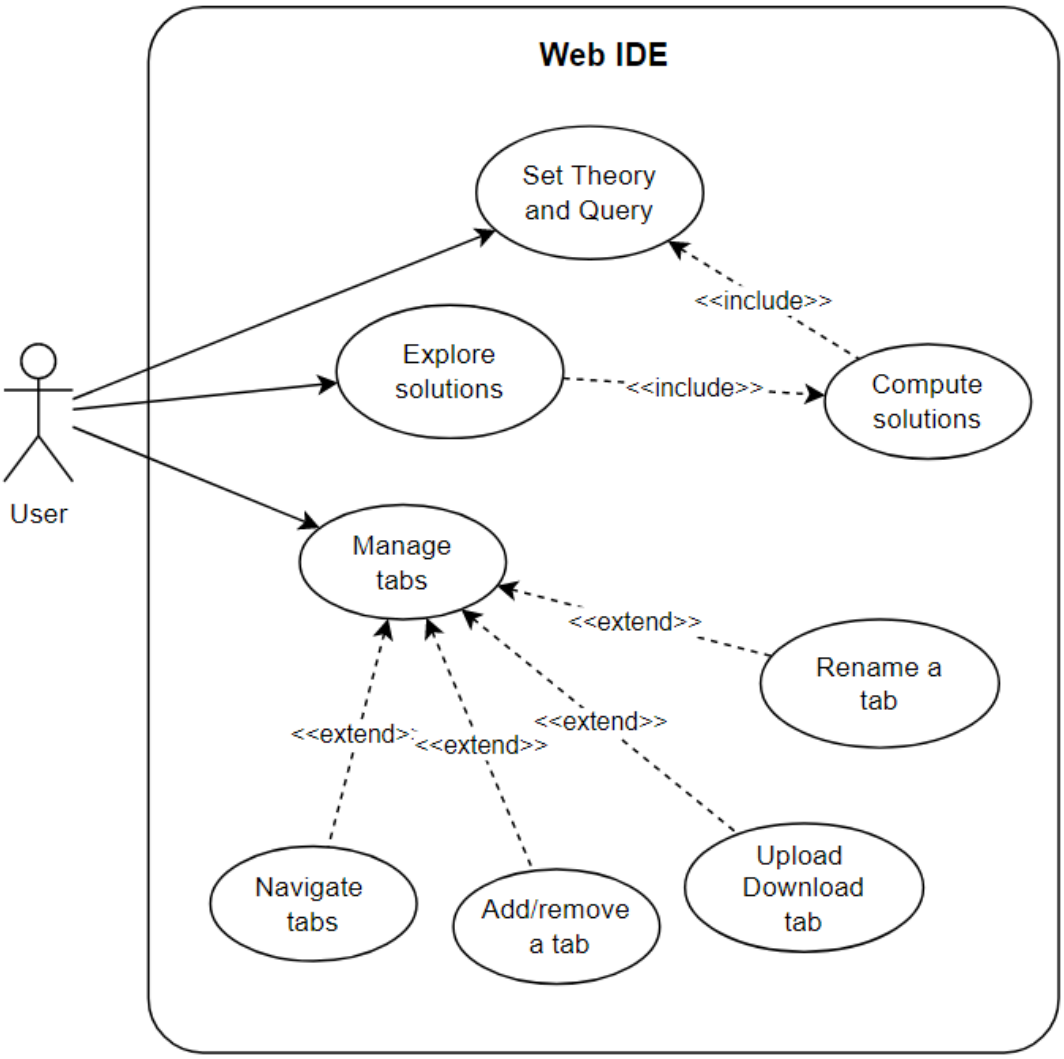

Figure 1: UML use case diagram of the main functionalities and entities.

Firstly, we will conduct a thorough comparison of our solution with the features developed by another similar application[1]. By carefully examining and analyzing the functionality offered by this benchmark application, we can ensure that our solution aligns and matches those features effectively. This comparative analysis will enable us to identify any gaps or discrepancies and make necessary adjustments to ensure the desired functionality is achieved.

Secondly, we recognize the importance of manual testing to ensure the correct behaviors of our application's functionalities. Through meticulous and systematic manual testing procedures, we will evaluate and validate the expected behaviors of each feature. This process will involve executing specific use cases, scenarios, and inputs to assess the functionality and ensure that it operates as intended. By focusing on manual testing, we can meticulously observe the system's behavior and detect any potential issues or deviations from the expected results.

2 Background

2.1 tuProlog

tuProlog[14] is a programming language and environment that belongs to the Prolog family. It is a logic programming language primarily used for developing applications in the field of artificial intelligence and knowledge representation.

tuProlog extends the standard Prolog language with additional features, libraries, and tools to enhance its capabilities. It provides a powerful rule-based language for defining facts, rules, and queries.

It provides a suitable environment for developing intelligent systems and applications in various domains.

2.2 Kotlin

Kotlin[6] is a modern programming language that was developed by JetBrains, the creators of popular development tools like IntelliJ IDEA. It is a statically-typed language that runs on the Java Virtual Machine (JVM) and can also be compiled to JavaScript or native code. Kotlin is designed to be concise, expressive, and interoperable with existing Java code.

One of the key features of Kotlin is its focus on reducing boilerplate code and improving developer productivity. It achieves this through various language constructs such as type inference, nullable types, extension functions, smart casts, and more. Kotlin also introduces innovative concepts like data classes, coroutines for asynchronous programming, and sealed classes for improved code structure and flexibility.

Kotlin is widely adopted across various domains and platforms, including Android app development. Considering that the tuProlog resource we use in our program is written in Kotlin we decided to use Kotlin/JS to easily enable the development of web based applications.

2.2.1 Kotlin/JS

Kotlin/JS[9] is a technology that allows you to write Kotlin code and compile it to JavaScript, enabling the development of web applications using the Kotlin programming language.

With Kotlin/JS, we can write code that runs in modern web browsers and server-side JavaScript environments. It provides easy interoperability with JavaScript, allowing the leveraging of existing JavaScript libraries and frameworks from your Kotlin code.

2.2.2 Kotlin Multiplatform

Kotlin Multiplatform[5] is a powerful technology that allows developers to write code that can be shared and executed across multiple platforms. It provides a unified development experience by enabling the creation of shared modules, libraries, and components that can be used in various platforms such as Android, web, and backend servers.

With Kotlin Multiplatform, developers can leverage the benefits of code reuse, reducing redundancy and increasing development efficiency. It eliminates the need to write separate codebases for different platforms, leading to faster development cycles and easier maintenance.

In the context of this project, we will interact with external Kotlin Multiplatform projects and we will need to understand their structure and implementation.

2.2.3 Gradle

In the development of the software application, one of the key technologies utilized is Gradle[3]. Gradle is a powerful build automation tool that provides a flexible and efficient approach to building, testing, and deploying software projects.

Gradle simplifies the management of project dependencies by providing a dependency resolution mechanism. It supports various dependency notations and can retrieve dependencies from repositories such as Maven Central or custom repositories. This feature ensures that the required libraries and modules are automatically downloaded and included in the project's build process.

Gradle provides robust support for multi-project builds, allowing the management of interconnected projects within a single build. This feature is particularly useful for modular applications or large codebases consisting of multiple subprojects. It facilitates sharing of common configurations, dependencies, and build logic across the project hierarchy.

2.3 Kotlin/JS Wrappers

Kotlin/JS Wrappers[8] are tools or libraries that facilitate the integration of Kotlin code with JavaScript frameworks, libraries, and APIs. They allow developers to write Kotlin code that can be transpiled into JavaScript and run in a JavaScript environment, such as web browsers or Node.js.

Kotlin/JS wrappers provide an abstraction layer that maps JavaScript APIs and functionalities to Kotlin's object-oriented syntax. This enables developers to leverage the power of Kotlin's expressive language features while working with JavaScript-based technologies.

The Kotlin/JS ecosystem includes a variety of wrappers for popular JavaScript libraries and frameworks, such as React, that we use in our project. Wrappers typically define Kotlin interfaces and classes that mirror the structure and functionality of the corresponding JavaScript APIs.

Kotlin is a statically-typed language, and the wrappers ensure type safety by providing Kotlin-specific type annotations and checking at compile-time. This helps catch potential errors early in the development process.

The JetBrains[4] organization provides a series of Kotlin/JS Wrappers for some of the well known web frameworks which can be found in a GitHub repository[8]. In our project we will use wrapped frameworks such as React, Redux and Material UI.

2.3.1 React

React[12] is a JavaScript library for building user interfaces. It was developed by Facebook and released in 2013. React is widely used for creating interactive and dynamic web applications.

One of the main features of React is its component-based architecture. React allows developers to break down the user interface into reusable and self-contained components. Each component can manage its own state, which represents the data and behavior of that particular component.

React utilizes a virtual DOM (Document Object Model) to efficiently update and render the user interface. Instead of directly manipulating the actual DOM, React creates a virtual representation of the DOM in memory. When the state of a component changes, React compares the virtual DOM with the previous version, and then applies those updates to the actual DOM. This approach significantly improves performance and makes React applications fast and responsive.

React has gained popularity due to its simplicity, performance, and large ecosystem of community-contributed packages and tools.

2.3.2 Redux

Redux[13] is a predictable state management library commonly used in JavaScript applications, particularly in conjunction with frameworks like React.

At its core, Redux follows a unidirectional data flow pattern. It emphasizes maintaining a single source of truth called the "store", which holds the entire state of the application. The state is represented as a plain JavaScript object or any data structure that can be serialized.

The key concepts in Redux are as follows:

- **Store** holds the application state and provides methods to access and modify it;
- **Actions** are plain JavaScript objects that describe an event that occurs in the application and they are dispatched by components or other parts of the application to trigger state changes;
- **Reducers** are pure functions that specify how the state should be updated in response to an action;
- **Dispatch** an action is the process of sending it to the store and trigger the execution of the corresponding reducer, which updates the state accordingly;
- **Selectors** are functions that retrieve specific parts of the state from the store and they provide an abstraction layer to access the state in a consistent and reusable way.

2.3.3 Material UI

For the graphic interface, we decided to use the MUI[10], a beautiful and simple way to create a UI, with simple components with the modern Material Design.

MUI is a popular open-source user interface (UI) framework for building web applications. It is based on Google's Material Design guidelines, which provide a set of principles and design components for creating modern and visually appealing interfaces.

The framework is written in JavaScript and primarily targets React. Material-UI provides a comprehensive set of UI components, such as buttons, inputs, menus, dialog boxes, and more, which follow the Material Design specifications.

In addition to the UI components, Material-UI also provides various utility classes, themes, and styling options to customize the look and feel of the application.

3 Requirements Analysis

As mentioned in the Background section 2, our application is based on an existing Kotlin project and we should develop a web based IDE for the tuProlog language. Therefore, it is important to maximize the utilization of the provided GitHub [1] implementation and establish a seamless interface with the original project.

In addition to the above requirement, we need to incorporate web frameworks into our Kotlin-based solution. This requirement arises from the need to deliver a web-based application within the Kotlin platform. The integration with Kotlin is itself a significant contribution to the overall outcome of this project.

Our solution builds upon a specific implementation of the 2p-kt repository which includes a well integrated interface with the tuProlog language. Moreover, our solution should operate within the context of a Model-View-Controller (MVC) architecture, where the Model and Controller components are already implemented. Our goal is to develop the View component of the MVC pattern by merging together Kotlin best practices and web frameworks as much as possible.

All these not explicitly stated requirements will be explored in the Design section 4 and multiple alternatives will be proposed with their own advantages and disadvantages.

4 Design

4.1 Design choices

The goal of this project is to develop a web-based Integrated Development Environment (IDE) using the tuProlog GitHub[1] repository as a foundation. Our primary focus will be on selecting the appropriate approach to effectively leverage the available resources in order to meet our requirements.

In particular we have to decide:

- which parts of the existing implementations should be used;
- how to interface with the chosen features.

Furthermore, the web environment offers a wide range of frameworks to choose from. It is crucial to carefully consider available options. In the subsequent sections, we will elaborate on the decisions made based on the aforementioned factors.

4.1.1 2p-kt implementation

Regarding the provided implementation, we should firstly see if multiple APIs are provided and, if necessary, chose the most appropriate one for our needs. Analysing the GitHub repository[1] and following the suggestions proposed during the planning of this project, we have found two distinct ways:

- use core modules such as :solve-classic and/or :parser-theory and implement ad-hoc interfaces;
- use a recent evolution of the code-base in the module :gui which is implemented in a developmental branch of the 2p-kt repository.

We decided to follow the second option for three reasons:

- the implementation should be easily integrated with our needs, and it is indeed meant for such cases;
- we do not need to write our own interface of the core modules and lesser need is required in understanding the underlying implementations and logic;
- we can provide a new test case of usage for the :gui module on a particularly constrained setup and further prove the effectiveness of its implementation.

The software architecture found in the :gui module consists of three main components: Application, Page, and Runner. Each Page represents a Solver and provides functionalities such as editing the Solver's staticKb, setting or executing queries, managing timeouts, and navigating through solutions. The Application serves as a container for Page objects, allowing the creation, opening, closing, renaming, and overall management of pages. It keeps track of the currently selected Page and facilitates switching

between different Pages. Both the Page and Application components follow the MVC pattern, enabling separation of concerns and handling state modifications through event listeners. Observable properties are utilized to observe and handle events, enabling updates to the View, which could be based on platform-specific requirements. There are two types of events: Event and SolverEvent. Event simply wraps the result of a task execution, whilst SolverEvent is specifically generated when an activity potentially affects the state of a Page's Solver. It provides the updated state of the Solver's observable properties.

The Runner plays a crucial role in the architecture by allocating callbacks and tasks on different threads depending on the target execution platform. It offers different methods allowing pages and apps to invoke callbacks on threads dedicated to specific activities such as background, foreground and I/O tasks.

4.1.2 How to interact with 2p-Kt

Having explored in the last chapter which starting implementation of 2p-kt should be used, we now have to understand the various options on how to effectively use desired features.

We found the following ones:

- access 2p-kt APIs as dependencies from an outside project;
- expand 2p-kt project with a new module and internally use other project modules.

The first case can then be expanded into other 2 options:

- use 2p-kt as a npm dependency and develop a full Javascript application;
- use 2p-kt as a Gradle dependency and develop a Kotlin application.

Considering the Javascript route, one advantage could be the fact that the integration with the 2p-kt Javascript porting shouldn't be too difficult. On the other hand, we have to note that the porting of Kotlin code to Javascript (through a Kotlin/JS Transpiler) does not allow to explore the tuProlog implementation with ease since the transpiled code is processed, making it hardly understandable. It seems like the intended usage of the Transpiler forces developers to design the application logic in Kotlin and lastly allow the usage of transpiled code into the Javascript environments (browser or Node.js) but not develop Javascript code on top of transpiled functionalities.

The option of develop a full Kotlin/JS application with Gradle dependencies should allow us to access the tuProlog documentation freely whilst making the integration of Kotlin/JS with Javascript frameworks more difficult. Luckily, there does exist an official porting of some well known web frameworks, like React and Material UI, that would allow us to easily access Javascript frameworks from a Kotlin environment.

Unfortunately, both the Gradle and npm options rely on the implementation available online of the 2p-kt project and we would not be able to access new features. This would

imply the need of developing our own version of tuProlog interfaces with additional efforts, or exporting the newest code as a JAR files[2] and include it in our dependencies.

This last reason pushed us to pursue the development of a new module inside the 2p-kt repository (a fork of the official one) and use the :gui module as our interface with the tuProlog core mechanisms without the need of developing our own interfaces.

Moreover, while closely working with the newest features we could easily explore the inner implementation and logic, and test this solution within our constrained requirements, working with Kotlin to develop Javascript and browser solution. Development challenges will be explored in the Section 5

4.2 Structure

Our solution follows the standard structure of a frontend web application, employing a well-established approach for organizing the codebase. The main structure of a typical solution is divided into several key components, including scripts that encapsulate specific functionalities, views that represent different UI screens, routing and navigation to facilitate seamless transitions between views, static files containing HTML and CSS for styling, APIs and utilities for data retrieval and other operations, and data management modules to handle state and data within the application. This structure ensures a logical separation of concerns and promotes modularity and maintainability throughout the development process.

In our case, we do not need to represent multiple views and routing parts, since the deliverable solution is a single page with a single user interface. The static files we should develop are only limited to an HTML document (which statically references a compiled Javascript file produced by compiling the source Kotlin/JS code) and a CSS file.

Our main focus in the design can be summarized into four components as shows in Figure 2:

- the 2p-kt :gui module which encloses all the logic on how to use the tuProlog core functionalities and adds some representation layers to the interactions;
- TuPrologController, the interface to the :gui functionalities and hooks to the controller generated events;
- React components, the representation layer that knows how to represent the core data and state;
- Redux Store, the management of the application data and how to map core data to the representation layer of the application.

For our application, we have decided to utilize React, a popular JavaScript library for building user interfaces. In addition, we have chosen to leverage Material UI, a widely-used UI framework that offers pre-designed components and styling options. The interface is organized into various structural components such as the application bar,

[2]JAR (Java Archive) files are a common file format used in Java programming for packaging and distributing compiled Java bytecode, resources, and libraries.

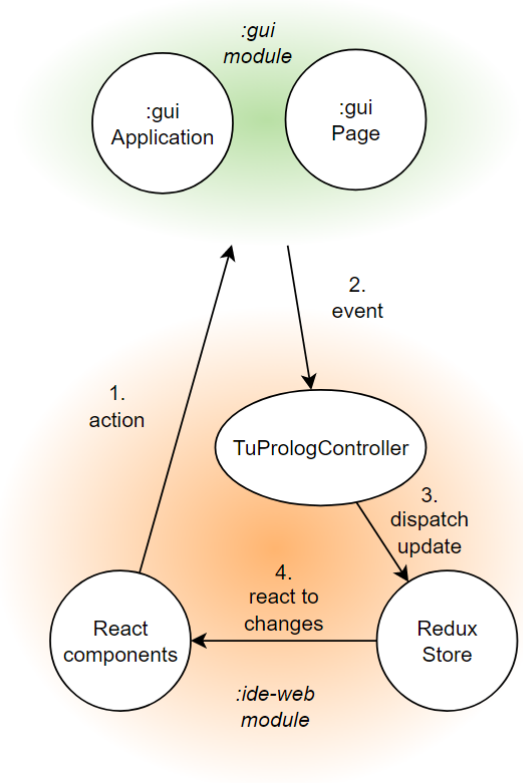

Figure 2: Representation of main components of the application.

theory and query editors, action buttons, and solutions views. Each of these components can further implement a hierarchy of nested components, providing a modular and scalable structure to our application's graphical representation.

In our application, the management of data could be implemented through two different approaches. In the first case, the state is maintained in the root React component and then passed down to its children components using the props mechanism. This approach allows for a more localized state management within the component hierarchy. On the other hand, the second case involves utilizing Redux, a popular state management library. Redux enables a centralized and global state management approach, where the state can be accessed from anywhere in the application. This approach provides more flexibility and scalability, especially for larger applications with complex data flows.

4.3 Behaviours and Interactions

The desired interactions of our system, using the Figure 2 as a reference, could be summarized in Figure 3. We can see how the execution of an action is firstly caught in the front-end and directly communicated to the Application or Page instances in the :gui module. These components will eventually communicate the result of an action via a set of hooks registered in the TuPrologController that we defined in our solution. The TuPrologController manages the behaviours on task completion and provide new data to the Redux state manager, which can automatically update the graphical components using the mutated data.

This approach allows for:

- usage of :gui module functionalities from anywhere in the application;
- centralize the management of updates and state changes;
- remove the need of data passing from component to component;
- decouple the source of an action and UI changes.

It is important to note that a React component should initially subscribe to a particular Redux store variable. Whenever the observed variable changes, the framework allows for automatic updates inside the component.

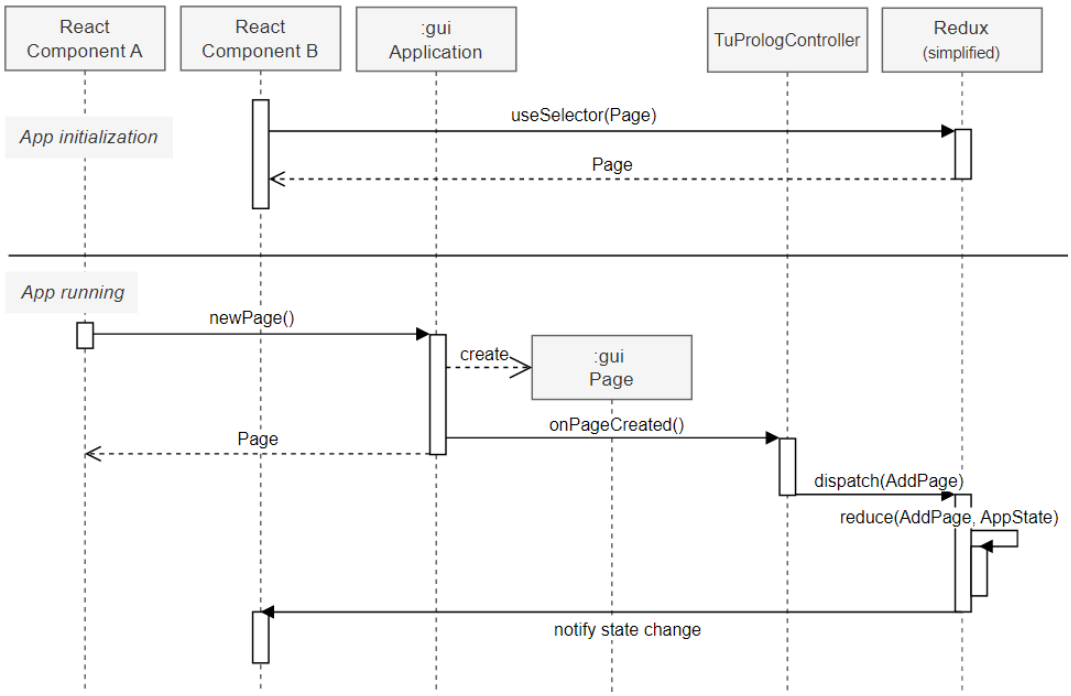

Figure 3: UML sequence diagram of a typical interaction between main components of the application.

5 Implementation Details

This project uses Kotlin/JS, a technology that allows us to write Kotlin code that can be compiled into JavaScript, thus allowing the development of web applications using the Kotlin programming language.

The simplified workflow of such a developmental process can be represented as follows:

1. use the Kotlin programming language to write the application logic;
2. interact with JavaScript APIs and modules allowing interoperability with existing JavaScript code and libraries;
3. compile Kotlin to JavaScript using the Kotlin/JS Transpiler to generate code that can be executed in a browser or any JavaScript environment;
4. bundle and optimize the JavaScript code and any dependencies using Webpack;
5. run the Kotlin/JS application using a web server and test it in a browser environment.

5.1 Integration with Javascript

Kotlin/JS wrappers are libraries or frameworks that provide Kotlin-friendly APIs and abstractions around existing JavaScript libraries or APIs, allowing us to easily interact with JavaScript code.

The most relevant packages we used are React, Redux, Material-UI, Emotion CSS[3], and Monaco Editor [4].

Most of those resources are provided by GitHub Kotlin/JS wrappers[8] and are already usable from a Kotlin environment without particular effort. In other cases, such as the Monaco Editor, we had to directly interface with Javascript libraries through the Node Package Manager (npm) and build our own wrappers to allow the usage of an advanced text editor from Kotlin code.

5.1.1 Monaco Editor wrapper example

To enable the usage of the Monaco Editor[11] component from Kotlin/JS, we needed to create a wrapper that provides a Kotlin-friendly API for interacting with the component.

To create it and define an external Kotlin interface that represents the props of the React component, we had to mirror the props used by the original JavaScript implementation.

The simplified result of our solution is listed below:

[3]Emotion CSS is a popular CSS-in-JS library for styling React components that provide a way to write CSS styles directly in Kotlin/JS

[4]Monaco Editor is a Kotlin/JS wrapper for the Monaco Editor, which is a feature-rich code editor that can be embedded in web applications

```
@file:JsModule("@monaco-editor/react")
@file:JsNonModule

import react.ComponentClass
import react.Props

external interface EditorProps : Props {
    var value: String
    var height: String
    var onChange: (String) -> Unit
    var theme: String
}

@JsName("default")
external val MonacoEditor: ComponentClass<EditorProps>
```

The following code shows how the newly created wrapper can be used inside a Kotlin React component:

```
val TheoryEditors = FC<Props> {
    ...
    editorTabs.forEach { (entry: Map.Entry<Page,
        PageWrapper>)
        ...
        MonacoEditor {
            value = entry.value.theory
            onChange = { (text: String) ->
                entry.key.theory = text
            }
            if (provTheme == Themes.Dark) {
                theme = "vs-dark"
            } else if (provTheme == Themes.Light) {
                theme = "vs-light"
            }
        }
    }
    ...
}
```

The logic behind this particular implementation is that for each object Page, we create a `MonacoEditor` component but only the current page editor is visible. Section 5.2 shows how the `editorTabs` variable is created and how we can constantly show an updated state.

5.2 Redux state management

To enable the usage of Redux functionalities in our project we used a Kotlin/JS wrapper implementation provided by a GitHub repository[7]. The integration of the framework required the definition of some constructs.

Firstly, we defined what the Store of the web application should contain, keeping in mind the state of :gui module and data extracted during the interactions.

The following Kotlin snippet shows the entire state:

```
data class PageWrapper(
    var id: PageID,
    var theory: String,
    var query: String,
    var solutions: Collection<Solution>,
    ...
) {
    companion object {
        fun fromPage(page: Page): PageWrapper {
            return PageWrapper(
                // definition of initial values
            )
        }
    }
}

// main container of the application state
data class TuProlog(
    var pages: MutableMap<Page, PageWrapper>,
    var currentPage: PageWrapper?,
)

data class AppState(
    var tuProlog: TuProlog
)
```

The code above is needed to expand the concept of Page of the :gui module to match our needs and make the state reactive during the execution.

The `Page`, `Solution`, `PageID`, `PageStatus` classes are references to the :gui module implementation.

An example of a Reducer interaction with a `PageWrapper` is shown below. The following snippets show what happens to the state when the user clicks on the "Add Page" button in the UI.

Initially the action is propagated to the :gui module Application and Page objects:

```
Button {
    ...
    onClick = {
```

```
        TuPrologController.application.newPage()
    }
    ...
}
```

As soon as the task is completed, the TuPrologController object catches the resulting event.

The following example shows how the Application and Page objects are observed by React components and how to interface with the Redux store to dispatch a state update:

```
object TuPrologController {
    var application: Application = Application.of( ... )
    private var pageBindings: List<Binding> = emptyList()
    private lateinit var store: Store<AppState, RAction,
        WrapperAction>

    private fun bindApplication(application: Application)
        {
        ...
        application.onPageCreated.bind {
            store.dispatch(AddPage(it.event))
            bindPage(it.event)
        }
        ...
    }

    private fun bindPage(page: Page) {
        ...
        pageBindings += page.onNewSolution.bind ...
        pageBindings += page.onStateChanged.bind ...
        pageBindings += page.onQueryChanged.bind ...
        pageBindings += page.onTheoryChanged.bind ...
        ...
    }
```

Regarding the previous example, when a Page is created, a series of bindings are registered. The variable `pageBindings` is necessary to safely manage bindings when the selected UI page changes, since we do not want to listen events from a closed Page.

Here we can see how the Redux action is reduced and converted to a state change:

```
class AddPage(val page: TPPage) : RAction

fun tuPrologReducer(state: TuProlog, action: RAction):
    TuProlog =
    when (action) {
        ...
        is AddPage -> {
```

```
7               state.pages += (action.page to PageWrapper.
                    fromPage(action.page))
8               state
9           }
10          ...
```

Finally, when a React component uses the `state.pages` variable, an UI update is propagated.

When a component wants to use a Redux state variable and react to its changes, we use the following instructions:

```
1
2   val editorTabs = useSelector<AppState, MutableMap<Page,
        PageWrapper>> { s -> s.tuProlog.pages }
3
4   val editorTabsNames = useSelector<AppState, Collection<
        PageID>> { s -> s.tuProlog.pages.keys.map { it.id } }
```

5.3 Webpack configuration

Kotlin/JS aims at the development of projects for both browsers and Node.js platforms and, depending on the target environment, several key differences must be considered. These differences arise due to the distinct purposes of these platforms. When developing for the browser, the code is executed on the client-side, within the user's web browser. On the other hand, Node.js allows JavaScript code to be executed on the server-side.

In the browser, JavaScript runs within a sandboxed environment, which restricts access to certain resources and APIs for security reasons. In Node.js, JavaScript has access to a broader range of resources and APIs, including the file system, networking capabilities, and operating system functionalities.

During the application's building phase, we encounter an error due to the usage of ANTLR[5] within 2p-kt implementation:

```
1   Module not found: Error: Can't resolve 'fs' in 'C:\Users\
        Fabio\Desktop\ise_workspace\2p-kt\build\js\
        node_modules\antlr4\src\antlr4'
```

The missing module `fs` allows reading, writing, and manipulating files and directories. In a browser environment, JavaScript has limited access to the file system for security reasons and the browser enforces strict measures to protect the user's system and data. JavaScript running in the browser generally does not have direct access to the file system. It cannot read or write files on the user's machine without explicit permission.

[5] ANTLR stands for Another Tool for Language Recognition. It is a powerful parser generator that allows developers to build language parsers and interpreters.

We were able to fix the issue by disabling certain configurations which were causing the problem during the bundling of Javascript files using Webpack.

The solution we found is that, by creating a folder `webpack.config.d` and adding a file named `fs.js`, we were able to define a specific Webpack configuration which ignores the problematic dependencies and skip the resolution of these missing modules.

The content of the newly created file is shown below:

```
config.resolve = {
    fallback: {
        fs: false,
        path: false,
        os: false,
    }
};
```

The given Webpack configuration sets the resolution behavior for certain modules in the project. Specifically, it adds a fallback property that specifies the behavior for resolving modules when they are not found in the usual paths. The three configurations for `fs`, `path` and `os`, indicate that when the module resolver encounters a module that requires them, it should not use a fallback mechanism. The false value indicates that if the module is not found, it should not attempt to provide a substitute or fallback implementation for the corresponding module.

5.4 Development choices

In the development of the web IDE we use a different logic for the management of multiple text editors and other components in the application.

Initially, to represent the tuProlog theory, we used a single React component and dynamically changed its value based on the currently selected Page object. However, to incorporate advanced editor functionalities (such as Undo and Redo), we adopted a different approach where multiple editors are created and managed, with each editor dedicated to a specific Page instance of the :gui module.

Thanks to this new logic, we have a one-to-one mapping between DOM [6] tree and open tabs. Each tab in the IDE corresponds to an editor instance that manages the content and interactions specific to it. Only one editor is displayed and when the user switches between files, the active editor dynamically updates to reflect the content of the selected file.

Other React components, such as the query editor, do not implement the same logic and instead whenever the selected page changes, the linked query is retrieved from the Redux state and it is displayed on the already created component.

[6] DOM stands for Document Object Model, a programming interface that represents the structure and content of a web page.

6 Deployment and deliverables

The project was developed as a fork[2] of the 2p-kt GitHub repository, which served as the foundation for our work and provided a valuable resource throughout the development process.

As a part of the project's deliverables, we focused on two key goals: the GitHub project itself and the pull request to submit to the original repository. Our GitHub fork served as a centralized location for managing and tracking the development progress.

Furthermore, the pull request was an essential deliverable that showcased our contributions to the original 2p-kt repository. By submitting a pull request, we aimed to integrate our changes and improvements into the main project.

6.1 Build the application

To build the application, we utilized the following Gradle command:

```
.\gradlew :ide-web:build
```

This command triggers the build process specifically for the "ide-web" module and other project dependencies. It ensured that all dependencies are resolved, tests are executed, and the necessary artifacts are generated for the application.

After a successful build, the web page distribution of the application can be found in the `:ide-web/build/distributions` directory. This directory contains the necessary files and resources required to deploy and host the web page. It provides a packaged version of the application that could be easily distributed.

To run the application, we used the following Gradle command:

```
.\gradlew :ide-web:run
```

This command executes the application and deploys a local web server to access the web tuProlog IDE. It allows us to interact with the functionality we implemented and test its behavior in a runtime environment.

7 Usage Examples

An usual interaction requires the user to open the browser and navigate to the web application.

From top to bottom, the application is composed by:

- an application bar with buttons for pages management;
- a series of tabs with their own text editors;
- a text field where the query must be defined and buttons on the left to start, stop and reset the resolution of a query;
- a bottom container with the results of an execution and additional information regarding the context of the currently displayed page;
- a footer with additional information and configuration options.

Figures 4 and 5 show the described structure with test theory, query and solutions.

The functionalities that users can access to manage tabs are:

- add a new tab;
- rename the currently opened tab;
- upload a theory from the user file system;
- download the currently selected tab theory;
- delete the selected tab.

To switch between tabs, the user can click on the desired tab in the interface. As soon as the tab changes, all the other components described above are updated as well.

Finally, Figures 6, 8 and 7 show the possible outcomes when the "Solve" buttons are pressed.

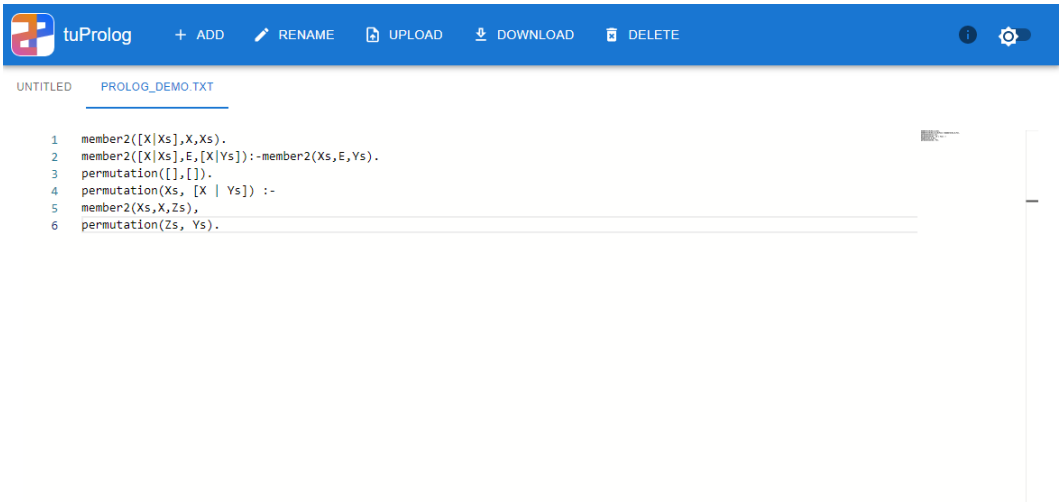

Figure 4: The running application top part.

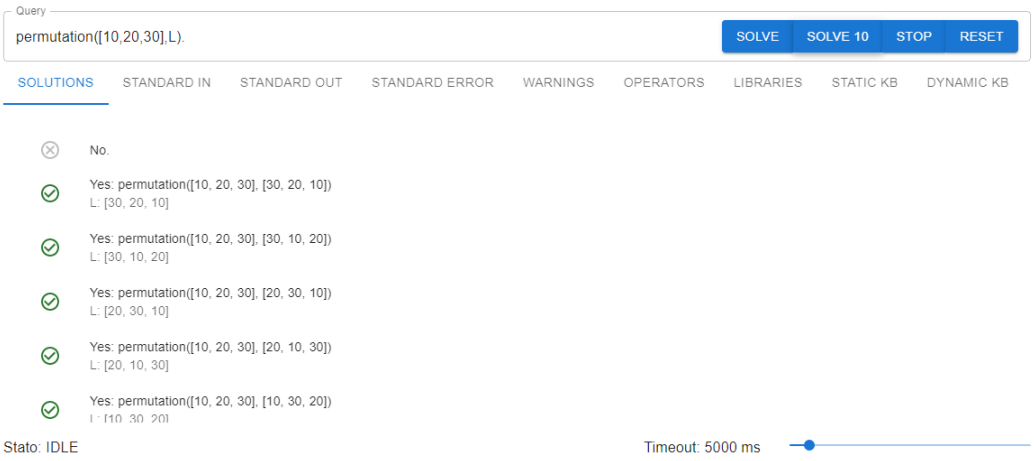

Figure 5: The running application bottom part.

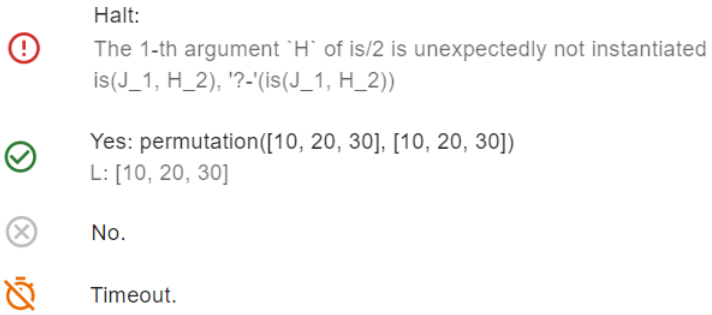

Figure 6: Example of all the possible solutions displayable.

Figure 7: A message displayed when a theory syntax error is found.

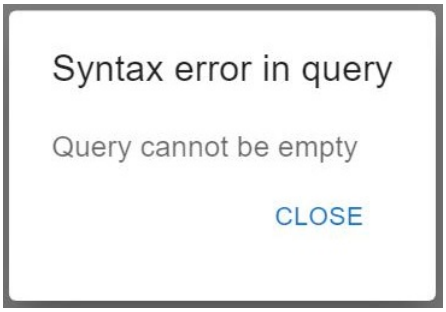

Figure 8: A message displayed when the query is missing.

8 Conclusions

With this project we realized a Web IDE for the tuProlog language and environment. We achieved most of the features we wished to realize and added other functionalities to improve the usability of the system even further, while trying to maintain software engineering best practices.

Also, we were able to effectively integrate a series of technologies in a novel environment.

8.1 What did we learn

Through this project, we gained valuable experience in web development by leveraging a diverse range of technologies such as React, Redux, and Kotlin/JS. We have successfully integrated these technologies and acquired proficiency in their use. Additionally, we became familiar with the structure of 2p-kt and effectively employed its controller for creating the web version of the application.

Furthermore, we have expanded our understanding of the tuProlog programming language, allowing us to enhance our proficiency in this domain. Additionally, our knowledge of web development toolkit for browsers has significantly improved, enabling us to create more robust and efficient web applications.

In terms of version control, we have sharpened our skills in using Git and GitHub. We have increased expertise in employing different branches for distinct scopes, enabling more organized and collaborative development processes.

8.2 Future development

In future evolutions, we think the following features could be implemented to further improve the functionality and usability of the application:

- introduce a tree view feature that provides a visual representation of solutions;
- export functionality to save the list of solutions in various formats;
- incorporate a debugger functionality for an effective troubleshoot of issues;
- improve libraries management within the tuProlog environment;
- code linting and decoration for Monaco Editor;
- deploy the application on a real domain.

List of Figures

References

[1] 2p-kt. https://github.com/tuProlog/2p-kt. Accessed: May 2023.

[2] 2p-kt project repository. https://github.com/fmuratori/2p-kt/tree/feature/web-gui. Accessed: May 2023.

[3] Gradle. https://gradle.org/. Accessed: May 2023.

[4] Jetbrains. https://www.jetbrains.com/. Accessed: May 2023.

[5] Kotlin multiplatform. https://kotlinlang.org/docs/multiplatform.html. Accessed: May 2023.

[6] Kotlin programming language. https://kotlinlang.org/. Accessed: May 2023.

[7] Kotlin wrapper redux. https://github.com/JetBrains/kotlin-wrappers/tree/master/kotlin-react-redux. Accessed: May 2023.

[8] Kotlin wrappers. https://github.com/JetBrains/kotlin-wrappers. Accessed: May 2023.

[9] Kotlin/js. https://kotlinlang.org/docs/js-overview.html. Accessed: May 2023.

[10] Material-ui. https://material-ui.com/. Accessed: May 2023.

[11] Monaco editor. https://www.npmjs.com/package/@monaco-editor/react. Accessed: May 2023.

[12] React. https://reactjs.org/. Accessed: May 2023.

[13] Redux. https://redux.js.org/. Accessed: May 2023.

[14] tuprolog. https://apice.unibo.it/xwiki/bin/view/Tuprolog/. Accessed: May 2023.