UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Kaarel-Richard Kaarelson

TeMoto Action Assistant: A Web-Based Human–Robot Interface for Designing UMRF Graphs

Bachelor's Thesis (9 ECTS)

Supervisors:
Karl Kruusamäe, Robert Valner

Tartu 2025

TeMoto Action Assistant: A Web-Based Human–Robot Interface for Designing UMRF Graphs

Abstract:

This thesis develops the TeMoto Action Assistant, a modern, web-based graphical user interface (GUI) designed to improve human-robot interaction within the TeMoto framework. It aims to simplify the creation and management of Unified Meaning Representation Format (UMRF) graphs, to speed up developers' time. Significantly, it uses a web-based architecture that many current ROS 2-based human-robot tnterface (HRI) solutions do not address. The proposed solution uses React, Python Flask, and the WebSocket protocol for real-time synchronization with the TeMoto Action Engine. Significantly, it provides GUI functionalities like creating robot tasks via drag-and-drop, editing action parameters, and built-in runtime execution and monitoring. By providing an open-source, multi-platform GUI, this work lowers the barrier for entry for robotic application development and hopes to expand the adoption of the TeMoto framework.

Keywords: robotics, TeMoto, Unified Meaning Representation Format, UMRF, ROS 2, graphical user interface, human-robot interaction, React, Open-source, Multi-Robot Systems

CERCS: T125

TeMoto Action Assistant: Veebipõhine Inimese ja Roboti Interaktsiooni Tööriist UMRF Graafide Loomiseks

Lühikokkuvõte:

Käesolev lõputöö arendab välja veebipõhise graafilise kasutajaliidese (GUI), TeMoto Action Assistanti, mis on loodud inimese ja roboti interaktsiooni (HRI) edendamiseks TeMoto tarkvararaamistikus. Täpsemalt, lihtsustab see Unified Meaning Representation Format (UMRF) graafide loomist ja haldamist, et kiirendada arendajate tööd. Selle eeliseks on veebipõhine arhitektuur, mida olemasolevad ROS 2-põhised GUI lahendused ei rakenda. Arhidektuuri tasemel kasutab veebiäpp Reacti, Python Flaski ja WebSocketi protokolli andmeedastuseks, et reaalajas suhelda TeMoto Action Engine'iga. Veebiäpp pakub olulisi funktsionaalsusi nagu robotite ülesannete loomine GUI kaudu, tegevuste parameetrite muutmist ning nende käivitamist. Avatud lähtekoodiga lahendusena on selle eesmärk alandada tehnilist lävendit robootikasüsteemide arendamisel ja laiendada TeMoto tarvkararaamistiku kasutavate arendajate hulka.

Võtmesõnad: robootika, TeMoto, Unified Meaning Representation Format, UMRF, ROS 2, graafiline kasutajaliides, inimese ja roboti interaktsioon, React, avatud lähtekood, mitme roboti süsteemid

CERCS: T125

Table of Contents

1. Introduction

This Chapter gives introction into the problem space of building robots for hazardous environments. It consists of four parts: Section 1.1 introduces the background and motivation of the study. Section 1.2 defines the problem statement. Section 1.3 analyses the significance of this thesis.

1.1. Background and Motivation

Across several domains like scientific research, space exploration, and disaster response, there are tasks that humans cannot safely or efficiently perform due to environmental dangers or physical limitations. For instance, particle accelerator facilities such as the European Organization for Nuclear Research (CERN) can expose humans to hazards like radiation, electrical shocks, and oxygen deficiency [1]. Another example is space exploration, where extraterrestrial environments like the Moon and Mars present significant challenges such as extreme temperatures and the limitations of human spaceflight [2]. Therefore, using non-biological systems like autonomous or semi-autonomous robots can effectively mitigate human risk and increase operational efficiency in performing such tasks.

In many hazardous environments, having the ability to remotely control the robot is desirable. Teleoperation has been used as a semi-autonomous method to enable human supervision of a robot from a distance, such as in nuclear decommissioning applications [3]. The sensor information and supervisor's control commands are transmitted through a communication network, but such data communication can involve time-varying delays or connection loss [4]. Thus, achieving stable connectivity can be a challenge, which may limit teleoperation availability. The underlying software is vital in handling those challenges. Even with continuous advancements in robot software tools, developing advanced frameworks remains challenging [5], [6]. There are existing software frameworks like Robot Operating System (ROS) [7] and YARP [8], but they can be too general for domain-specific scenarios. This necessitates reliable domain-specific software frameworks for robots in hazardous environments to navigate such challenges with autonomy.

Recognizing the need for dependable and adaptive software solutions for autonomous robots in high-risk and high-complexity task domains, Valner et al. developed the TeMoto software framework [9]. It provides a software toolkit and, most importantly, a task management capability for robots in hazardous environments [9], making it a domain-specific solution. The existing alternative software is often tied to a specific programming language and does not

support dynamic tasks for more complex robot behaviors [9] limiting their flexibility. Importantly, it uses Unified Meaning Representation Format (UMRF) to define robot tasks, to provide adaptive and multi-agent, and Human-Robot Interface (HRI) oriented design [9], as detailed more in Section 2.2. Therefore, it's a significant and novel improvement that could be used with robots in more complex settings.

1.2. Problem Statement

While the TeMoto framework provides a robust foundation for developing autonomous robotic applications, creating and managing UMRF graphs can present a steep learning curve and manual labor for robot developers. To address this, this thesis focuses on developing a user-friendly graphical user interface (GUI) that simplifies the process of creating UMRF graphs. This tool aims to lower the barrier to entry for robot developers wishing to leverage the capabilities of the TeMoto framework for their projects.

1.3. Significance of the Study

This thesis contributes to the field of robotic software development by providing a user-friendly tool, TeMoto Action Assistant, that goal is to lowers the barrier to entry for utilizing the powerful capabilities of the TeMoto framework. By simplifying the creation and management of UMRF graphs, the developed design tool has the potential to enhance the productivity of robot developers by reducing the time and effort required to define complex robotic tasks. Moreover, delivering such tool has potential to make TeMoto more accessible to developers with varying levels of expertise and broaden its adoption.

2. Related Work

This section provides an overview of the relevant software that supports the technical contributions presented throughout this thesis. In addition, it analyses existing GUIs of task management tools, commonly used with ROS-based software frameworks. The chapter is organized into five subsections: Section 2.1 introduces Robot Operating System 2 (ROS 2), a widely used open-source framework for building robot applications. Section 2.2 focuses on the TeMoto software framework, an extension of ROS 2, which this work directly contributes to: section 2.3 presents React, a popular frontend framework for developing modern user interfaces. Section 2.4 reviews relevant GUIs of other task management frameworks; Section 2.5 outlines the challenges encountered when creating UMRF graphs with current tools inside the TeMoto framework.

2.1. ROS 2

Robot Operating System 2 (ROS 2) is an open-source software development kit (SDK) designed for developing robot applications [7]. The ecosystem is divided into three categories: middleware (e.g. network APIs), algorithms (e.g. perception and planning) and developer tools (e.g. visualization, simulation and debugging) [7]. It uses a communication technology Data Distribution Service (DDS) [5], which is a middleware that helps to provide reliable data transfer between devices [10]. The primary goal of ROS 2 is to simplify robot application development and lower the barrier of entry for building robotic systems in industry and research

ROS 2 employs a distributed systems architecture, where devices are independent components or programs with separate contexts that can share data [7]. Collectively, these components conform to a computational graph called a node [7]. These can encapsulate the following communication patterns as detailed below:

a) **Topic** is an asynchronous message channel framework that is most frequently used to communicate between nodes [7]. It uses a publisher-subscriber method with strong type interfaces of the messages [7], reducing data validation errors. The main advantage is that it allows many-to-many communication between robots [7], scaling well with an increasing number of robots. Lastly, a developer can create a subscription to the topic to access the message, without any modification to the topic itself [7], which allows the client to complete other processes while waiting for the reply.

b) **Service** is a more traditional communication protocol that allows a client node to send a request and receive a response from the server node [7]. ROS 2 is designed so that the request is non-blocking [7], which allows the client to complete other processes while waiting for the reply.

c) **Action** is an asynchronous goal-oriented communication pattern [7] that, in comparative terms, is the most distinctive of the three. Notably, it provides periodic feedback between the nodes and the ability to be cancelled [7]. It benefits robotics applications where some processes may take too long or plans may change.

ROS 2 architecture is distributed across packages so that users can only use the pieces of the software they require or replace components themselves [7], while also being able to contribute to the ecosystem. The modularity and standardized communication patterns enable ROS 2 to be adapted to various applications, from embedded systems to large-scale multi-robot deployments.

2.2. TeMoto Software Framework

TeMoto is a software framework for adaptive, scalable, multi-agent, and human-robot collaboration-oriented robot applications [11]. It was developed to address the needs for high-risk task domains while following a standard robot software stack [11]. Moreover, it is an open-source standard integrating with ROS and ROS 2 [11].

At its core, TeMoto uses a decentralized three-layer architecture that includes an operational, functional (resource management), and executive layer (task management) [11]. In the executive layer, tasks are a combination of modular behaviors called actions [12]. Tasks are represented as the Unified Meaning Representation Format (UMRF) graphs, a novel domain-specific language based on JSON [11]. Notably UMRF graphs enable the combination sequential, concurrent and cyclical actions [11], allowing applications of all levels of sophistication. Moreover, it allows actions to be sub-graphs, supporting hierarchical graphs [11]. Lastly, it supports action parametrization, conditionals, error management and shared multi-agent graphs [11] as outlined as examples in Figure 1.

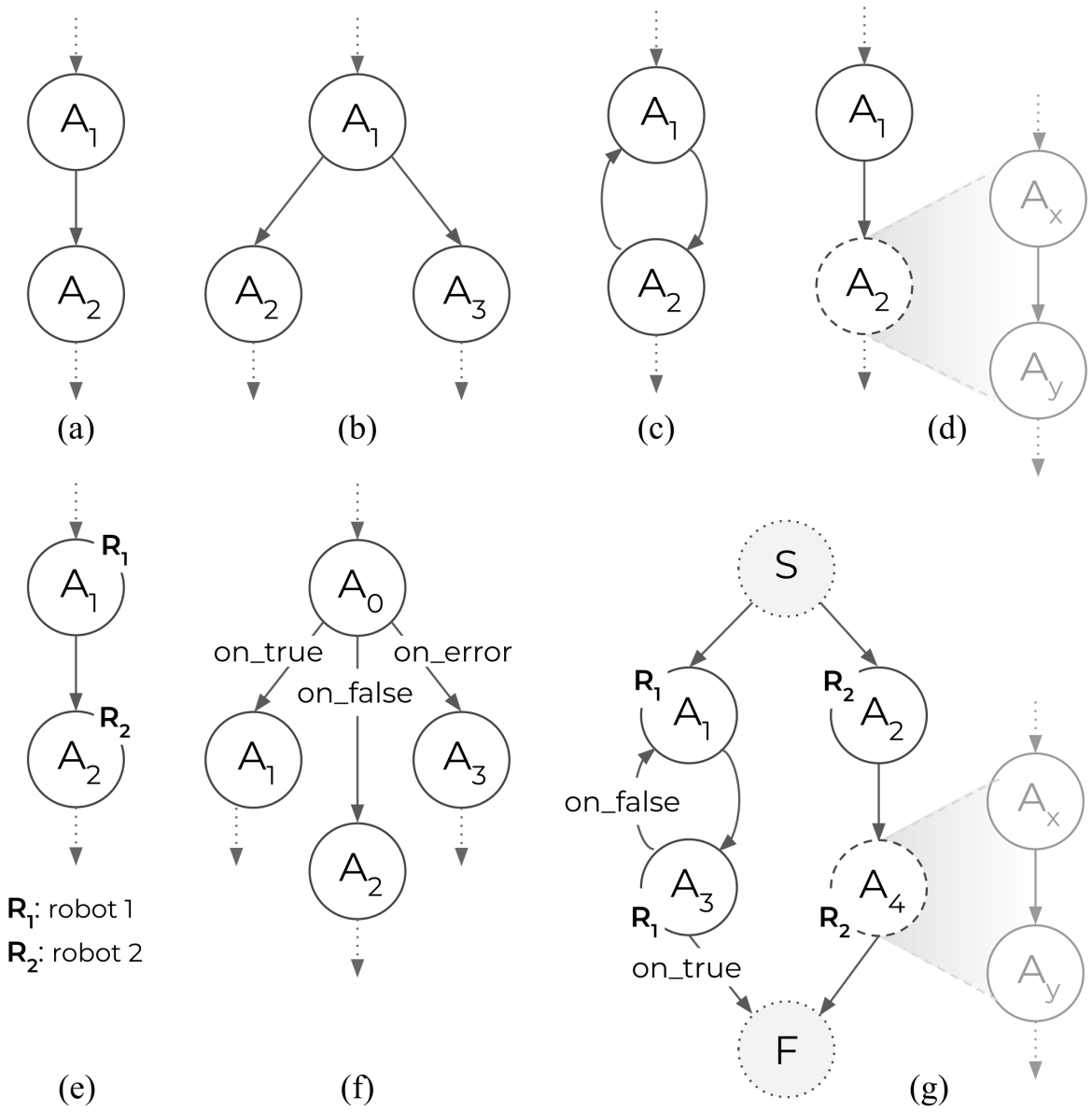

Figure 1. "Main semantic properties of UMRF graph notation, including sequences (a), concurrency (b), cycles (c), hierarchical graphs (d), multi-agent graphs (e), conditionals, and error management (f). Example UMRF graph with aforementioned properties combined, where "S" denotes graph entry and "E" denotes graph exit (g)". Reprinted from [9].

While most HRI systems must have hard-coded input modalities for tasks, TeMoto abstracts this away by using domain-specific language (DSL) to extract task implementation from input sources [9], as shown in Figure 2. The execution of UMRF-defined tasks is managed by the TeMoto Action Engine, a C++ library that interprets the UMRF semantics and controls the robot's actions [11]. Actions are implemented as plugins, dynamically loaded to the Action

Engine during run-time, which can be invoked, modified, and stopped during the execution [12], providing compelling platform for developers to build robots.

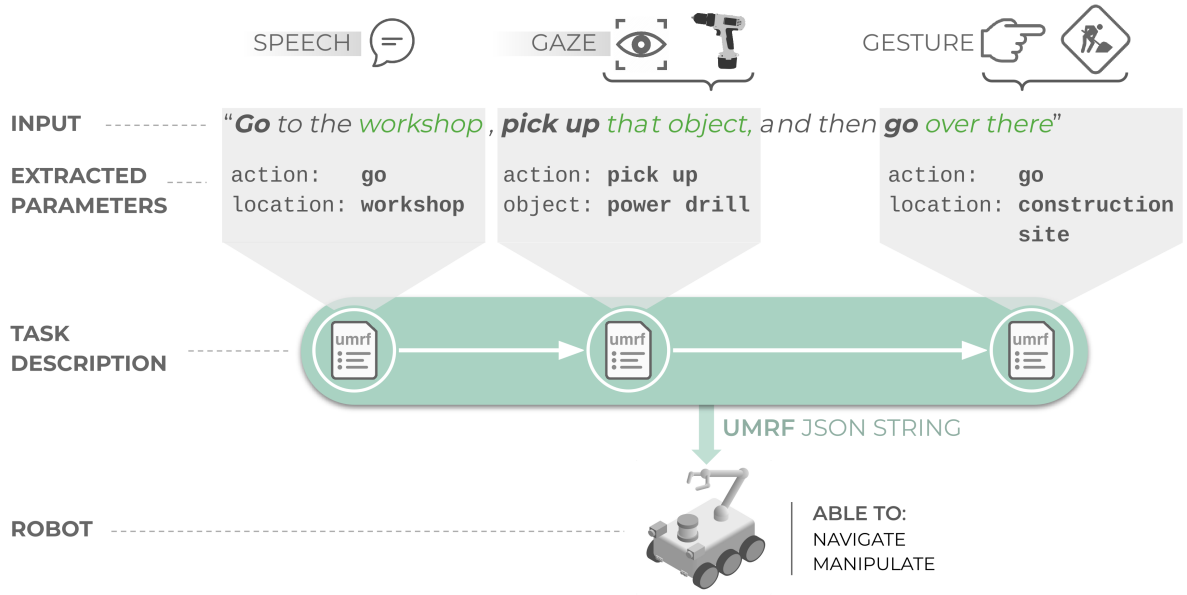

Figure 2. "Potential use-case of a multi-modal HRI system that benefits from having an intermediate task description format for fusing the modalities and commanding the robot.|" Reprinted from [9].

Collectively, the features of TeMoto form a complete middlewear platform to build multi-robot applications. The framework abstracts much of the boilerplate associated with the ROS 2 distributed systems architecture, allowing developers to focus on domain logic and workflows rather than infrastructure concerns.

2.3. React

React is a frontend framework developed by Meta [13] that helps to streamline frontend modern user interface development. In 2025, it was the 2nd most used web framework by professional developers with 46.9% market share [14], and 5.3% of all websites in the world use it [15], making it one of the most popular web technologies. React provides efficient performance with its virtual Domain Object Model (DOM) [16], which speeds up user interface rendering for the website. Furthermore, it scales well with large-scale applications [16], making it a popular choice in the enterprises.

It breaks the user interface into components [13], which can be atomic pieces like a button, form, or a sidebar. Notably, React provides a markup syntax language called JavaScript syntax

extension (JSX), enabling developers to write HTML-style language in JavaScript [13]. This makes it easy to create and combine multiple components through code-splitting, and enhance maintainability [13].

Significantly, it allows building both web apps and mobile apps using the same standard, making the development of cross-platform apps quicker [13]. It provides hooks like **useEffect** and **useState**, which facilitate the rendering of components and sharing app data [13]. Using built-in context management tools significantly speeds up prototyping as developers don't have to spend time on implement data infrastructure themselves. Also, it provides properties or **Props**, that is, the data in the form of parameters that is passed to child components of the app [13].

Overall, React's strengths in virtual DOM efficiency, modularity of its components, and extensions such as JSX, hooks, and props enable developers to build responsive, maintainable applications that span web and mobile environments with minimal overhead. Thus, React's modern UI has become integral to many full-stack ecosystems, both in enterprises and among non-professional programmers.

2.4. Task Management Frameworks and GUIs

BehaviorTree.CPP is a C++ framework that implements behavior trees, offering a modular, extensible approach to task management [17]. It supports dynamic task loading through plugins and describes behaviors via XML, which helps decouple task definitions from the library's codebase [17]. While tasks can be added during runtime, users must manually register new behaviors, limiting flexibility [9]. Additionally, the framework assumes single-robot execution and lacks mechanisms for distributed multi-robot tasks [9]. The rigidity in hierarchical representation - requiring explicit subtree definitions - can complicate automatic task generation across diverse robotic applications.

It provides a GUI editor IDE, **Groot2** (Figure 3), that simplifies editing the XML files [17], that is a C++ application using Qt framework. and visualize the state during the program execution[17]. It provides BT Editor with a simple drag and drop interface[17]. On the right, there's a visualization panel for trees that takes up 2/3 of the space as showcased in Figure 3. On the left side, there's a list of actions and trees. Users can drag and drop actions into the trees[17], and select which tree is active. Furthermore, the interfaces provides feature to view the logs and even replay its execution [17].

Finally, it restricts access to PRO features with a paywall, like adding interactive breakpoints, fault injection, and search functionality in a large tree [17], which can be seen as a significant drawback for developers and open-source community. While it is a popular GUI, the availability of installers exclusively for Windows and Linux [17] restricts its potential application in the, where robotics engineers may prefer using tablets or mobile devices for on-the-go control and navigation.

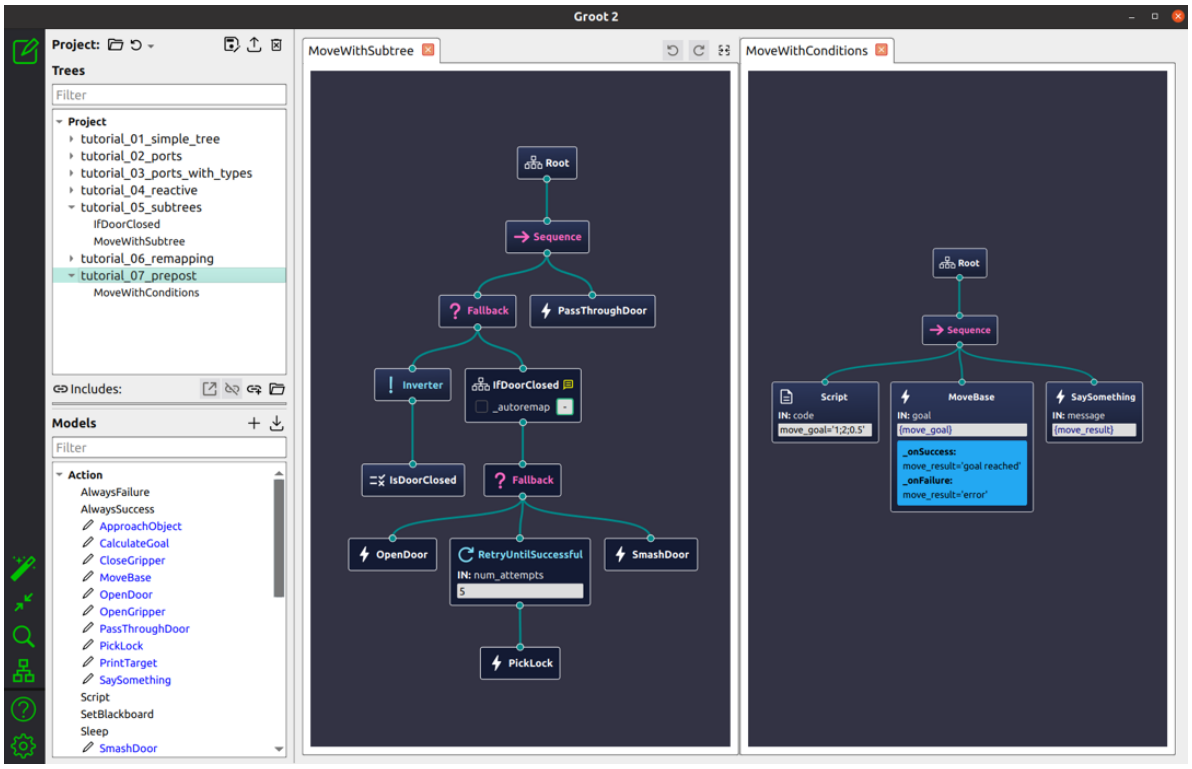

Figure 3. The GUI of Groot2 [18].

SkiROS 2 combines behavior trees with knowledge reasoning in a Python-based framework, enabling robots to select appropriate skills based on world conditions [19]. It is an open-source code, which makes it widely available for developers to modify and extend.

The **GUI of SkiROS2** (Figure 4) is written in ROS' rqt [19], a Qt framework for developing ROS applications [20]. It's platform compatibility is constrained to Ubuntu and Python [21], limiting its use-case to desktop developers. It lets users get an overview of loaded skills and their parameters [19], enhancing interpetability and debugging. The parameters of the skills are modifiable, and they can be started and stopped [19]. Furthermore, the UI allows users to see the contents of the world model (WM) and introduce new relations and integrations using dialogues [19]. Finally, an integration provides for modifying the pose of a WM elements [19].

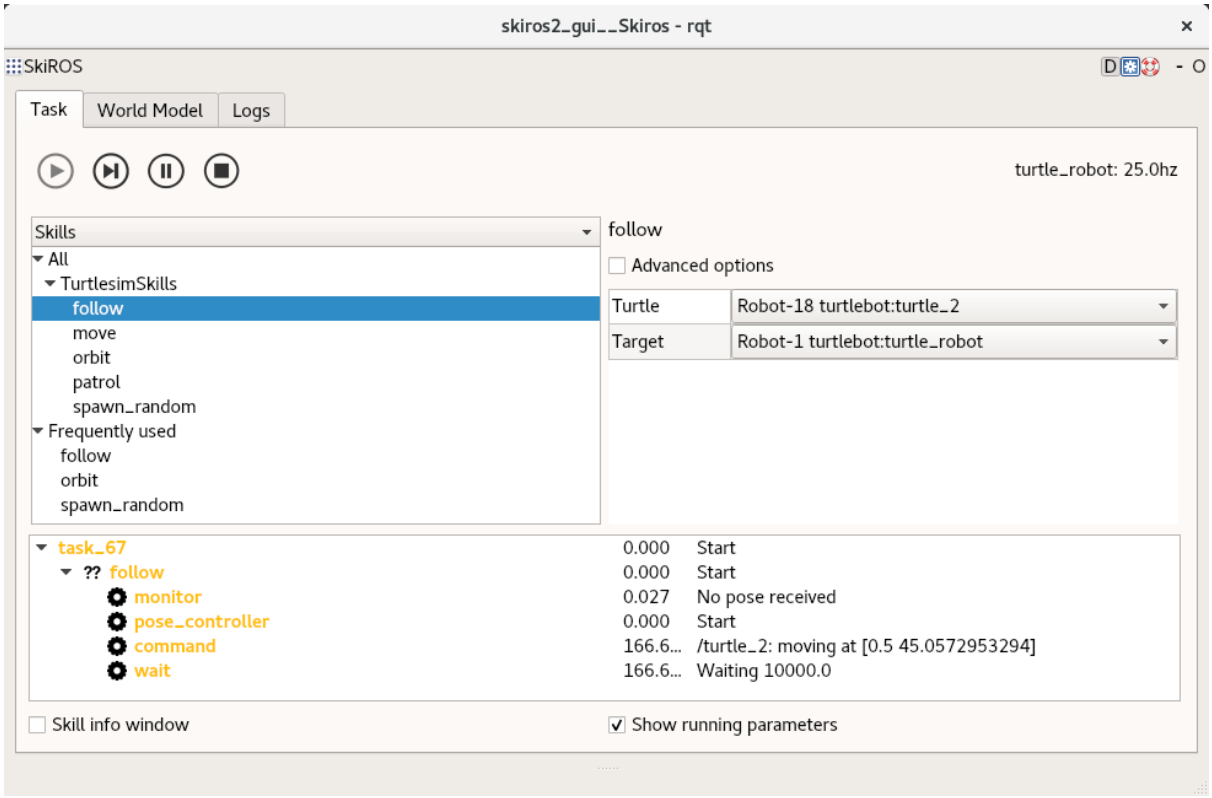

Figure 4. The GUI of SKIROS 2 [21].

RAFCON (Figure 5) offers a visual programming tool for creating hierarchical state machines [22]. It's written in Python and GUI is built with GTK+ widget toolkit [22], that is an API for cross-platform app development [23]. In principle, users can run it on both desktop and mobile devices.

It provides an integrated development environment (IDE) for creating hierarchical state machines [16]. The interface consists of the central canvas and three sidebars, as shown in Figure 5.The central canvas takes up majority of the space compared to other components. It allows visualizing complex state machines with highly nested hierarchies [22]. Additionally, it supports a zooming feature to view different hierarchy levels in varying degrees of detail [22].

The right sidebar is the state editor, which allows for modifying state properties, e.g., name and description [22]. There's also a functionality to directly change the values of the state machines data ports [22], which is great for experimenting with different actions. The left sidebar includes state machine-related widgets, like library manager, state machine tree, and execution history [22], allowing to debug the program. At the bottom sidebar allows inspection of program log outputs with a filtering feature [22]. This makes finding execution traces very quick. Finally, there's a source code editor that helps to change the code with syntax highlighting [22].

In summary, the GUI supports intuitive operations like drag and drop, copying and pasting states, and reorganizing connections [24]. The advantage of different view modes is that they help the user focus on the information they are interested in [24].

Figure 5. The GUI of RAFCON [25].

SMACH Viewer (Figure 6) is a hierarchical state machines in Python [26], similarly to RAFCON. It's implemented in Python and uses Qt framework for GUI [26]. Although, no mentions exist about platform compatibility, it's likely only supported on Linux. It enables visualization of transitions between states and data that is being shared between the states [26]. The GUI consists of two components: an editor and a left side panel.

The editor takes up the majority of the space. It represents a node as a state and an arc as a state transition [26]. If the outcome of the state and state machine match, then there is no arc [26]. Moreover, the user can view all state transitions when the user clicks the button "show implicit" [26]. Finally, there's a feature that enables users to view sub-state machines upon clicking on the node box [26], as outlined in Figure 7. The left side panel shows the current values of userdata [27]. It includes fields for message header and pose information, e.g., position and orientation coordinates. As a notable feature, there's a tree view that displays hierarchical relationships between state machines to indicate which ones are sub-state machines of others [26].

Notably, it supports task reuse and extension but does not provide reactive states or runtime task modifications. In SMACH, the user can start tasks during runtime, but cannot modify them

while running [9]. Additionally, it lacks support for shared and decentralized multi-robot tasks [9], limiting it's usability to single robot applications. Unfortunately, that project is also not maintained anymore [26].

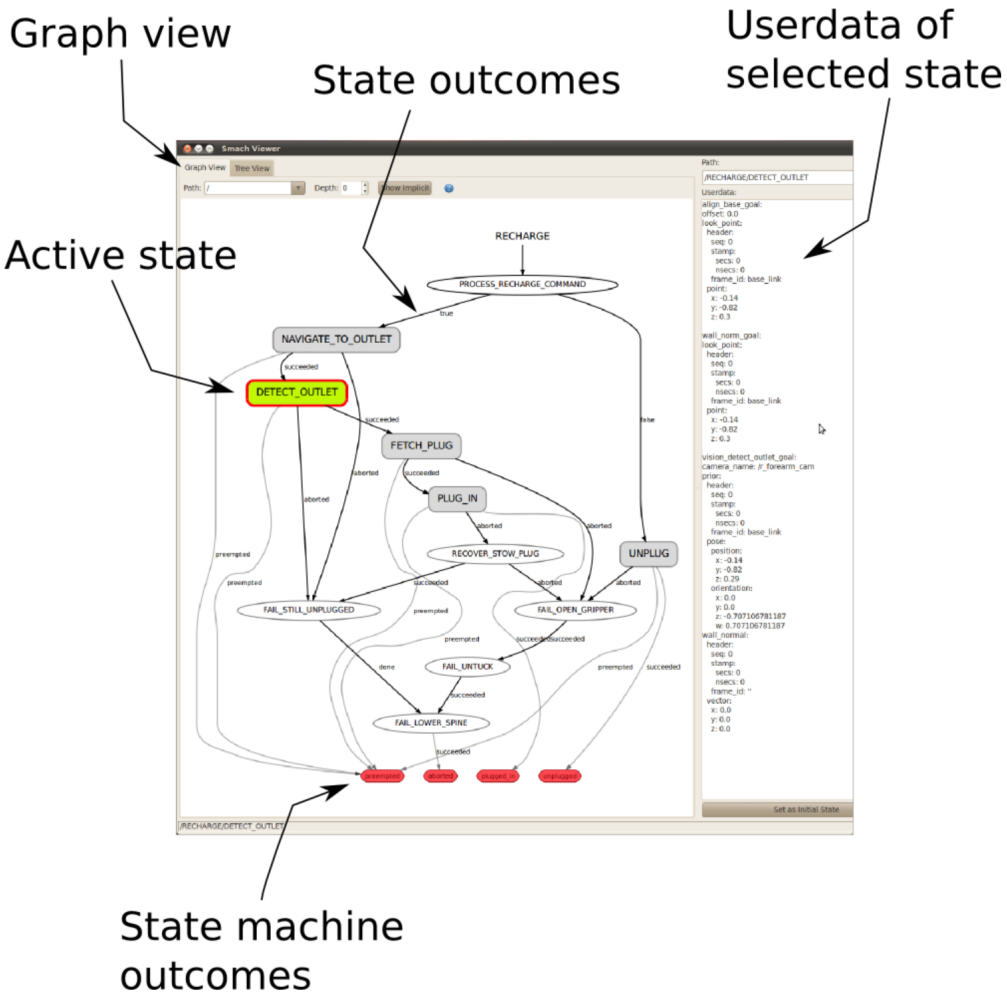

Figure 6. The GUI of Smach Viewer [26].

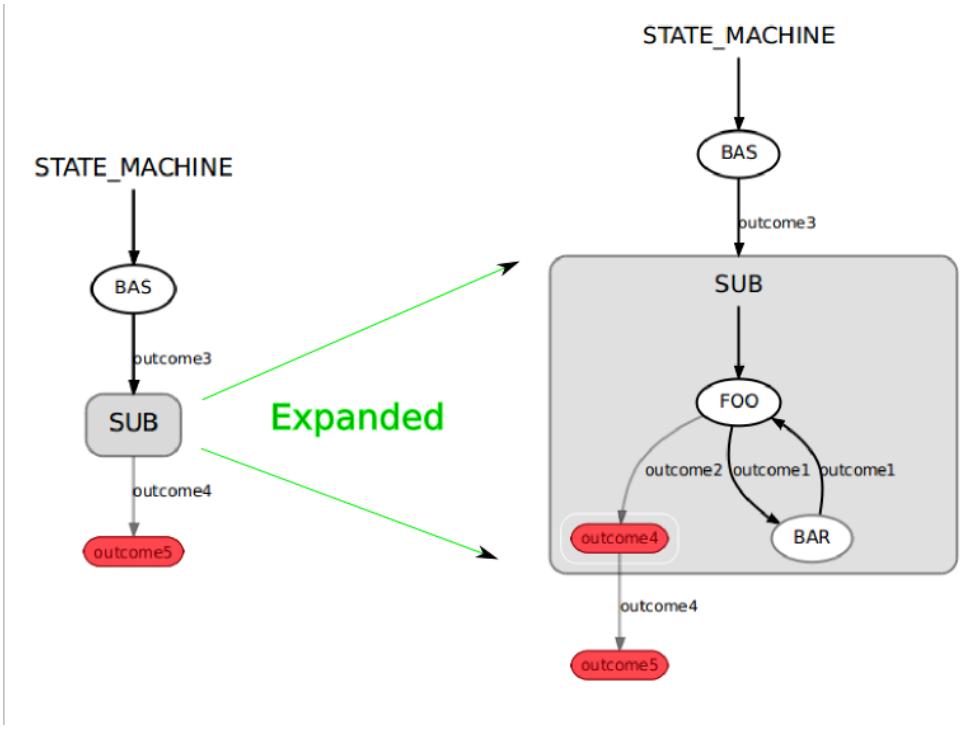

Figure 7. The depth feature in Smach Viewer GUI [26].

SMACC is a C++ state machine package that uses Boost Statechart Library and the actionlib interface [28]. It has drawn inspiration from previous work from SMACH [29], the major difference being that this is written in C++ rather than Python. Similarly, there's little mention about platform compatibility, suggesting a high likelihood of being supported only on Linux. A key differentiation of SMACC is its use of orthogonals as containerized components that encapsulate clients and their behaviors [29].

The package includes a GUI called **SMACC Viewer** [28] as shown in Figure 7. It consists of a display panel occupying more than half of the interface on the left and a transition log window on the right. The display panel visualizes the current state machine, representing states as nodes and transitions as arcs.

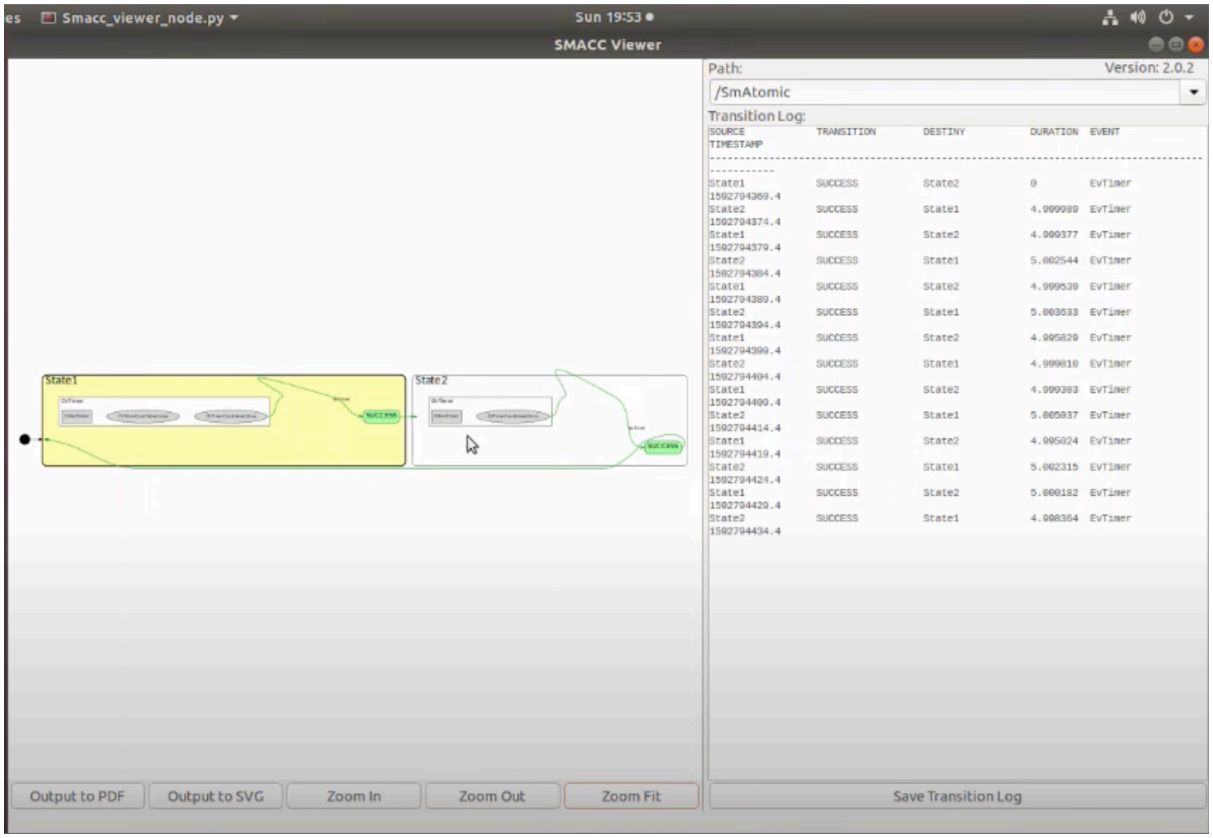

Figure 8. Screenshot of the SMACC Viewer GUI [30].

On the right, there's a transition log showing state transition changes with their respective timestamps. Users can zoom in and out of the state machine using the buttons on the display panel. However, direct editing of state machines in the GUI is not supported. Additionally, the GUI provides export functionality to save the state machine as a PDF or SVG and export the state transition log. Notably, SMACC Viewer is closed source [28], restricting developers from modifying or extending its functionality.

2.5. Current Method of Managing UMRF Graphs in TeMoto

At the current stage, developers using the TeMoto framework have two options to create graphs. The first option (1) requires manually editing the UMRF JSON files, which is time-consuming and prone to human error. As outlined in Section 2.2., TeMoto essentially works on sharing data between different UMRF graphs that occupy a known data structure. Modifying those properties of tasks and actions is the main way to introduce changes in how robot systems behave.

The second option (2) is to use TeMoto Action Designer, written in C++ developed by Valner et al. [9] Essentially, it’s a tool that abstracts modifyin the same JSON files with a GUI. However, this only supports ROS 1 and lacks support for the conditions and mechanisms of different middlewear layers [9]. This means for ROS 2 applications developers are only limited to option 1.

The absence of a ROS 2-compatible GUI leaves a considerable gap in TeMoto's HRI experience. Closing this gap could simplify the process of managing UMRF graphs and represent a critical step toward enabling richer, more user-friendly integration of TeMoto in advanced robotic workflows.

3. Methodology

The goal is to design an HRI GUI, TeMoto Action Assistant, a user-friendly design tool that simplifies the creation of UMRF graphs inside the TeMoto Software framework. This chapter defines the technical requirements and offers design and implementation decisions to meet them. The methodology is structured as follows: Section 3.1 sets the baseline requirements for the GUI, taking inspiration from previous GUIs. Section 3.2 defines concrete technical requirements to achieve features that have an advantage over the existing solutions. Section 3.3 sets the stage for what the GUI layout should look like and its rationale. Section 3.4 describes the frontend components. Section 3.5 describes the backend and its integration with the TeMoto Action Engine and ROS 2. Section 3.6 describes the design philosophy and relationship between nodes and edges in the context of UMRF graphs.

3.1. Baseline Requirements

As outlined in Chapter 3, the existing GUIs have advantages and drawbacks that could be used to draw inspiration from and improved upon in the context of this work. Table 1 displays the most important baseline requirements compared with previous work.

First, all existing GUIs are desktop-based applications, written in Python or C++, catering towards stationary users. However, there are types of roboticists, e.g., field roboticists, who work in diverse environments and prefer to use native devices like tablets or mobiles. Thus, being constrained to desktop devices is an explicit limitation for them. To overcome this limitation, TeMoto Action Assistant should use a GUI framework that is **platform-agnostic** and compatible with both desktop and native devices. Current web technologies, such as HTML5, enable software to be used on any device with a web browser [31]. Hence, it is the best paradigm to address multi-platform needs. According to W3Techs Web Technologies Survey, 98.9% of websites use JavaScript as their client-side programming language [32], so JavaScript is a well-reasoned reasoned implementation language for the web app.

	Implemented in	GUI Framework	Availability	Runtime execution and monitoring
Groot 2	C++	Qt	Proprietary	✓
SkiROS2	Python	Qt	Publicly available	✓
RAFCON	Python	GTK+	Publicly available	✓
SMACH Viewer	Python	Qt	Limited	×
SMACC2 Viewer	C++	N/A	Proprietary	×
TaskForce	Python	N/A	Limited	✓
TeMoto Action Assistant	Javascript	React	Publicly available	✓

Table 1. Comparison of different GUI and their features.

Second, the GUI framework should be advanced enough to support sophisticated Document Object Model (DOM) state updates. This can contribute greatly to user-friendly design, as otherwise, slow adaptability to display the correct information to the developer can pose safety risks, as developers may be too late to react to start or stop specific actions of the robot. e.g., a robot receiving the stop action 1 second late before hitting a wall. As outlined in Section 2.3, React allows to build of cross-platform apps with dynamic components, enhancing modern user interface development. Additionally, the communication between the front-end and back-end should be **low-latency** and have **real-time synchronization** with the TeMoto Action Engine. For this reason, React is chosen as the GUI library for TeMoto Action Assistant, to build a robot developer-friendly design tool that can satisfy both computer and tablet for users building robotics applications. Third, the communication between the front-end and back-end should be low-latency and have real-time synchronization with the TeMoto Action Engine. Websockets can achieve 2.3 to 4.5 times lower latency over HTTP-based polling methods [33], thus being faster method of data transfer that fits well into the requirements of this work.

Fourth, runtime execution and monitoring are an important part of an HRI application. The interpretability about what what is going on inside the system, what actions are actively processed between different robots, and their results is crucial to understanding cause- and-

effect in developing robot applications. Moreover, the ability to start **task execution** and conduct **monitoring** using UI rather than on a command line can be significantly more user-friendly. These features are present in Groot 2, SkiROS2 and RAFCON, which necessitates TeMoto Action Assistant to have it aswell.

Fifth, GUIs like Groot 2, SMACH Viewer, and SMACC2 Viewer are not open-source. This means that the code of the application is not widely available to view, modify, and extend. This poses many limitations and does not let the community contribute to it, potentially limiting its potential and adoption. For this simple reason, TeMoto Action Assistant will be **open-source**.

Satisfying all those features would enable TeMoto Action Assistant to be feature complete and have an advantage over all the existing GUIs at least in one category. This provides baseline requirements, that could be an essential advantage to inviting more developers to use TeMoto framework to build robot applications.

3.2. Technical Requirements for TeMoto Action Assistant

Besides the baseline requirements, we must define concrete technical requirements for the result. As outlined in Section 3.1, on a base level, the user must be able to use the GUI to execute, create, modify, and execute tasks and receive feedback about them. This means the web app should be on the backend interface with the already developed TeMoto Action engine to facilitate such capabilities. Figure 9 provides a high-level picture of how the frontend should interact with the action engine. Its main task is to provide feedback on graph modification and send requests to start or stop graph execution. Then, Temoto Action engine handles the underlying ROS 2 with proper communication protocols and captures feedback from the real robot.

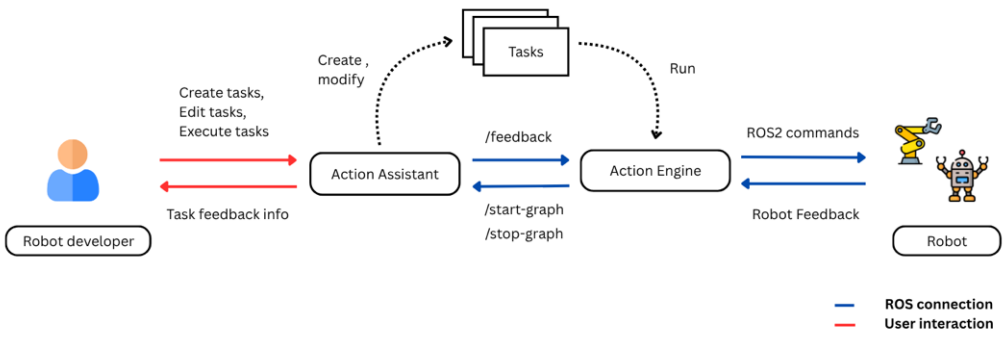

Figure 9. A high-level picture of the role of TeMoto Action Assistan and how it interacts with the rest of the framework

The following **technical requirements (Rs)** have been identified to achieve the desired GUI. **R1-R4** are related to the baseline requirements outlined in the Section 4.1. and **R4-R16** are related to runtime monitoring and execution requirements.

From architectural point-of-view, the application GUI must be:

1) **Implemented using web architecture, e.g. programming language like Javascript.**
2) **Using a web GUI framework with fast component rendering e.g. React.js.**
3) **Released as an open-source codebase repository.**
4) **Equipped with backend with bidirectional communication protocol that interfaces with TeMoto Action Engine e.g. Python-based server with WebSocket protocol.**

The a graphical user interface (GUI) must enable users to:

5) **View and select existing tasks from a list.**
6) **View and select existing TeMoto actions from a list.**
7) **Visualize the task as a relationship of nodes and edges**
8) **Display task and action details on an info panel.**
9) **Differentiate between actions assigned to different robots.**
10) **Create and edit TeMoto actions:**
    a) Rename actions
    b) Set and edit values of the input parameters of actions:
        i) Name
        ii) Data type
11) **Generate new TeMoto action packages.**
12) **Create and edit tasks (UMRF graphs).**
    a) Rename
    b) Add actions
    c) Remove actions
    d) Connect actions together on predefined conditions.
13) **Execute a task.**
14) **Visualize dynamic feedback about task progress during execution.**
15) **View and create hierarchical Tasks**

This set of features allows to deliver the proof-of-concept functionality of the Action Assistant to give developers the core features to utilize the benefits of the TeMoto Software framework. In a way, these set of requirements define the functional features that the app should have and fulfilling these is a prerequisite to delivering a functional and state-of-the art GUI.

3.3. The Layout of the GUI

As a first step, it needs to be clear what kind of user experience the app needs, all while fulfilling our existing requirements from Section 3.2. There should be a clear expectation of what panels the app needs, how large they should be, and where they will be located.

First, the diagram is an important component of the GUI because it visualizes the abstraction of how the system behaves. As shown in Section 2.4, GUIs like Groot2, RAFCON, SMACH Viewer, and SMACC Viewer include a central canvas that takes up the majority of the space. The idea of the central canvas is to visualize the states and their transitions, so it's reasonable that this takes up most of the space. Thus, we have to include a central canvas that would help to visualize actions and transitions that are part of UMRF graphs.

Second, the user must be able to manage many UMRF graphs at once. So there needs to be a way to pick and choose which graph is displayed on the visualizer at any given moment. Taking inspiration from Groot2, users can select active trees and actions from the left side panel. This is sensible because these should not be in the main field of view and should be classified as more of a menu functionality. Thus, TeMoto Action Assistant should have a left panel with a list of graphs and actions.

Third, creating and modifying tasks is essential to a robot developer's job. A solution proposed in Groot 2 and RAFCON is to allow users to add new actions to tasks by simply dragging and dropping them to the visualizer. It's a simple and intuitive solution. Thus, the same experience should be incorporated into TeMoto's GUI.

Fourth, since modifications in the previous paragraph can significantly alter the underlying UMRF graph, developers should be able to observe what is going on in the real JSON file. Viewing parametrized values is important, as this helps debug the robot itself. At the same time, displaying all the information from the UMRF graph to the central visualizer component could be overwhelming. Thus, a separate panel is more reasonable as RAFCON has implemented a side panel for modifying state properties and data ports. For this reason, it will serve a function to display UMRF data and modification functionality.

Fifth, there should be a button to execute a task. From existing GUIs, RAFCON displays a "play" sign at the bottom of the visualizer, while SkiROS 2 has it on the top left of its skills list. Since the plan for TeMoto Action Assistant is to place the list of tasks on the left side, it would make sense that the "play" button is also placed near the active task. Therefore, the button for executing the graphs should be placed near the tasks themselves on the left side of the panel.

Finally, operating with hierarchical graphs calls for a more sophisticated visualizer component than just a one-level view of the graph. As depicted in Figure 7, Smach Viewer provides a good example of a GUI, where a user can simply expand a sub-state machine by a simple click. Having such a feature in TeMoto would allow working with multi-level graphs. On this basis, one-click expansion of a sub-graph should be implemented inside the central visualizer.

In conclusion, this high-level layout provides the basis for a state-of-the-art user experience from existing GUIs, extracting great ideas from authors of existing works. At the same time, it allows for flexibility to implement our own outlined requirements.

3.4. The Design of a Node and Edges

As high-level purpose of the UI is to help the developers visualize multi-robot systems and coordinate actions between them, the frontend should be implemented as a flow diagram, where the objects are abstracted into nodes and edges. For the TeMoto Software framework this means the following.

First, users can create new **actions** for a robot, e.g., move a robot from point A to point B. Second, they can use the drag-and-drop method to assemble **graphs** with actions chained together with conditions, e.g., make a sensor robot (robot A) start inspection when mobile robot (robot B) arrives on the site. Finally, a GUI should allow visualization of both tasks and its parameters.

As described in Section 2.2, several properties of the UMRF graph facilitate the creation of multi-robot interactions. Since the graph consists of many actions run sequentially or in parallel, it makes sense to represent each action in a graph as a node, as previous works outlined in Section 2.3. As a logical continuation, the nodes should be connected by edges in a directed manner. This is easy for child node relationships, as the child action should always be activated when the parent node completes its task. For this reason, we're choosing to implement one child edge source for a node, which sits on top of the node like in Figure 10.

Figure 10. A node with four edge source types.

For the parent node, the case is a bit trickier, as there one of four cases can happen before the execution of a child node: are four cases for the child action:

1. **action is completed successfully – action returning True or False.**
2. **error – action fails.**
3. **stop – actions stops entirely e.g., when spotting an unseen object.**

This means means there are multiple end states to the parent node that should be displayed as conditions edges from parent to child. To make it clear for the user, which end state it's connecting nodes on, four possible edge sources for the parent were implemented as shown in Figure 10.

4. Results

This chapter presents the development outcomes of the proposed TeMoto Action Assistant and evaluation against the technical requirements outlined in Chapter 3. This chapter is organized into two parts. First, frontend and backend implementations are described in Sections 4.1 and 4.2. Then, the technical requirements (**R1–R16**) fulfillment is assessed in Sections 4.3-4.7.

4.1. Frontend

The architecture of the frontend is displayed on Figure 10. The entry point of the web app is **ActionInterface** , which, upon mount, establishes a socket connection with the Python Flask API, receives UMRF graphs, and saves them to the program state. The graphs are passed to the **GraphListPanel**, which is a sidebar component where the user can select the graph of her choice. Similarly, the API provides modularized actions, which are displayed on the **ActionListPanel**. Additionally, users can create new graphs or actions by clicking on the **"+"** button.

On the right side of the app, there is a **GraphInfoPanel** that displays the JSON info with basic syntax highlighting for the developer. It works on a conditional basis, showing the selected element of either the graph or the action. It also provides basic editing features, like renaming the graphs and defining conditions. Upon changing the properties, the app will save them to the API.

In the center of the interface is **NodeEditorPanel**, which is responsible for displaying the UMRF graphs using an interactive diagram. Upon selecting the graph from the **GraphListPanel** sidebar, the graph is passed to the **NodeEditorPanel** component, which handles displaying the graph using React Flow. **React Flow** is an open-source library that provides out-of-the-box elements like nodes, edges, and interactive callback events [34] making it easy to create flow-based diagrams.

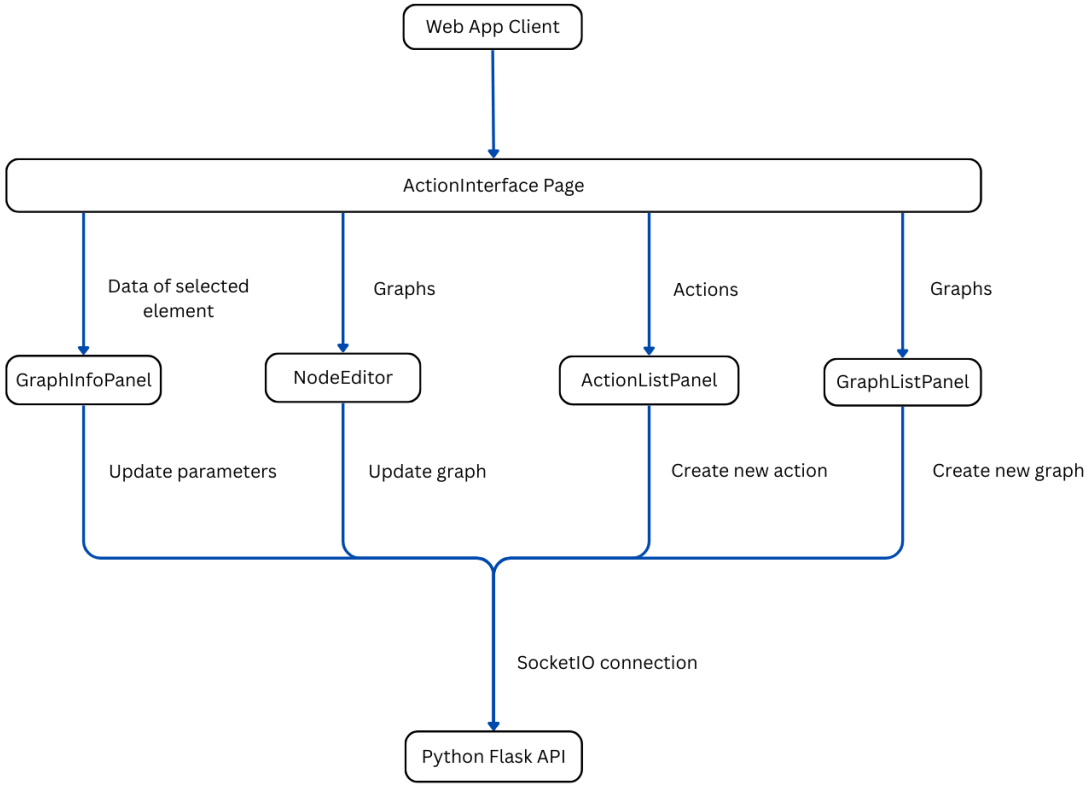

Figure 10. The architecture of frontend.

In **NodeEditorPanel**, a graph is passed to a mapping function **jsonToFlow**, which converts the UMRF JSON structure to React Flow data structure, where each:

- **TeMoto action** represents a node,
- **The parent-child relationship** between TeMoto actions represents an edge.
- **Run-time condition** "run" sets the edge source for one of the four condition types: "on_true", "on_false", "on_error", or "on_stopped".

After this, the user is displayed an interactive diagram as displayed on Figure 11.

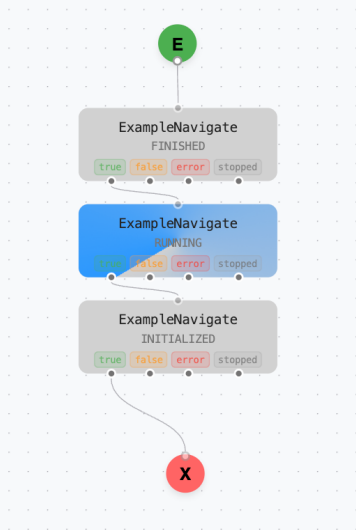

Figure 11. Screenshot of a sample UMRF graph constructed using the tool, including actions and real-time status updates during execution.

In contrast, the mirror mapping function **flowToJson** converts the Reactflow data structure back to the UMRF JSON format. This guarantees that if users make changes to the graph, those changes will propagate to the existing graph's state and Flask API afterwards. The mapping in the UMRF graph follows similarly:

- **Node** converts to a TeMoto action object.
- **Edge** adds a parent-child relation for the two TeMoto actions.
- **Edge source** sets the condition value to **'true'** for one of the four condition keys: "on_true", "on_false", "on_error", or "on_stopped".

4.2. Backend

The goal of the backend is to handle communication between the web app and the TeMoto Action engine. Since user interaction with the web app diagram requirees fast component updates, we need something reactive UI that handles high-frequency data transmission to synchronize the UMRF graph with the backend. To achieve this goal, we implemented the backend in the following way.

The backend is written in **Python** using **Flask**, a lightweight Web Server Gateway Interface (WSGI) built for web application development [35]. It serves as an interface between the web app and TeMoto Action Engine. We use the **Flask-SocketIO** library to create a connection between a client and a server [35], which allows sending messages without having to wait for the server for a reply [36]. This suits real-time data transferring tasks well.

The server can be started in two modes. The first is **run-time mode**, which queries data from the TeMoto action engine. The second is development (or non-run-time mode), which saves data in memory and does not require the TeMoto Action Engine instance. The main entry point of the backend is ***app.py***, where the WebSocket connections are made, and API endpoints are set up using the **setup_action_socket()** method via ***action_socket.py***

In the ***action_socket.py***, we have API methods like:

- **GET get_graph** - retrieves the graph from memory and returns the graph based on the given key.
- **PUT set_graph** - updates the graph in memory based on the given key.
- **PUT exec_graph** - executes the graph using ROS 2 interface via TeMoto Action. Alternatively, it stops the execution of a graph if it is already running.
- **POST create_graph** - creates a new graph with the provided data and saves it to the disk. Finally, it notifies the client about the creation of a new graph.

The task management workflow, including graph selection, creation, and editing operations, is illustrated in Figure 12.

- **POST create_action** - creates a new action with the provided data, and saves it to the disk. In the end, it notifies the client about the changes.
- **POST generate_action** - creates a new package of TeMoto action using the ROS 2 interface, and notifies the clients about the updates.

Figure 13 shows the complete action management sequence, from action creation through ROS 2 package generation.

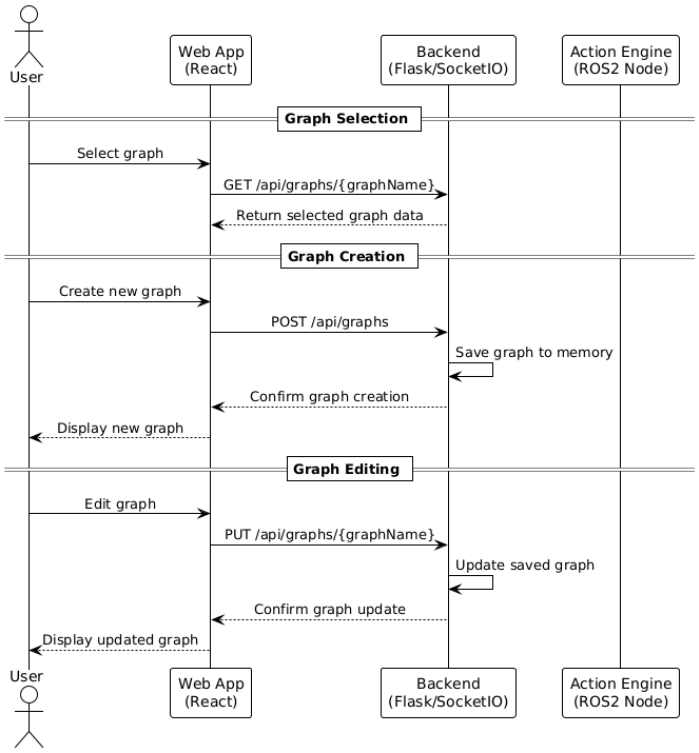

Figure 12. Task management sequence diagram

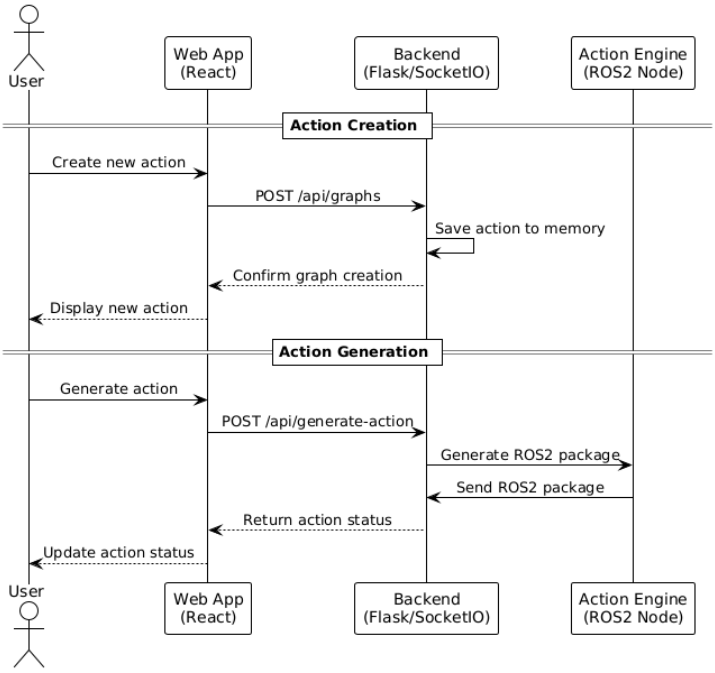

Figure 13. Action management sequence diagram.

Furthermore, an **ActionNode** class was implemented in ***action_ros.py***, which serves as a ROS 2 node to manage communication between other nodes for executing actions. The following ROS 2 publishers and a subscriber are initialized:

- **PUBLISHER** to topic /**umrf_graph_start** - execute an action in a graph.
- **PUBLISHER** to topic /**umrf_graph_stop** - terminate an action in a graph.
- **SUBSCRIBER** to topic /**umrf_graph_feedback** - receives real-time feedback on the running graph.

The real-time execution workflow, including start/stop operations and feedback broadcasting, is detailed in Figure 14.

Moreover, there is also the following client:

- **CLIENT umrf_graph_get** - requests and retrieves information about all available UMRF graphs in the ROS 2 ecosystem.

Finally, there is a utility functions that help to manage ROS 2 node lifecycle:

- **run_ros_action_thread** - creates a new thread to run the **ActionNode** ROS 2 node, without blocking the primary Flask web server.

All those functionalities make it possible to orchestrate the interaction between the web app and TeMoto Action Engine.

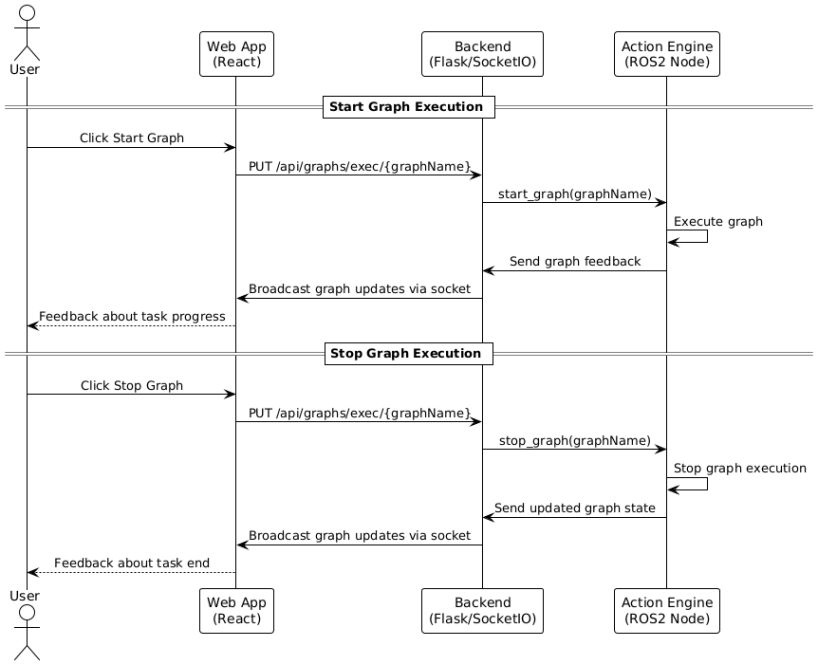

Figure 14. Real-time execution sequence diagram.

4.3. Programming Language, Architecture, and Availability (R1-R4)

The codebase of the GUI was released as open-source repository on Github [37] for anyone to inspect and modify, satisfying **R3**. The frontend application was implemented in Javascript and React, prioritizing web-based architecture and reactive components, fulfilling **R1** and **R2**. The backend server was written in Python using websocket based Flask API, addressing **R4** and the need for bi-directional communication.

4.4. Tasks and Temoto Actions (R5-R9)

Tasks and TeMoto Actions are loaded using bi-directional websockets from the backend. Using the left sidebar, the user can simply activate one task and action at a time, as shown in Figure 15, satisfying **R5** and **R6**. As shown in Figure 17, an active task will be displayed in the center of the UI's flowchart as nodes and edges using React Flow. This fulfills **R7**. Moreover, there's additional feature that displays specific graphs that have labels for different and labels actions actions assigned these robots, fulfilling **R9**. Finally, the properties of an active element will be displayed as a syntax-highlighted JSON on the right panel of the UI, depicted in Figure 16. This fills **R8**.

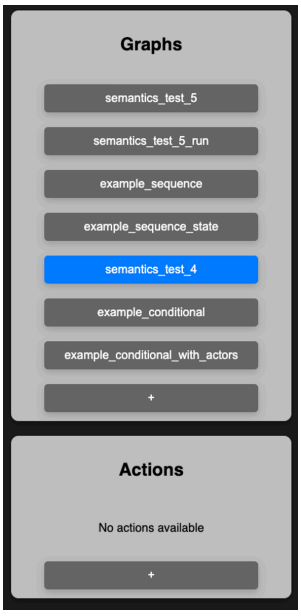

Figure 15. Screenshot of the left side panel of the GUI, consisting of list of tasks and TeMoto Actions.

Figure 16. Screenshot of the right side of the panel of GUI, showcasing all poperties of a task.

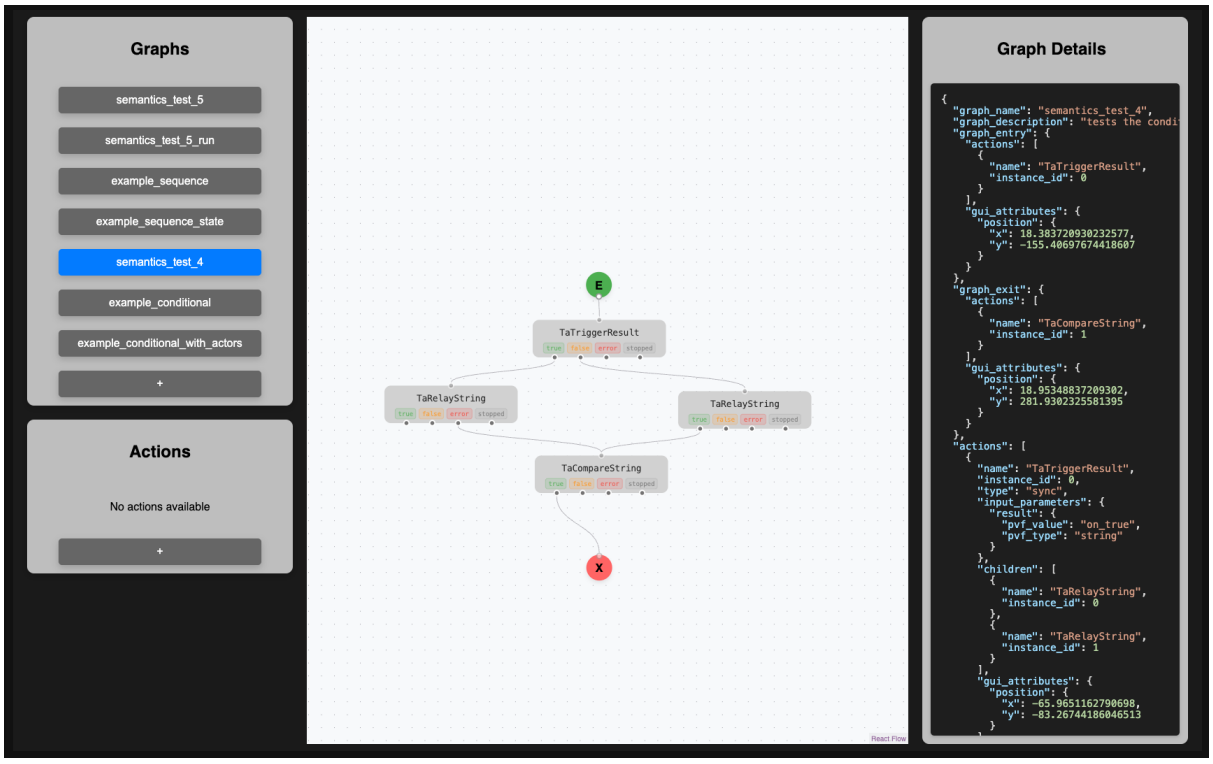

Figure 17. The user interface of TeMoto Action Assistant.

4.5. Creation of Tasks and Temoto Actions (R10-R12)

As shown in Figure 17, there is a **"+"** button for both tasks and TeMoto Actions. By pressing on it, users can create entirely new data objects. In the case of tasks, it will generate an empty graph, where users can drag and drop TeMoto Actions into and assemble logic with edge connections. Under the hood, this follows rules for constructing UMRF graphs, satisfying **R12**.

For TeMoto Actions, a similar process exists, except upon clicking **the "+"** button, the user is presented with a panel on the right to insert predefined action parameters and the name. Finally, after pressing "Generate Action," the backend generates a formal TeMoto Action package which can be used inside UMRF graphs. These fulfill both **R10-R11.**

4.6. Task Execution and Monitoring (R13-R14)

Finally, users can execute the tasks on a real robot, assuming it's running on the TeMoto Software framework. Developers can easily start a task by clicking the **"play"** button on the right side of the panel, next to each task. The button is displayed in a production environment GUI, but is currently absent from developer enviroment version. Upon click, the web app sends requests to the TeMoto Action engine to execute the Task on the robot. The TeMoto Action engine keeps Python's server up-to-date on the state of the UMRF graph, and these changes are subsequently updated through websockets to the web App. At the same time, the web app displays the UI changes using visual cues of task execution, satisfying **R13**.

There are two main visual cues:

1. Circling blue line with the state "RUNNING" on it - when the robot action is running.
2. Default - when the robot action is not running.

Concurrently, the user can also see the state updates on the right side of the graph info panel, giving them various ways to monitor the state of the execution. This execution and monitoring capability fills **R14**.

4.7. Hierarchical Tasks (R15)

Implementing a hierarchical task for the GUI was unfeasible during this thesis's timeframe. For this reason**, R15** is not satisfied. Unfortunately, working with graphs with multiple logic levels is not supported in the current web app version.

5. Future Work

This chapter suggests possible directions to extend and improve the TeMoto Actions Assistant. The chapter is divided into Section 5.1, which discusses hierarchical graph support, and Section 5.2, which proposes migration to a type-safe programming language.

5.1. Hierarchical UMRF Graph Support

As mentioned in Section 4.7, the requirement **R15** of hierarchical graphs was not fulfilled. The requirement is key because such a feature would allow developers to reuse existing graphs in other robot workflows. Thus, this presents a capstone feature for future work that has the potential to deliver the most utility.

5.2. Code Migration to TypeScript

The code should be migrated to a type-safe language for a production-ready web app. TypeScript is a strongly typed programming language and a superset of JavaScript [38]. It helps developers to catch runtime errors directly from the editor [38], reducing the time it takes to debug the code. While the current vanilla JavaScript implementation is convenient, its maintainability will decrease as the codebase grows. Adopting TypeScript would therefore enhance developer productivity and guarantee the application is built on a sustainable foundation in the long term.

5.3. More Complex Testing and User Feedback

One limitation of the study is that the GUI tool was only evaluated on a small group of TeMoto developers and not tested with external users. Collecting more feedback from other users would allow us to assess the quality of the features and introduce more design improvements. Additionally, more complex real-world (especially on-field tasks) were not tested on a real deployment of the robot, which might bring out edge cases that could not be addressed in the time frame of this thesis. Addressing these would allow us to discover needs that could be met..

6. Conclusion

The thesis aimed to develop a web-based GUI for the TeMoto Software framework that allows developers to create UMRF graph-based tasks easily. This effort promised to eliminate the need for developers to modify highly structured JSON-based graphs manually and speed up their workflow. The intended outcome is simply the process of task creation, execution, monitoring, and making TeMoto more accessible.

As outlined in Sections 4.3, 4.4, 4.5, 4.6, and 4.7, most of the defined technical requirements (**R1-R14**) are met except **R15**. Despite this, the primary objective was achieved: the TeMoto Action Assistant has features that allow developers to create, edit, execute, and monitor UMRF graphs through a GUI accessible both on desktop and mobile.

Compared to prior solutions, the system proposes a novel web-based architecture for HRI GUIs with modern UI responsiveness. Its open-source availability distinguishes it from several ROS 2 task management tools, taking steps towards code transparency and community-driven development.

The main contributions of this work include:

1. Designing and implementing the first web-based GUI for the TeMoto framework.
2. Introducing real-time task execution and monitoring features to the GUI that is compatible with the TeMoto runtime.
3. Providing a cross-platform experience across desktop and mobile for TeMoto Action Assistant, widening accessibility to all roboticists.

Despite some limitations like the absence of hierarchical task support, codebase type-safety concerns, and limited user testing, the work demonstrates a fresh perspective on modernizing task management tools for robots. The project demonstrates that modern web technologies can reasonably be applied to robotics software stacks without compromising flexibility.

Its open-source approach invites collaboration, allowing the robotics community to build upon and adapt the tool to their needs. As the field advances, this work positions TeMoto to play a more significant role in modern HRI software frameworks in domains where robots navigate in hazardous environments.

7. References

[1] K. A. Szczurek, R. M. Prades, E. Matheson, J. Rodriguez-Nogueira, and M. Di Castro, "Multimodal Multi-User Mixed Reality Human–Robot Interface for Remote Operations in Hazardous Environments," *IEEE Access, Access, IEEE*, vol. 11, pp. 17305–17333, 2023, doi: 10.1109/ACCESS.2023.3245833.

[2] Y. Gao and S. Chien, "Review on space robotics: Toward top-level science through space exploration," *Sci Robot*, vol. 2, no. 7, p. 28, Jun. 2017, doi: 10.1126/SCIROBOTICS.AAN5074/ASSET/9A37AD9A-6852-4720-A617-F118AABFD931/ASSETS/GRAPHIC/AAN5074-FX5.JPEG.

[3] E. J. Lopez Pulgarin *et al.*, "From traditional robotic deployments towards assisted robotic deployments in nuclear decommissioning," *Front Robot AI*, vol. 12, 2025, doi: 10.3389/FROBT.2025.1432845.

[4] M. D. Moniruzzaman, A. Rassau, D. Chai, and S. M. S. Islam, "Teleoperation methods and enhancement techniques for mobile robots: A comprehensive survey," *Rob Auton Syst*, vol. 150, Apr. 2022, doi: 10.1016/J.ROBOT.2021.103973.

[5] S. Bensalem, F. Ingrand, and J. Sifakis, "Autonomous Robot Software Design Challenge," Aug. 2008.

[6] X. Mao, H. Huang, and S. Wang, "Software engineering for autonomous robot: Challenges, progresses and opportunities," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2020-December, pp. 100–108, Dec. 2020, doi: 10.1109/APSEC51365.2020.00018.

[7] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, Architecture, and Uses In The Wild," 2022, doi: 10.1126/scirobotics.abm6074.

[8] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet another robot platform," *Int J Adv Robot Syst*, vol. 3, no. 1, pp. 043–048, 2006, doi: 10.5772/5761/ASSET/BC89CF1C-8951-4FD2-B611-D86A64EAE0DE/ASSETS/IMAGES/LARGE/10.5772_5761-FIG3.JPG.

[9] R. Valner, *Design of TeMoto, a software framework for dependable, adaptive, and collaborative autonomous robots*. 2024. Accessed: May 12, 2025. [Online]. Available: https://hdl.handle.net/10062/105994

[10] J. Zhang, X. Yu, S. Ha, J. P. Queralta, and T. Westerlund, “Comparison of Middlewares in Edge-to-Edge and Edge-to-Cloud Communication for Distributed ROS2 Systems,” *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 110, no. 4, Nov. 2024, doi: 10.1007/s10846-024-02187-z.

[11] R. Valner, S. Wanna, K. Kruusamäe, and M. Pryor, “Unified Meaning Representation Format (UMRF)-A Task Description and Execution Formalism for HRI,” *ACM Trans Hum Robot Interact*, vol. 11, no. 4, p. 38, Sep. 2022, doi: 10.1145/3522580/SUPPL_FILE/THRI_UMRF_SUPPLEMENTAL_MATERIAL.ZIP.

[12] R. Valner, V. Vunder, A. Aabloo, M. Pryor, and K. Kruusamae, “TeMoto: A Software Framework for Adaptive and Dependable Robotic Autonomy With Dynamic Resource Management,” *IEEE Access*, vol. 10, pp. 51889–51907, 2022, doi: 10.1109/ACCESS.2022.3173647.

[13] “React.” Accessed: May 13, 2025. [Online]. Available: https://react.dev/

[14] “Technology | 2025 Stack Overflow Developer Survey.” Accessed: Jul. 31, 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology#most-popular-technologies-webframe-prof

[15] “Usage Statistics and Market Share of React for Websites, May 2025.” Accessed: May 16, 2025. [Online]. Available: https://w3techs.com/technologies/details/js-react

[16] Mikita Piastou, “Comprehensive Performance and Scalability Assessment of Front-End Frameworks: React, Angular, and Vue.js,” *World Journal of Advanced Engineering Technology and Sciences*, vol. 9, no. 1, pp. 366–376, Jun. 2023, doi: 10.30574/WJAETS.2023.9.2.0153.

[17] “BehaviorTree/BehaviorTree.CPP: Behavior Trees Library in C++. Batteries included.” Accessed: Apr. 27, 2025. [Online]. Available: https://github.com/BehaviorTree/BehaviorTree.CPP

[18] “Groot | BehaviorTree.CPP.” Accessed: Aug. 04, 2025. [Online]. Available: https://www.behaviortree.dev/groot/

[19] M. Mayr, F. Rovida, and V. Krueger, “SkiROS2: A skill-based Robot Control Platform for ROS,” *IEEE International Conference on Intelligent Robots and Systems*, pp. 6273–6280, Jun. 2023, doi: 10.1109/IROS55552.2023.10342216.

[20] “rqt - ROS Wiki.” Accessed: Jul. 23, 2025. [Online]. Available: https://wiki.ros.org/rqt

[21] “RobotLabLTH/SkiROS2: A skill-based platform for ROS v.2 with knowledge representating, planning and reasoning.” Accessed: Apr. 26, 2025. [Online]. Available: https://github.com/RobotLabLTH/skiros2

[22] S. G. Brunner, F. Steinmetz, R. Belder, and A. Dömel, “RAFCON: a Graphical Tool for Task Programming and Mission Control,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9776 LNAI, pp. 347–355, May 2016, doi: 10.1007/978-3-319-68792-6_29.

[23] “The GTK Project - A free and open-source cross-platform widget toolkit.” Accessed: Aug. 10, 2025. [Online]. Available: https://www.gtk.org/

[24] “9. GUI Guide — RAFCON 2.2.1 documentation.” Accessed: May 11, 2025. [Online]. Available: https://rafcon.readthedocs.io/en/latest/gui_guide.html

[25] “RAFCON – Develop your robotic tasks using an intuitive graphical user interface | RAFCON.” Accessed: May 11, 2025. [Online]. Available: https://dlr-rm.github.io/RAFCON/

[26] “smach_viewer - ROS Wiki.” Accessed: May 11, 2025. [Online]. Available: http://wiki.ros.org/smach_viewer

[27] “smach/Tutorials/Smach Viewer - ROS Wiki.” Accessed: Jul. 27, 2025. [Online]. Available: http://wiki.ros.org/smach/Tutorials/Smach%20Viewer

[28] “SMACC – State Machine Asynchronous C++ | An Event-Driven, Asynchronous, Behavioral State Machine Library for real-time ROS (Robotic Operating System) applications written in C++.” Accessed: May 12, 2025. [Online]. Available: https://smacc.dev/

[29] “robosoft-ai/SMACC: An Event-Driven, Asynchronous, Behavioral State Machine Library for ROS (Robotic Operating System) applications written in C++.” Accessed: May 12, 2025. [Online]. Available: https://github.com/robosoft-ai/SMACC

[30] “(73) Getting Started with SMACC and the SMACC Viewer - YouTube.” Accessed: Aug. 05, 2025. [Online]. Available: https://www.youtube.com/watch?v=2KJSXTrn4QE&t=879s

[31] J. Z. Blanco and D. Lucrédio, “A holistic approach for cross-platform software development,” Apr. 2021, doi: 10.1016/j.jss.2021.110985.

[32] “JavaScript usage statistics, July 2025.” Accessed: Jul. 29, 2025. [Online]. Available: https://w3techs.com/technologies/comparison/cp-javascript?utm_source=chatgpt.com

[33] V. Pimentel and B. G. Nickerson, “Communicating and displaying real-time data with WebSocket,” *IEEE Internet Comput*, vol. 16, no. 4, pp. 45–53, 2012, doi: 10.1109/MIC.2012.64.

[34] “Node-Based UIs in React - React Flow.” Accessed: May 04, 2025. [Online]. Available: https://reactflow.dev/

[35] “Flask-SocketIO — Flask-SocketIO documentation.” Accessed: Aug. 07, 2025. [Online]. Available: https://flask-socketio.readthedocs.io/en/latest/

[36] “The WebSocket API (WebSockets) - Web APIs | MDN.” Accessed: Aug. 07, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[37] “temoto-framework/temoto_action_assistant.” Accessed: Aug. 07, 2025. [Online]. Available: https://github.com/temoto-framework/temoto_action_assistant

[38] “TypeScript: JavaScript With Syntax For Types.” Accessed: Aug. 10, 2025. [Online]. Available: https://www.typescriptlang.org/

8. Appendix A. Installation and Usage Guide for TeMoto Action Assistant

This appendix describes how to installation and run the **TeMoto Action Assistant** developed in this thesis.

The source code and documentation are available at:

https://github.com/temoto-framework/temoto_action_assistant.

Software requirements:

i) **Node.js:** version 18 or newer
ii) **React:** version 18 or newer
iii) **Python:** version 3.10 or newer
iv) **d) ROS 2:** Humble or newer

Hardware requirements:

a) **Minimum:** 4-core CPU, 8 GB RAM
b) **Recommended:** 8-core CPU, 16 GB RAM

Operating System: Requires Linux distribution to run in backend server in runtime mode, Developer mode can be run also on Windows and MacOS. Developed and tested with ROS 2 on Ubuntu 22.04 LTS; other modern Linux distributions may also work.

To setup the project, follow the instruction below:

1. **Install the repository using command line.**

```
git clone https://github.com/temoto-framework/temoto_action_assistant/edit/master/README.md
```

2. **Navigate to the root folder.**

```
cd temoto_action_assistant/
```

Start the backend server:

3. **Navigate to the backend folder**

```
cd backend/
```

4. **Install Python dependencies**

```
pip install -r requirements.txt
```

5. **Run the server:**

Run the server in runtime mode (using ROS 2)

python3 app.py --runtime

Or, run the server in developer mode (without ROS 2)

python3 app.py

Start the frontend GUI:

6. **Navigate to the root folder.**

```
cd temoto_action_assistant/
```

7. **Install dependencies**

```
npm install
```

8. **Run frontend GUI**

```
npm start
```

Open http://localhost:3000 to view it the GUI in your browser.

9. License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Kaarel-Richard Kaarelson ,
(author´s name)

grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

Design Tool for TeMoto Software Framework ,
(title of thesis)

supervised by Karl Kruusamäe and Robert Valner ;
(supervisor´s name)

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Kaarel-Richard Kaarelson
11/08/2025