CSSR for

Culturally Sensitive Social Robotics for Africa

D5.4.3 Robot Mission Interpreter

Due date: **31/12/2024**
Submission Date: **16/12/2024**
Revision Date: **07/06/2025**

Start date of project: **01/07/2023** Duration: **36 months**

Lead organisation for this deliverable: **Carnegie Mellon University Africa**

Responsible Person: **Tsegazeab Tefferi, CMU-Africa** Revision: **1.8**

Project funded by the African Engineering and Technology Network (Afretec) Inclusive Digital Transformation Research Grant Programme		
Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants (including Afretec Administration)	
RE	Restricted to a group specified by the consortium (including Afretec Administration)	
CO	Confidential, only for members of the consortium (including Afretec Administration)	

Executive Summary

Deliverable D5.4.3 focuses on the development of the `Robot Mission Interpreter` module, a central component within the CSSR4Africa software architecture. This module, also referred to as `behaviorController` ROS node, serves as the orchestrator for interpreting and executing robot mission specifications defined in Task 5.4.2, effectively bridging high-level mission planning with low-level robot execution. By leveraging the BehaviorTree.CPP library, the `behaviorController` node interprets XML-based mission scripts.

The `Robot Mission Interpreter` integrates seamlessly with eight distinct ROS nodes, including `animateBehavior`, `overtAttention`, `gestureExecution`, `textToSpeech`, `robotNavigation`, `tabletEvent`, `speechEvent`, and `robotLocalization`. These interactions are facilitated through ROS service calls and publish-subscribe mechanisms, allowing the module to control behaviors such as animating lifelike patterns, executing gestures, navigating environments, processing speech inputs, and detecting human presence. This comprehensive integration ensures that the robot can effectively engage with its environment and perform mission-critical tasks.

The development process encompasses a structured software engineering methodology, including requirements definition, module design, coded implementation, and unit testing, all adhering to the standards outlined in Deliverable D3.2 Use Case Scenario Definition.

In the work plan, this deliverable is assigned to the University of the Witswatersrand. However, the material in this version was developed and written by Carnegie Mellon University Africa. This was necessary because the Wits version was not available and, without it, it is not possible to build a complete, operational system and demonstrate the required CSSR4Africa functionality.

Contents

1 Introduction

This document describes the development and implementation of a ROS node for the execution of robot missions on the Pepper robot. The Robot Mission Interpreter functions as a central ROS node within the CSSR4Africa software architecture, orchestrating system behavior through ROS service calls and publish-subscribe mechanisms. It performs two key functions translating Robot Mission Specifications into executable robot commands, and processing real-time sensor data and status updates from other ROS nodes to guide robot behavior.

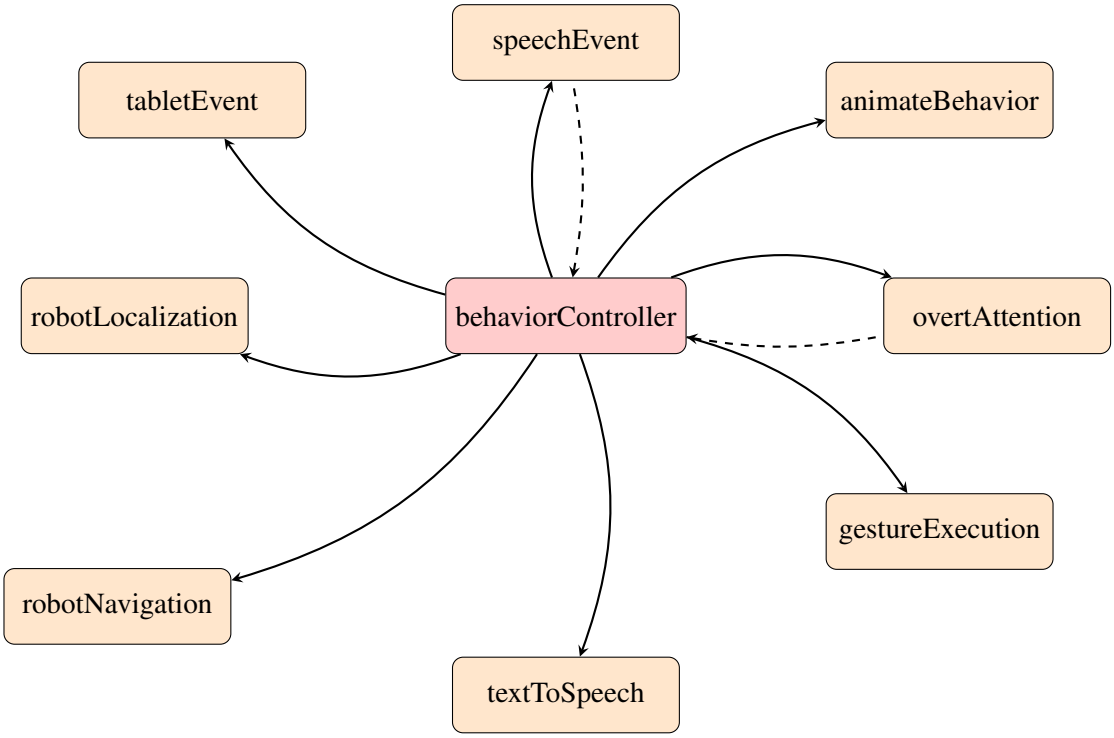

Figure 1: Interaction Diagram of the Robot Mission Interpreter Node with other ROS Nodes

Figure 1 illustrates the communication architecture of the Robot Mission Interpreter. The central node, shown in red, coordinates with eight peripheral ROS nodes shown in orange. The bidirectional communication is represented by arrows: solid arrows indicate service call from the interpreter to the nodes, while dashed arrows topic messages sent from the nodes to the interpreter.

This deliverable presents a comprehensive report detailing each phase of the software development lifecycle for the Robot Mission Interpreter module. Section 2 outlines the requirements definition process, aligning functional necessities with the project's overarching goals, thereby establishing the foundation for subsequent development efforts.

Section 3 delves into the module design, documenting the module's dependencies including the `BehaviorTree.CPP` library and the knowledge bases which contain cultural and environmental knowledge utilized during mission executions. The subsection explaining the mission execution framework provides detailed insights into critical components such as the `initializeTree` function (3.2.1), the Blackboard mechanism (3.2.2), custom Action and Condition Nodes (3.2.3), and the `HandleFallback` Action Node (3.2.4). The interface design is addressed with a focus on data exchange mechanisms utilizing ROS middleware and file input/output operations.

Section 4 describes the implementation of the module, including file organization and the configuration of various essential files. This includes external dependencies (4.1.1), main source files (4.1.2), configuration files (4.1.3), robot mission specifications (4.1.4), service definition files (4.1.5), message definition files (4.1.6), and package relevant files (4.1.7). Further details on the configuration file (4.2), input file (4.3), output file (4.4), topics file (4.5), topics subscribed (4.6), topics published (4.7), services supported (4.8), services called (4.9), and robot mission nodes (4.10) are provided to ensure a clear understanding of the module's operational setup.

Section 5 focuses on executing missions, detailing the prerequisites (5.1), the execution process (5.2), and the procedures for switching missions (5.3). Finally, Section 6 outlines the unit testing strategies, covering test configuration (6.1) and the launching of tests (6.2), ensuring the reliability and accuracy of the module's functionalities.

2 Requirements Definition

The `behaviorController` node is a vital component of the CSSR4Africa software system, responsible for interpreting robot mission specifications developed in Task 5.4.2 by implementing the use case scenarios outlined in Deliverable D2.1 User Case Scenario Definition. This deliverable is essential for identifying and addressing specific user expectations, ensuring that the node can accurately interpret and execute diverse mission scenarios across various operational environments, including both the physical robot and the simulator.

At its core, the `behaviorController` must proficiently parse mission specifications from a file written in the custom script language defined in Deliverable D5.4.2 Robot Mission Language, comprehending both syntax and semantics to extract meaningful instructions that dictate robot behaviors. It must orchestrate complex interaction dynamics between the robot and visitors, faithfully executing all defined behaviors and interaction protocols as specified in Deliverable D5.4.2 Robot Mission Language.

The node must interact seamlessly with various CSSR4Africa modules to perform tasks and process relevant information. To achieve this, it must communicate with eight different nodes by subscribing to two topics and making service calls to six servers. These interactions enable functionalities such as animating behaviors, executing gestures, controlling navigation, managing overt attention mechanisms, and initiating and processing interactions with humans.

The module must operate in two distinct modes: normal and verbose. In normal mode, the module executes mission scenarios without additional logging. In contrast, verbose mode allows the module to provide detailed information by printing logs to the terminal, which is invaluable for debugging and monitoring purposes. Additionally, the module must function with Automatic Speech Recognition (ASR) either enabled or disabled. When ASR is enabled, the module must processes human speech to facilitate audio-based interactions. When disabled, interactions must be managed through touch or physical inputs via the robot's tablet, allowing for alternative methods of communication.

Furthermore, the module must ensure robustness and reliability in mission execution by continuously monitoring the performance of nodes it communicates with and initiate fallback behaviors when necessary. It must employ error detection and recovery mechanisms to gracefully handle errors, service call failures, and unexpected behaviors, thereby maintaining system stability.

3 Module Design

The Robot Mission Interpreter module serves as the central orchestrator of the CSSR4Africa software architecture. It processes robot mission specifications defined in Task 5.4.2, interpreting and executing each action and condition node within the specification. Operating as a mission coordinator, it combines XML-based mission specifications with real-time inputs from other nodes and relevant information from the Knowledge Base, executing custom internal logic for each action and condition node to generate appropriate control commands.

As the primary control node of the system, it acts as the coordination layer between high-level mission planning and low-level robot execution. Operating within the ROS framework, this module interfaces with other nodes in the CSSR4Africa software package through ROS service calls and publish-subscribe mechanisms. It maintains active communication channels with eight different nodes, orchestrating their operations to achieve the specified mission objectives by executing various robot behaviors.

1. `animateBehavior`: Triggers the execution of life-like behavioral patterns to foster human-robot interaction
2. `gestureExecution`: Initiates specific body and hand gestures through the robot's motion control system
3. `overtAttention`: Initiates the robot's attention system to focus on specific environmental features or locations relevant to social interaction
4. `robotLocalization`: Determines the current pose of the robot.
5. `robotNavigation`: Directs the robot's movement system to navigate to specified locations or follow predetermined paths
6. `speechEvent`: Activates speech recognition processes to capture and process human verbal inputs
7. `tabletEvent`: Triggers the display of contextually appropriate interaction menus on the robot's tablet interface
8. `textToSpeech`: Commands the robot to deliver verbal messages through its speech synthesis system

The `behaviorController` node operates in two distinct modes: normal and verbose. In normal mode, the module executes mission scenarios with minimal output. In verbose mode, it provides comprehensive logging information to the terminal, facilitating debugging and system monitoring. The module also supports two interaction modes based on Automatic Speech Recognition (ASR) configuration. With ASR enabled, the module can process verbal commands through speech recognition. With ASR disabled, the module relies on the robot's tablet interface for touch-based interactions and physical inputs. These operational modes are configured through the `behaviorControllerConfiguration.ini` file.

3.1 Dependencies

The `behaviorController` node has the following external dependencies that are essential for its core functionality. These dependencies provide fundamental capabilities required for proper operation of the module and must be properly configured during setup.

3.1.1 BehaviorTree.CPP

BehaviorTree.CPP is an open-source C++ library designed to implement, read, and execute behavior trees. It utilizes an external Domain-Specific Language (DSL) instead of an internal one, as internal DSLs can hinder the maintainability and analyzability of the behavior model in the long term [1]. The library employs template metaprogramming instead of code generation, which provides a degree of type safety when implementing custom tree nodes without requiring users to engage with specialized code-generation tools, making the library feel akin to using a regular C++ library [2].

Trees in BehaviorTree.CPP are defined using a Domain-Specific scripting language based on XML, allowing them to be loaded at run-time; this means that even if the trees are written in C++, their morphology is not hard-coded, offering greater flexibility [3]. They can be created using any text editor, but this library provides a user-friendly graphical editor called Groot [4]. A more detailed description of how the robot mission specifications are developed is explained in D5.4.2 Robot Mission Language.

BehaviorTree.CPP was one of only two libraries out of 18 surveyed to have ROS support, be open-source, have a GUI tool to design missions, and be written in C++ [5]. It is popular, with 3.1k stars on GitHub, and the repository remains actively maintained with continuous contributions and updates as of December 15, 2024 [6]. It is currently being used in essential ROS2 packages, such as the Nav2 navigation stack [7]. Version 4.0 of the software was released in October of 2022 and was presented at ROSCon 2022 in Japan [8].

The `behaviorController` node was built by importing the fundamental components from this library, including the core behavior tree types, the XML parsing functionality for loading tree specifications at runtime, and the tree execution engine. This dependency provides all the necessary building blocks for constructing and executing behavior trees while allowing the mission interpreter to focus on implementing the specific behaviors and actions.

3.1.2 eSpeak

eSpeak is a compact, open-source text-to-speech (TTS) synthesizer available for Linux, Windows, Android, and other platforms. Supporting over 100 languages and accents, it is designed to be lightweight while offering broad linguistic coverage.

While the resulting speech is clear and can be produced at high speeds, it is less natural or smooth compared to larger synthesizers based on human speech recordings.

The software is available as a command-line program (for both Linux and Windows), capable of speaking text from files or standard input (stdin).[9].

eSpeak is not strictly required for the normal operation of the `behaviorController` node, but it becomes useful during debugging, particularly when vocalizing the names of action and condition nodes being executed, along with their parameters.

3.1.3 Knowledge Base

The Knowledge Base is a critical component of the CSSR4Africa software architecture, serving as a centralized data store containing essential information for the Pepper robot to operate effectively. It functions as the primary source of data upon which the `behaviorController` node depends, providing comprehensive details relevant to the robot's operational context.

Within this centralized repository, two main categories of knowledge are stored: Cultural knowledge information, as detailed in D5.4.1 Cultural Knowledge Ontology & Knowledge Base, and Environmental knowledge information, defined in D5.4.2 Robot Mission Language. Cultural knowledge allows the Pepper robot to perform tasks in a culturally sensitive manner, ensuring respectful interactions within specific cultural contexts. Environmental knowledge equips the robot with critical physical information necessary for safe navigation and effective interaction with its surroundings.

The Knowledge Base is made available within the "Utilities" software package, which includes both cultural and environmental knowledge base files alongside the software components required for accessing and retrieving stored values. Its internal organization adheres to the directory structure outlined in D3.2 Software Engineering Standards Manual. The contents merge code deliverables from D5.4.1 and D5.4.2 into a single, unified package to enhance ease of use, access, and maintainability.

Environment Knowledge

The Environment Knowledge Base is an organized repository explicitly developed to formally represent and store physical information about the robot's environment, facilitating robot navigation and interaction within its surroundings.

Utilizing the ontology defined in D5.4.2 Robot Mission Language, the Environment Knowledge Base systematically stores environmental parameters as key-value pairs within the file "environmentalKnowledgeBaseInput.dat," located in the "data" directory of the "Utilities" package. Each key-value pair comprises an alphanumeric key paired with numeric or alphanumeric values that encapsulate the environment knowledge.

Access to this environmental information is managed through the specialized C++ helper class named `EnvironmentalKnowledgeBase`, also defined in D5.4.2 Robot Mission Language. This class serves as the primary interface to the knowledge base, efficiently managing file access and providing public methods for enabling retrieval of stored environmental parameters.

Specifically tailored for the "Lab Tour" scenario described in D2.1 Use Case Scenario Definition, the `EnvironmentalKnowledgeBase` helper class encapsulates the necessary functionality for retrieving exhibit information, the robot's home location, and other critical details required to guide the robot's behavior and decision-making processes during its missions.

Code snippets illustrating the data structures used to represent these specifications, methods for accessing the data, detailed examples demonstrating the use of the class to read the environment knowledge base file, and practical examples showing how to retrieve values associated with a robot location given its identification number and how to retrieve the sequence of robot locations involved in a tour are provided in D5.4.2 Robot Mission Language.

Cultural Knowledge

The Cultural Knowledge Base is a structured system explicitly developed to formally represent and store culturally sensitive knowledge, specifically tailored for African contexts, with a particular emphasis on Rwandan culture. The ontology defining this cultural knowledge was initially provided in document D1.2 Rwandan Cultural Knowledge. Subsequently, based on this ontology, a repository

was implemented as described in document D5.4.1 Cultural Knowledge Ontology & Culture Knowledge Base. This repository comprises cultural parameter values, enabling robots to exhibit culturally appropriate and respectful behaviors, activities, actions, and motions during interactions with Rwandan individuals.

Technically, the system organizes cultural knowledge into key-value pairs, stored within the file "cultureKnowledgeBaseInput.dat", located in the "data" directory of the "Utilities" package. Each key-value pair consists of an alphanumeric key and an associated numeric or symbolic value, explicitly capturing aspects of cultural information.

Access to this structured cultural information is facilitated through a specialized C++ helper class named "CultureKnowledgeBase", which serves as the primary interface to the knowledge base. This helper class efficiently manages file access and provides two public methods, notably the critical "getValue()" function, enabling seamless retrieval of stored cultural parameters.

The `behaviorController` is the sole accessor to the Cultural Knowledge Base, serving as the central distributor of all cultural knowledge required by other nodes within the system, such as retrieving appropriate gesture parameters for respectful gestures for the `gestureExecution` ROS node and providing accurate distance values for culturally sensitive navigation to the `robotNavigation` ROS node.

One such example of cultural knowledge usage is the language used to conduct the current mission. This value is stored in the Cultural Knowledge Base's "cultureKnowledgeBaseInput.dat" file under the key "`PhraseLanguage`". During initialization, this value is retrieved using the "`getValue()`" accessor method and then transmitted to the `speechEvent` ROS node. This transmission occurs once during initialization and subsequently each time a request is made to the `/textToSpeech/say_text` service

```
KeyValueType keyValue;
CultureKnowledgeBase culturalKnowledgebase;

culturalKnowledgebase.getValue("PhraseLanguage", &keyValue);
std::string missionLanguage = keyValue.alphanumericValue;

//make service call to /speechEvent/setLanguage
//store in blackboard for use by /textToSpeech/say_text service
```

3.2 Mission Execution

The mission execution process is the core functionality of the `behaviorController` node. It involves the interpretation and execution of robot mission specifications defined in XML format, as described in D5.4.2 Robot Mission Language, using the BehaviorTree.CPP library. This process begins by parsing the XML file that defines the mission specification, which includes hierarchical nodes. The library instantiates these nodes and initializes a "BehaviorTree" object. During execution, the root node is recursively "ticked", with control nodes orchestrating the execution of child nodes, while leaf nodes interface with tasks that make up the selected scenario, as described in D2.1 Use Case Scenario Definition. Node statuses ("Success", "Failure", "Running") propagate dynamically, enabling real-time adaptation to environmental feedback. A shared "Blackboard" facilitates data exchange between nodes, ensuring parameterization and state persistence. This reactive loop allows the Pepper robot to adjust the mission on-the-fly.

Underpinning this process are several critical components, detailed below, each of which plays a crucial role in ensuring the successful execution of the mission.

3.2.1 The `initializeTree` function

The function `BT::Tree initializeTree(std::string missionSpecification)` is responsible for preparing the behavior tree for execution based on the robot mission specification given in XML format. It performs three critical steps:

1. File Loading: The function accepts a mission specification name as input and constructs the full file path by appending a ".xml" extension and locating it in the module's data directory.

2. Node Registration: Before the tree can be created, all custom Action and Condition nodes that will be used in the mission must be registered with the behavior tree factory. This is accomplished using the template function:

   ```
   template <typename T, typename... ExtraArgs>
   void registerNodeType(const std::string& ID, ExtraArgs... args)
   ```

   This registration step is crucial - if the XML specification references a node type that hasn't been properly registered, the `behaviorController` will encounter a fatal error when the tree's execution reaches that node.

3. Tree Creation: Finally, the function uses BehaviorTree.CPP's factory method:

   ```
   BT::Tree BehaviorTreeFactory::createTreeFromFile(
       const std::filesystem::path& file_path)
   ```

   to parse the XML specification and construct an executable behavior tree. This factory method handles the creation of all nodes, establishes their connections according to the XML structure, and returns a fully initialized `BT::Tree` object ready for execution.

3.2.2 Blackboard

The Blackboard serves as a centralized key/value storage mechanism that facilitates data sharing across all nodes within a Behavior Tree [3]. It provides two essential functions in the module.

First, it enables information sharing between different nodes executing within the same behavior tree. This is crucial for coordinating actions and maintaining state across different branches of the tree.

Second, since action and condition nodes have limited lifespans that end after their execution, the Blackboard acts as a persistent storage solution. Any data that needs to survive beyond a node's execution cycle must be stored in the Blackboard for future access.

An important point to note is that the Blackboard doesn't need to be explicitly created- it's automatically instantiated during tree initialization. Each action and condition node has access to it through the `config` object.

3.2.3 Custom Action and Condition Nodes

In a robot mission specification implemented using behavior trees, the composite nodes (Control Nodes and Decorator nodes) are provided by the BehaviorTree.CPP library. Custom implementations are necessary exclusively for Leaf Nodes, which consist of Action and Condition nodes.

Class Definition

Each Leaf Node is implemented as a C++ class that inherits from either `BT::SyncActionNode` or `BT::ConditionNode`, depending on whether it represents an Action or a Condition node. The class definition includes the following components:

1. **Constructor** The constructor for an Action node follows the

```
BT::SyncActionNode(const std::string& name, const NodeConfig& config)
```

signature, while that for a Condition node follows the

```
BT::ConditionNode(const std::string& name, const NodeConfig& config)
```

signature. During initialization, the constructor sets up necessary ROS components, such as service clients and topic subscribers, to facilitate the node's functionality.

```
BT::NodeStatus tick()
```

The `tick()` method encapsulates the core logic of each Leaf Node and is invoked during behavior tree traversal. Serving as the primary execution method for both Action and Condition nodes, `tick()` performs the node's specific functionality, which may include modifying Blackboard values, making service calls to CSSR4Africa ROS nodes, or executing custom actions based on messages received from subscribed topics.

This method returns one of three possible states:

- `BT::NodeStatus::SUCCESS`: Execution completed successfully.
- `BT::NodeStatus::FAILURE`: Execution failed.
- `BT::NodeStatus::RUNNING`: Execution is ongoing (applicable only to Action nodes).

3. Private Section The private section contains:

- Class member variables shared between the constructor and the `tick()` method.
- Callback methods for handling topic subscriptions.
- Other implementation-specific private members essential for the node's operation.

3.2.4 `HandleFallBack` Action Node

The `HandleFallback` action node is an important component within the `behaviorController`, that must be able to manage error recovery mechanisms when preceding actions encounter failures. Although its logic is not yet implemented, the `HandleFallback` node is intended to provide robust fallback strategies to ensure mission continuity and system resilience. Upon activation, typically triggered by a failure in a prior action node, `HandleFallback` will assess the nature of the failure and determine the appropriate recovery approach. This can involve executing a default recovery procedure that attempts to restore the system to a safe or initial state or invoking a custom, action-specific recovery function tailored to address the specific failure context.

Work is currently in progress to identify a suitable recovery strategy for the `HandleFallback` node that can ensure that it can effectively manage a wide range of failure scenarios and maintain mission integrity under adverse conditions.

4 Implementation

4.1 File Organization

The `behaviorController` node's codebase is organized into two primary components: the external dependency `BehaviorTree.CPP` and the core `behaviorController` module.

The `behaviorController` module follows a standard ROS package structure with dedicated directories for configuration files, mission data, service definitions, message specifications, and build configurations.

```
cssr_system
├── BehaviorTree.CPP
└── behaviorController
    ├── CMakeLists.txt
    ├── CSSR5AfricaLogo.svg
    ├── README.md
    ├── config
    │   ├── behaviorControllerConfiguration.ini
    │   ├── cultureKnowledgeBaseConfiguration.ini
    │   └── environmentKnowledgeBaseConfiguration.ini
    ├── data
    │   ├── cultureKnowledgeBaseInput.dat
    │   ├── cultureKnowledgeValueTypesInput.dat
    │   ├── environmentKnowledgeBaseInput.dat
    │   └── labTour.xml
    ├── include
    │   └── behaviorController
    │       ├── behaviorControllerInterface.h
    │       ├── cultureKnowledgeBaseInterface.h
    │       └── environmentKnowledgeBaseInterface.h
    ├── msg
    │   └── overtAttentionMode.msg
    ├── src
    │   ├── behaviorControllerApplication.cpp
    │   ├── behaviorControllerImplementation.cpp
    │   ├── cultureKnowledgeBaseImplementation.cpp
    │   └── environmentKnowledgeBaseImplementation.cpp
    └── srv
        ├── animateBehaviorSetActivation.srv
        ├── gestureExecutionPerformGesture.srv
        ├── overtAttentionSetMode.srv
        ├── robotLocalizationSetPose.srv
        ├── robotNavigationSetGoal.srv
        ├── speechEventSetLanguage.srv
        ├── speechEventSetStatus.srv
        ├── tabletEventPromptAndGetResponse.srv
        └── textToSpeechSayText.srv
```

Figure 2: File structure of the Robot Mission Interpreter

4.1.1 External Dependencies

- `BehaviorTree.CPP`: This directory houses the source code for the external BehaviorTree.CPP library, an essential dependency for the `behaviorController` node. Including the library's source code directly alongside the module ensures seamless integration and distribution, allowing for the use of the module right from the get go without having to install the library separately.

4.1.2 Main Source Files

- `include/behaviorController/`behaviorControllerInterface.h: Header file defining the public interfaces and abstract classes for the interpreter, facilitating communication between different components.
- `src/`behaviorControllerImplementation.cpp: The primary source file containing the core implementation of the Robot Mission Interpreter. This file encompasses the behavior tree initialization, node registration system, action and condition node implementations, and all other custom made minor functionalities.
- `include/behaviorController/`cultureKnowledgeBaseInterface.h and `include/behaviorController/`environmentKnowledgeBaseInterface.h: Header files that define the public interfaces for the Culture Knowledge Base and the Environment Knowledge Base, respectively, providing access to the corresponding knowledge parameters.
- `src/`cultureKnowledgeBaseImplementation.cpp and `src/`environmentKnowledgeBaseImplementation.cpp: Source files that implement the Culture Knowledge Base and the Environment Knowledge Base, respectively, providing methods for accessing and retrieving the corresponding knowledge parameters.
- `src/`behaviorControllerApplication.cpp: Source file serving as the entry point for the node, initializing the ROS node, loading configurations, and managing the execution flow.

4.1.3 Configuration Files

- `config/`behaviorControllerConfiguration.ini: Main configuration file specifying operational parameters, mode settings (normal or verbose), and other necessary configurations for the interpreter.
- `config/`cultureKnowledgeBaseConfiguration.ini: Configuration file for the Culture Knowledge Base, defining parameters such as the knowledge base file and the value types.
- `config/`environmentKnowledgeBaseConfiguration.ini: Configuration file for the Environment Knowledge Base, specifying the location of the environmental knowledge base file.

4.1.4 Data Files

- `data/`labTour.xml: Scenario script file written in the custom script language, detailing the interaction dynamics for a specific use case scenario, in this case the "Lab Tour" scenario, described in D2.1 Use Case Scenario Definition.

- `data/`cultureKnowledgeBaseInput.dat: File containing cultural knowledge parameters, such as gesture parameters and culturally relevant phrases, used by the robot during interactions.
- `data/`cultureKnowledgeValueTypesInput.dat: File containing the the type of the values in each cultural knowledge key-value pair is specified
- `data/`environmentKnowledgeBaseInput.dat: File containing environmental knowledge parameters, such as exhibit information and list of relevant phrases, used by the robot for navigation and interaction.

4.1.5 Service Definition Files

Located in srv/ directory, the service definition files are as follows:

- `srv/`animateBehaviorSetActivation.srv: Service definition for the `animateBehavior/set_activation` server.
- `srv/`gestureExecutionPerformGesture.srv: Service definition for the `gestureExecution/perform_gesture` server.
- `srv/`overtAttentionSetMode.srv: Service definition for the `overtAttention/set_mode` server.
- `srv/`robotLocalizationSetPose.srv: Service definition for the `robotLocalization/set_pose` server.
- `srv/`robotNavigationSetGoal.srv: Service definition for the `robotNavigation/set_goal` server.
- `srv/`speechEventSetLanguage.srv: Service definition for the `speechEvent/set_language` server.
- `srv/`speechEventSetStatus.srv: Service definition for the `speechEvent/set_status` server.
- `srv/`tabletEventPromptAndGetResponse.srv: Service definition for the `tabletEvent/prompt_and_get_response` server.
- `srv/`textToSpeechSayText.srv: Service definition for the `textToSpeech/say_text` server.

4.1.6 Message Definition Files

Located in msg/ directory, the message definition files are as follows:

- `overtAttentionMode.msg`: Message definition for `/overtAttention/mode` topic.

4.1.7 Package Relevant Files

- `README.md`: Documentation file providing an overview of the `behaviorController` node, setup instructions, usage guidelines, and other relevant information for developers and users.
- `CMakeLists.txt`: Build configuration file specifying dependencies, include directories, and compilation instructions necessary to build the `behaviorController` node within the ROS workspace.
- `package.xml`: ROS package manifest detailing package metadata, dependencies on other ROS packages, and other essential information required for package management and integration within the ROS ecosystem.

4.2 Configuration File

The operation of the behaviorController node is determined by the contents of the configuration file that contains a list of key-value pairs as shown below. The configuration file is named `behaviorController.ini`

Table 1: Configuration Parameters for the Robot Mission Interpreter Node

Key	Value	Description
`scenarioSpecification`	`<specificationFile>`	Specifies the name of the robot mission specification file. The node will look for an xml file, excluding extension, with that name from the data folder.
`verboseMode`	`true` or `false`	Specifies whether diagnostic data is to be printed to the terminal.
`asrEnabled`	`true` or `false`	Specifies whether Automatic Speech Recognition is enabled on the platform or not.
`audioMode`	`true` or `false`	Specifies whether vocalization of audio debugging messages should be enabled or not

4.3 Input File

There are three input files to the system. The `scenarioSpecification` defined in the configuration file serves as the source for the robot mission specification. This file must contain a robot mission specification in XML format, as a behavior tree, that adheres to the Groot2 output structure, as detailed in Deliverable 5.4.2. Any deviation from this format will cause the `behaviorController` node to fail during initialization when parsing the specification. Additionally, the `cultureKnowledgeBaseInput.dat` and `environmentKnowledgeBaseInput.dat` files are used to provide cultural and environmental knowledge parameters, respectively. These files must be located in the "data" directory of the `behaviorController` package.

4.4 Output File

There is no output data file for the behaviorController node. The result of the execution of each action and condition node is outputted as a diagnostic messages on the screen, depending on the value of `verboseMode` key in the configuration file. Additionally, the Groot2 IDE can also be connected to the behaviorController node and status of the mission and of each node displayed in its user interface.

4.5 Topics File

There are no topic files for the behaviorController node.

4.6 Topics Subscribed

This node subscribes to the following two topics listed in the table. The table describes the topics subscribed, the format of the messages published by the respective node, the description of that message and the deliverable within which there is further explanation of the type of variable used.

Table 2: Topics Subscribed by the Robot Mission Interpreter

Topic	Message Format	Description	Deliverable
`/overtAttention/mode`	`state, value`	Contains the currently set mode and additional status for the "seeking", "social", and "scanning" modes.	D5.3
`/speechEvent/text`	`detected_text`	Contains the transcriptions of the utterances detected by the Speech Event node.	D4.3.2

4.7 Topics Published

This node doesn't publish any topics.

4.8 Services Supported

This node doesn't provide or advertise any server for any service.

4.9 Services Called

The node interacts with the services detailed in the table below. Each entry specifies the service name, the message format required, the expected effects or observations, and references to the corresponding deliverables that provide comprehensive explanations of the arguments passed and the return values from each service call.

Table 3: Service Messages and Their Effects

Service	Message Format	Effect	Deliverable
`animateBehaviour/set_activation`	`state`	Enable or Disable the Animate Behavior mechanism.	D5.2
`gestureExecution/perform_gesture`	`gesture_type`, `gesture_id`, `gesture_duration`, `bow_nod_angle`, `location_x`, `location_y`, `location_z`	Invokes the gesture subsystem to perform a type of gesture with specified coordinates. The coordinates are only necessary if the `gesture_type` is `deictic`.	D5.5.1
`overtAttention/set_mode`	`state`, `location_x`, `location_y`, `location_z`	Sets the mode of attention for the Overt Attention node to follow.	D5.3
`robotLocalization/set_pose`	`x, y, theta`	Informs the localization subsystem of the current pose of the robot	D4.2.4
`robotNavigation/set_goal`	`goal_x, goal_y, goal_theta`	Invokes the navigation subsystem to move to the specified coordinates.	D5.5.4
`speechEvent/set_enabled`	`status`	Enables and Disables the ASR	D5.5.4
`speechEvent/set_language`	`language`	Sets the language for the ASR to transcribe utterances to	D5.5.4
`tabletEvent/prompt_and_get_response`	`message`	Sends the text to be printed as a message on the Tablet.	D4.3.1
`textToSpeech/say_text`	`message`, `language`	Sends the text to be converted to an audio signal based on the language specified and played on the robot's loudspeakers.	D5.5.2.4

4.10 Robot Mission Nodes

A total of 23 action and condition nodes have been developed for the `behaviorController` module. These custom nodes are comprehensively listed in Table 4. The table provides the following details for each:

- Name: The identifier of the node.
- Type: Specifies whether the node is an action or a condition.
- Description: A brief overview of the node's functionality.
- Subscribed Topic: The topic to which the node subscribes.
- Service Called: The service that the node invokes.

While these nodes were crafted with the "Lab Tour scenario" in mind, described in D2.1 Use Case Scenario, they are designed to be modular and highly reusable across various missions. With a few exceptions, these nodes can be seamlessly integrated into different mission scenarios by supplying a different mission specification as input to the interpreter, by modifing the configuration file. This modularity ensures that the behavior tree remains flexible and scalable, allowing for efficient adaptation to diverse operational requirements without necessitating extensive reconfiguration or redevelopment of existing nodes.

Table 4: Implemented Mission Nodes

Node	Type	Description	Topic Subscribed	Service Called
`DescribeExhibitSpeech`	Action	Sends the description of the current exhibit to the textToSpeech ROS node.	None	`/textToSpeech/say_text`
`GetVisitorResponse`	Action	Retrieves the response of the visitor from the ASR as transcribed text and sets it in the blackboard.	`/speechEvent/text`	None
`HandleFallBack`	Action	A generic node used to handle scenarios when an Action node returns a failure. Its functionality is currently undefined.	None	None
`HasVisitorResponded`	Condition	Checks if the visitor has responded to a query by retrieving the flag from the blackboard.	None	None
`IsASREnabled`	Condition	Checks the preset value from the configuration to determine if Automatic Speech Recognition is enabled.	None	None
`IsListWithExhibit`	Condition	Checks if there is an exhibit that has not yet been visited.	None	None
`IsMutualGazeDiscovered`	Condition	Checks if the overtAttention ROS node is publishing "seeking" for the "state" and '1' for the "value" parameters.	`/overtAttention/mode`	None
`IsVisitorDiscovered`	Condition	Checks if the overtAttention ROS node is publishing "scanning" for the "state" and '1' for the "value" parameters.	`/overtAttention/mode`	None
`IsVisitorPresent`	Condition	Checks if the overtAttention ROS node is publishing "social" for the "state" and '1' for the "value" parameters.	`/overtAttention/mode`	None
`IsVisitorResponseYes`	Condition	Checks if the visitor response is akin to a "Yes" or "No" by retrieving it from the Blackboard.	None	None

Node	Type	Description	Topic Subscribed	Service Called
`Navigate`	Action	Sends the coordinates to be navigated towards to the robotNavigation ROS node.	None	`/robotNavigation/set_goal`
`PerformDeicticGesture`	Action	Retrieves the coordinate values for the current gesture from the Blackboard and sends them to the gestureExecution ROS node. The "gesture_type" parameter is set to "deictic".	None	`/gestureExecution/perform_gesture`
`PerformIconicGesture`	Action	Sends the appropriate "iconic" gesture parameters to the gestureExecution ROS node.	None	`/gestureExecution/perform_gesture`
`PressYesNoDialogue`	Action	Sends a message to the tabletEvent ROS node to initiate a "Yes/No" dialogue on the robot's tablet. Stores the response in the Blackboard.	None	`/tabletEvent/prompt_and_get_response`
`RetrieveInitialLocation`	Action	Retrieves the coordinates of start location from the `Environment Knowledge Base` and stores the value in the Blackboard.	None	None
`RetrieveListOfExhibits`	Action	Retrieves the exhibits to be visited from the Knowledge Base. The retrieved values are stored in the Blackboard.	None	None
`SayText`	Action	Sends a text to the text-to-speech ROS node to be vocalized	None	`/textToSpeech/say_text`
`SelectExhibit`	Action	Selects the next exhibit to visit from the Blackboard. Sets values for other nodes to retrieve and use.	None	None
`SetAnimateBehavior`	Action	Sets the animate parameter "state" to "enabled" or "disabled" and sends it to the animateBehavior ROS node.	None	`/animateBehavior/set_activation`

Node	Type	Description	Topic Subscribed	Service Called
SetOvertAttentionMode	Action	Sets the attention parameter "state" (e.g., "disabled", "scanning", "social") and sends it to the overtAttention ROS node.	None	/overtAttention/set_mode
SetRobotPose	Action	Sends the start location retrieved from the Environment Knowledge Base to the robot localization ROS node	None	/robotLocalization/set_pose
SetSpeechEvent	Action	Sets the "status" parameter for the ASR and sends it to the speechEvent ROS node	None	/speechEvent/set_enabled
StartOfTree	Action	A node used for debugging purposes to indicate the start of the mission by printing on the terminal and to set certain initialization parameters.	None	None

5 Executing Missions

5.1 Prerequisites

Before executing the `behaviorController` node, several prerequisites must be met:

- All eight CSSR4Africa ROS nodes (detailed in the "Module Design" section) must be running under the same `roscore` instance, either on the physical robot or simulator platform
- The `behaviorController` node must be properly built and sourced in the ROS workspace
- The `behaviorControllerConfiguration.ini` file must be correctly configured with the desired settings
- The target mission specification file must be present in the `data/` directory

Optional: Enabling Audio Debugging Mode
If audio debugging is desired, the `eSpeak` utility must be installed on the system. `eSpeak` is a compact, open-source software speech synthesizer for English and other languages [9]. It is commonly used in Linux environments to convert text to spoken voice, which can be useful for debugging the execution of the robot mission execution.

To install `eSpeak` on a Debian-based Linux system (e.g., Ubuntu), run the following command in the terminal:

```
# Install eSpeak for audio debugging
sudo apt-get install espeak
```

Once installed, ensure that your debugging configuration enables audio output by setting `audioMode` as "true" in the configuration file.

5.2 Execution Process

To launch the `behaviorController` node, execute:

```
# Launch the Robot Mission Interpreter
rosrun cssr_system behaviorController
```

If any of the required ROS nodes are not running, the `behaviorController` will display an error message and terminate execution.

5.3 Switching Missions

To execute a different mission:

1. Place the new robot mission specification file in the `data/` directory
2. Update the `behaviorControllerConfiguration.ini` file to reference the new specification file
3. Execute the `behaviorController` node as described above

6 Unit Tests

The unit tests for the `behaviorController` node are organized within a dedicated ROS package located in the `unit_tests` directory. The testing methodology adheres to the guidelines outlined in Deliverable D3.5 System Integration and Quality Assurance. This package encapsulates all required components for validating the behaviorController's functionality, including configuration files, stubs and drivers to simulate the actual ROS nodes' functionality, as well as source code for test harnesses and utility functions.

```
unit_tests/
└── behaviorControllerTest/
    ├── CMakeLists.txt
    ├── CSSR4AfricaLogo.svg
    ├── README.md
    ├── config/
    │   └── behaviorControllerTestConfiguration.ini
    ├── data/
    │   └── behaviorControllerTestOutput.dat
    ├── include/
    │   └── behaviorControllerTest/
    │       ├── behaviorControllerTestInterface.h
    │       └── behaviorControllerTestUtilitiesInterface.h
    ├── launch/
    │   ├── behaviorControllerTestLaunchHarness.launch
    │   ├── behaviorControllerTestLaunchRobot.launch
    │   └── behaviorControllerTestLaunchSimulator.launch
    ├── msg/
    │   └── overtAttentionMode.msg
    ├── src/
    │   ├── behaviorControllerTestApplication.cpp
    │   ├── behaviorControllerTestDriver.cpp
    │   ├── behaviorControllerTestImplementation.cpp
    │   ├── behaviorControllerTestStub.cpp
    │   └── behaviorControllerTestUtilities.cpp
    └── srv/
        ├── animateBehaviorSetActivation.srv
        ├── gestureExecutionPerformGesture.srv
        ├── overtAttentionSetMode.srv
        ├── robotLocalizationSetPose.srv
        ├── robotNavigationSetGoal.srv
        ├── speechEventSetLanguage.srv
        ├── speechEventSetStatus.srv
        ├── tabletEventPromptAndGetResponse.srv
        └── textToSpeechSayText.srv
```

Figure 3: File structure of the Unit Tests for the Robot Mission Interpreter

To execute the unit tests, users must first install the required software packages as detailed in Deliverable D3.3 Software Installation Manual. The unit tests' operation is controlled by the configuration file `behaviorControllerTestConfiguration.ini`, which contains essential key-value pairs for test execution.

6.1 Test Configuration

Table 5: Configuration Parameters for the Robot Mission Interpreter Node Unit Tests

Key	Value	Description
`failureRate`	`0.1 – 1`	Specifies the failure rate of the service calls made to servers advertised by the stubs. A value of 0.1 corresponds to a 10% failure rate, meaning that 10% of service calls will result in a simulated failure response. The default value is 0.1.
`arrivalRate`	$1 - \infty$	Specifies the rate at which messages are sent to a topic by the drivers, modeled using a Poisson distribution. A value of 1 indicates that, on average, 1 message is sent per minute.
`verboseMode`	`true` or `false`	Determines whether diagnostic data is printed to the terminal. Setting this to `true` enables detailed logging, which is useful for debugging and monitoring, while `false` suppresses such output for normal operation.
`traversal`	`partial##` or `complete`	Determines which path of the robot mission specification to execute for the test. For the "Lab Tour" mission, it accepts 5 possible values. Four different "partial" paths and one "complete" path

6.2 Traversal

In the `behaviorControllerTest`, the `traversal` configuration key determines which part of the mission specification (a behavior tree) should be executed and tested. Since the mission is structured as a tree, this effectively selects which sub-path of the tree to traverse during testing.

For the *Lab Tour* scenario, there are four predefined partial paths, each representing a specific failure condition, as well as one complete path representing a successful execution.

Table 6: Predefined Traversal Paths in Lab Tour Scenario

Key	Description	Failing Condition Node
`partial01`	Visitor is detected but mutual gaze is not established	`IsMutualGazeDiscovered`
`partial02`	Visitor does not respond when asked if they would like a tour	`HasVisitorResponded`
`partial03`	Visitor explicitly declines the offer to have a tour	`IsVisitorResponseYes`
`partial04`	Visitor leaves midway through the tour	`IsVisitorPresent`
`complete`	The tour completes successfully without triggering any failure conditions	None

By default, the traversal configuration is set to `complete`, meaning that the entire mission specification will be executed without any simulated failures. This allows for a comprehensive test of the

behaviorController's functionality under normal conditions.

If any of the action or condition nodes that are part of the execution flow of the selected mission specification file fail, the `behaviorControllerTest` application will identify and log the failure. A sample of the result of such a test is presented below.

6.3 Launching Tests

To execute the `behaviorController` tests, use the following launch command:

```
    # Launch the Physical Robot
    roslaunch unit_tests behaviorControllerLaunchRobot.launch \
    robot_ip:=<robot_ip> roscore_ip:=<roscore_ip> \
    network_interface:=<network_interface>
```

Ensure that the IP addresses robot_ip and roscore_ip and the network interface network_interface are correctly set based on your robot's configuration and your computer's network interface.

```
# Launch the Robot Mission Interpreter Unit Tests
    roslaunch unit_tests behaviorControllerTestLaunchHarness.launch \
    launch_drivers:=true \                                          \
    launch_test:=true
```

The above command will launch all eight simulated ROS nodes along with the `behaviorController` node. The `behaviorController` will execute the mission by interacting with with the servers and topics made available by the stubs and drivers. The list of the topics and servers that will be simulated are the ones listed in Table 2 and Table 3 respecively. It is important to note that the robot mission specification file used for the tests is the one defined within the configuration file of the `behaviorController` node.

Test results are displayed on the terminal and recorded in `behaviorControllerTestOutput.dat`. This file logs the Action or Condition nodes that failed, the Topic or Service associated with each failed node, and any additional messages available at the time of failure. A sample of the test results with a `complete` path traversal is shown below.

```
        Test Results
        Date: 2025-06-07 12:30:10

        StartOfTree -> Passed!
        SetSpeechEvent -> Passed!
        RetrieveListOfExhibits -> Passed!
        SetRobotPose -> Passed!
        SetAnimateBehavior -> Passed!
        SetOvertAttentionMode -> Passed!
        IsVisitorDiscovered -> Passed!
        SayText -> Passed!
        PerformIconicGesture -> Passed!
        IsMutualGazeDiscovered -> Passed!
        GetVisitorResponse -> Passed!
        HasVisitorResponded -> Passed!
        IsVisitorResponseYes -> Passed!
        IsListWithExhibit -> Passed!
        SelectExhibit -> Passed!
        Navigate -> Passed!
        IsVisitorPresent -> Passed!
        DescribeExhibit -> Passed!
        PerformDeicticGesture -> Passed!
```

References

[1] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. Behavior trees and state machines in robotics applications, 2023. arXiv preprint arXiv:2208.04211.

[2] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wasowski. Behavior trees in action: A study of robotics applications. In *Proc. 13th ACM SIGPLAN Int. Conf. on Software Language Engineering*, volume SPLASH '20, pages 196–209, 2020.

[3] Behaviortree.cpp website. `https://www.behaviortree.dev`. Accessed: 2024-12-14.

[4] Eric Dortmans, Teade Punter, and Ramadoni Syahputra. Behavior trees for smart robots practical guidelines for robot software development. *Journal of Robotics*, 2022, 2022.

[5] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A survey of behavior trees in robotics and ai, 2020. arXiv preprint arXiv:2005.05842.

[6] Behaviortree.cpp github repository. `https://github.com/BehaviorTree/BehaviorTree.CPP`. Accessed: 2024-12-14.

[7] Ros navigation stack. `https://navigation.ros.org`. Accessed: 2024-12-14.

[8] Roscon 2022 presentation. `https://roscon.ros.org/2022/`. Accessed: 2024-12-14.

[9] espeak source code repository. `https://github.com/espeak-ng/espeak-ng`. Accessed: 2025-06-07.

Principal Contributors

The main authors of this deliverable are as follows (in alphabetical order).

Tsegazeab Tefferi, Carnegie Mellon University Africa.
David Vernon, Carnegie Mellon University Africa.

Document History

Version 1.0

First draft.
Tsegazeab Tefferi.
15 December 2024.

Version 1.1

Updated Executive Summary.
Fixed formatting issues.
David Vernon.
31 December 2024.

Version 1.2

Reflected the change in ROS node name from "scriptInterpreter" to "behaviorController"
Added a new subsection in Section 3, "Data Specifications"
Corrected errata in the document.
Tsegazeab Tefferi.
24 January 2025.

Version 1.3

Removed `\include` directives and inserted all text in a single .tex source file for ease of editing.
Augmented description of `scenarioSpecification` in Table 5.
David Vernon.
28 January 2025.

Version 1.4

Removed all references of the `knowledgeBase` ROS node, to reflect changes made to the workplan
Added a sub-section under "3.1 Dependencies", titled "3.1.1 Knowledge Base" that provides an explanation of the component and the helper class used to access it.
Removed "3.2.5 Data Specifications" and combined its content with "3.1.1 Knowledge Base"
Expanded upon 3.1 Mission Execution to better explain the process.
Corrected remaining improperly used quotation marks.
Regularized deliverable referencing format.
Fixed formatting issues.
Tsegazeab Tefferi.
31 January 2025.

Version 1.5

Completely re-wrote the "3.1.2 Knowledge Base" that containes the descriptions of Environment and Cultural Knowledge Bases, and updated all relevant sections.
Updated the "4.10 Services Called" to reflect the changes made to the services made available by textToSpeech and speechEvent ROS nodes.
Added a new style for C++ code snippets.
Tsegazeab Tefferi.
22 March 2025.

Version 1.6

Corrected the issue of paragraphs, figures, and tables overflowing beyond the margin boundaries.
Tsegazeab Tefferi.
26 March 2025.

Version 1.7

Fixed formatting problems caused by using geometry and lmodern packages: incorrect margins and incorrect fonts on cover page, respectively. Changed to a smaller size for teletype font, replacing instances of `...` with `....` Removed excessive use of bold font by replacing instances of **...** with .... Fixed several formatting issues where lines extend beyond margin boundaries and table columns. Added internal documentation to highlight sections, subsections, subsubsections, and paragraphs.
David Vernon.
4 May 2025.

Version 1.8

Updated multiple sections to reflect the changes made to the `behaviorController` node codebase since the documents last update. Minor changes to Introduction and Module Design and significant changes to Implementation.
Expanded the unit tests section.
Tsegazeab Tefferi.
07 June 2025.