

## D5.4.2 Robot Mission Language

Due date: **30/06/2024**  
Submission Date: **05/03/2025**  
Revision Date: **18/04/2025**

Start date of project: **01/07/2023**

Duration: **36 months**

Lead organisation for this deliverable: **Carnegie Mellon University Africa (for Wits)**

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

Revision: **1.6**

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

## Executive Summary

This deliverable represents the outcome of Task 5.4.2. It comprises three elements: (i) a mode of abstract modelling — behavior trees — that can be used to formally specify the interactions in the use case scenarios and enact them in a culturally sensitive manner using the culture knowledge base and an environment knowledge base, (ii) an environment knowledge base file with the information required to complete the robot mission, and (iii) the documented software required to compile a C++ helper class `EnvironmentKnowledgeBase` to read the environment knowledge base file, store the knowledge, and make the knowledge accessible through a suite of access methods. As such, this deliverable, along with Deliverable D6.1 Use Case Implementation with the specification of the two use case scenarios using behavior trees, provides the input for the development in Task 5.4.3 of an interpreter that can translate this abstract specification into robot actions, thereby enacting the use case scenarios defined in Tasks 2.1, 2.2, and 2.3 and documented in Deliverables D2.1, D2.2, and D2.3.

In the CSSR4Africa work plan, this deliverable and Deliverables D5.4.1 and D5.4.3 were assigned to the University of the Witwatersrand. However, the material in this report was developed and written by Carnegie Mellon University Africa. This was necessary because, due to extensive delays in the delivery of the Pepper robot to the University of the Witwatersrand, little or no progress had been made. Since the cultural knowledge ontology and culture knowledge base, the robot mission language, and the robot mission interpreter are essential for integrating and demonstrating the use case scenarios, Carnegie Mellon University Africa took joint responsibility for these three deliverables (among others, specifically D5.5.2.1, D5.5.4, D6.1, and D6.2). Since this involved a significant amount of additional, unplanned effort, only one use case, the laboratory tour, has been implemented to date, leaving the second use case, the receptionist, to be implemented later.

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Specification of Robot Missions with Behavior Trees</b>	<b>4</b>
2.1	Behavior Trees . . . . .	4
2.2	Flow of Execution . . . . .	5
2.3	Structure . . . . .	5
2.3.1	Composite Nodes . . . . .	5
2.3.2	Leaf Nodes . . . . .	8
2.4	Groot2 . . . . .	10
<b>3</b>	<b>Environment Knowledge Ontology and Knowledge Base</b>	<b>14</b>
<b>4</b>	<b>Environment Knowledge Base Implementation</b>	<b>16</b>
4.1	File Organization . . . . .	16
4.2	Configuration File . . . . .	16
4.3	Environment Knowledge Base . . . . .	18
4.4	Output Data File . . . . .	18
4.5	Class Definition . . . . .	19
4.5.1	Constructor . . . . .	19
4.5.2	Destructor . . . . .	19
4.5.3	Private Data . . . . .	19
4.5.4	Public Access Methods . . . . .	21
<b>5</b>	<b>Example Application</b>	<b>22</b>
	<b>Appendix A The EnvironmentKnowledgeBase Class</b>	<b>25</b>
	<b>References</b>	<b>27</b>
	<b>Principal Contributors</b>	<b>28</b>
	<b>Document History</b>	<b>29</b>

## 1 Introduction

This deliverable documents the outcome of Task 5.4.2 which is concerned with providing a methodology, or language, for specifying robot missions. Section 2 provides the foundational knowledge necessary for understanding the specification and implementation of robot missions using behavior trees. Behavior trees have emerged as a robust and flexible alternative to state machines, offering a structured and intuitive approach for formally specifying interactions and decision-making processes within use-case scenarios [1]. This section begins with a short overview behavior trees (Section 2.1). Subsequently, the execution flow of behavior trees is discussed (Section 2.2), clarifying how nodes are evaluated and executed. The structural components essential to behavior trees are then systematically detailed (Section 2.3). Finally, the Groot2 tool, a graphical interface specifically developed to facilitate the design and management of behavior trees, is introduced (Section 2.4) along with a practical demonstration of its usage provided by designing a simple behavior tree example.

The actual specification of the robot missions for the two use case scenarios defined in Tasks 2.1, 2.2, and 2.3 and documented in Deliverables D2.1, D2.2, and D2.3 is detailed in Deliverable D6.1 Use Case Implementation.

Since we are particularly focused on enacting these missions in a culturally sensitive manner, we require both a cultural knowledge ontology & culture knowledge base, and an environment knowledge ontology & environment knowledge base. The former is described in Deliverable D.5.4.1, while the latter is described in Section 3 of this deliverable.

The deliverable concludes with Section 4 which addresses the implementation of the environment knowledge base and, specifically, with the description of a C++ helper class to read the environment knowledge base file, store the knowledge, and make the knowledge accessible through a suite of access methods. As such, it provides the input for the development in Task 5.4.3 of an interpreter that can translate the abstract behavior tree specifications shown in “Implementation” into robot actions, thereby enacting the use case scenarios defined in Tasks 2.1, 2.2, and 2.3 and documented in Deliverables D2.1, D2.2, and D2.3.

In the work plan, this deliverable and deliverable D5.4.3 were assigned to the University of the Witwatersrand. However, the material in this report was developed and written by Carnegie Mellon University Africa. This was necessary because of unavoidable delays in the completion of the associated task by Wits, and because the robot mission language and the robot mission interpreter, are essential for integrating and demonstrating the use case scenarios.

## 2 Specification of Robot Missions with Behavior Trees

### 2.1 Behavior Trees

A behavior tree is a powerful and flexible framework for designing the control architecture of an autonomous agent, such as a robot or a virtual character in a computer

game. Behavior trees decompose complex tasks into a hierarchy of simpler, modular actions and decisions. This hierarchical structure typically consists of **composite nodes**, which control the flow between tasks and **mission/leaf nodes**, which execute the actual actions or check conditions [1, 2].

## 2.2 Flow of Execution

The execution of a behavior tree begins at its root node, which periodically generates signals called “**Ticks**” at a defined frequency. These ticks propagate downward through the tree, following the traversal logic defined by control-flow nodes. When ticked, each node executes its associated task; leaf nodes specifically perform robotic actions or evaluate conditions, returning one of three possible statuses: **Running**, if the task is still ongoing, **Success**, if the goal has been achieved, or **Failure**, if the task cannot be completed. Unlike state machines, Behavior Trees do not allow direct jumps or “goto” statements; instead, composite nodes locally influence the traversal logic, guiding the flow of execution dynamically and modularly [3, 4].

The variant of behavior trees used in robotics is predominantly a time-triggered, activity-based behavioral modeling language. Rather than explicitly shifting control tokens or managing states as in traditional control loops, this approach periodically triggers the entire model at fixed intervals, similar to clock-driven circuits. Each tick initiates a full traversal of the behavior tree, dynamically branching according to various node types [1]. In contrast, behavior tree implementations in video games use event driven programming as opposed to time-triggered controls [3, 5].

## 2.3 Structure

### 2.3.1 Composite Nodes

Composite nodes are internal nodes in a behavior tree that manage the control flow among their child nodes. They do not execute actions or evaluate conditions directly. Instead, they define the order and conditions under which their children are ticked (activated), ensuring that the overall decision-making process follows a structured pattern.

### Root

The Root node serves as the entry point for every traversal of the behavior tree. It has exactly one child node and is re-entered at each epoch. The tree is traversed from the Root to the leaf nodes and back, following a depth-first approach. It is typically represented as an inverted triangle shape with the label “Root” at the center; see Fig. 1.

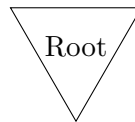


Figure 1: Root node of a behavior tree.

### Sequence

The Sequence node is a composite node that enforces a strict order of execution among its children. When ticked, it processes its child nodes sequentially from left to right. The Sequence node behaves as follows.

- It sends a tick to the first child.
- If the child returns **Success**, the tick is passed to the next child.
- If a child returns either **Failure** or **Running**, the Sequence node immediately returns that status and halts further processing.
- Only if every child returns **Success** does the Sequence node return *Success*.

This design makes the Sequence node ideal for modeling tasks that require multiple steps to be completed in a specific order. While many representations depict the Sequence node with a (‘‘Seq’’) label inside a box, alternative symbols — such as a rectangle containing a right-pointing arrow (‘‘→’’) to indicate directional flow — may also be used, depending on the chosen visual convention; see Fig. 2.

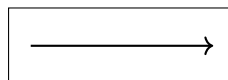


Figure 2: Sequence node of a behavior tree.

### Fallback

The Fallback node, also known as the Selector, is a composite node that provides an alternative execution path among its children. When ticked, it processes its children sequentially from left to right with the following behavior.

- It sends a tick to the first child.
- If the child returns **Failure**, it proceeds to tick the next child.
- If a child returns either **Success** or **Running**, the Fallback node immediately returns that status and stops processing further children.
- The Fallback node returns **Failure** only if all its children return **Failure**.

This design makes the Fallback node ideal for situations where multiple alternative behaviors are available, and success is achieved as soon as one alternative succeeds. It is commonly represented by a box containing the label “?”; see Fig. 3.

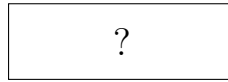


Figure 3: Fallback node of a behavior tree.

### Parallel

The Parallel node is a composite node that simultaneously triggers all of its child nodes and aggregates their outcomes to determine its own status. Its behavior is as follows.

- The node sends a tick to every child at the same time.
- A user-defined threshold  $M$  is set (with  $M \leq N$ , where  $N$  is the total number of children).
- The Parallel node returns **Success** if at least  $M$  children return **Success**.
- It returns **Failure** if at least  $N - M + 1$  children return **Failure**.
- If neither condition is met, the node returns **Running**.

It is important to note that “parallel” does not always mean that the children are executed concurrently; some implementations may process the children sequentially while simulating parallel behavior [3, 6]. The Parallel node is typically symbolized by a rectangle that contains the two right directed arrows in parallel inside of it; see Fig. 4.

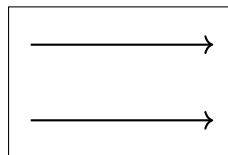


Figure 4: Parallel node of a behavior tree.

### Decorator

Decorator nodes are unary composite nodes that modify the control flow or data of their single child. They wrap around another node and adjust its return status according to user-defined rules. There are many types of decorator nodes but the common examples include:

- **Inverter:** Flips the return status of its child, returning **Success** if the child fails and **Failure** if the child succeeds.
- **Succeder:** Always returns **Success**, regardless of the child's actual return status.
- **Repeat:** Acts like a for-loop by repeatedly triggering its child for a predetermined number of ticks. It increments an internal counter on each tick, succeeds (and resets the counter) when a set bound is reached, or fails (and resets the counter) if the child fails.

Decorator nodes are typically represented as a rhombus (or diamond) containing the symbol for the specific type of decorator; see Fig. 5.

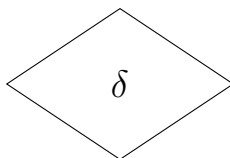


Figure 5: Condition node of a behavior tree. Label indicates the type of decorator.

### 2.3.2 Leaf Nodes

Leaf nodes represent the terminal elements of a behavior tree and serve as the interface between the tree's decision-making process and the environment. They are responsible for executing specific operations that directly affect the system or agent. Importantly, unlike composite or control flow nodes, the logic behind leaf nodes is not predefined; it is left for the user to implement. This means that it's up to the programmers to tailor the exact actions or conditions to their specific application needs.

### Action Nodes

An Action node executes a command when it receives a tick signal. This command is defined by the user and can be any arbitrary operation that affects the system. For instance, an action node might be programmed to initiate navigation to a certain location, make a certain gesture or process sensor data. The node actively monitors the progress of the command, as follows.



- **Execution:** Upon receiving a tick, the node triggers the user-defined command.
- **Status Returns:**
  - **Success:** Returned when the command completes successfully.
  - **Failure:** Returned if the command encounters errors or the desired outcome is not achieved.
  - **Running:** Returned while the command is still being executed, indicating that the operation is ongoing.
- **Representation:** Typically depicted as a box containing the label of the action; see Fig. 6.



Figure 6: Action node of a behavior tree. Label describes action performed.

### Condition Nodes

A Condition node evaluates a proposition when it receives a tick signal. The specific logic for this evaluation is defined by the user, allowing the condition to be customized according to the application's requirements. For example, a condition node might check if a person is present, or verify whether a variable meets a certain threshold.

- **Evaluation:** The node performs a user-defined check to determine whether the condition holds true.
- **Status Returns:**
  - **Success:** Returned if the condition is met.
  - **Failure:** Returned if the condition is not met.
- **No Running Status:** Unlike Action nodes, Condition nodes complete their check immediately and do not return a **Running** status.
- **Representation:** Typically depicted as a diamond containing the label of the condition; see Fig. 7.

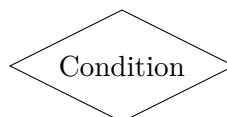


Figure 7: Condition node of a behavior tree. Label describes the condition to be checked.

## 2.4 Groot2

To implement the robot mission interpreter within the CSSR4Africa software ecosystem, which serves as the ROS node responsible for reading and executing behavior trees, we selected **BehaviorTree.CPP 4.6** as our library. The rationale behind this choice is detailed in **D5.4.3 Robot Mission Interpreter**. BehaviorTree.CPP is an open-source C++ library that facilitates the implementation, reading, and execution of behavior trees. These trees are defined using a domain-specific scripting language based on XML [1]. Although XML files can be edited with any text editor, BehaviorTree.CPP enhances usability by providing a dedicated graphical editor called Groot [2]. The edition of Groot bundled with BehaviorTree.CPP 4.6 is version 2, known as **Groot2** [7].

Groot2 is the official integrated development environment (IDE) for editing, monitoring, and interacting with behavior trees created using BehaviorTree.CPP. As a companion application to the library, it enables users to create, edit, and visualize behavior trees using a drag-and-drop interface. Additionally, it offers a real-time monitoring feature that connects to a running BehaviorTree.CPP executor, allowing users to observe the current state of the tree as it executes [7].

Unlike the BehaviorTree.CPP library itself, Groot2 is not open source. In fact, it follows a freemium business model. For behavior trees that extend beyond basic complexity, a licence is required to access the monitoring and debugging features essential for developing detailed mission specifications.

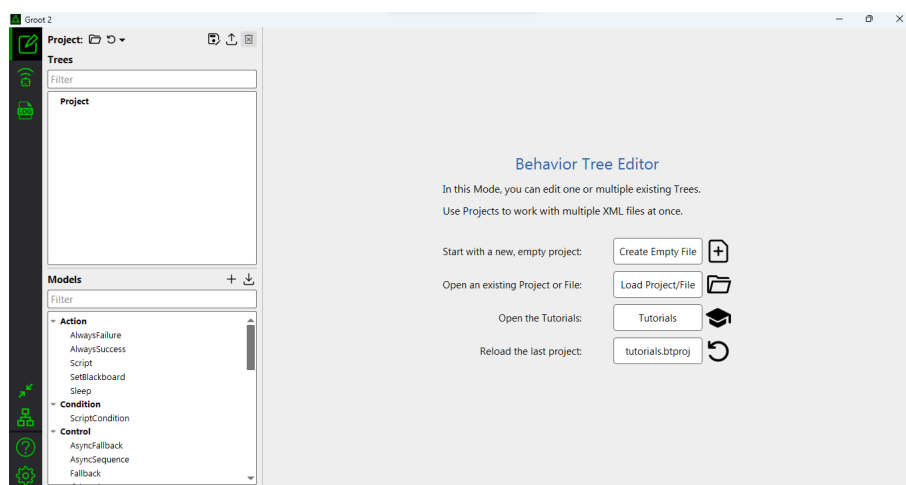


Figure 8: Groot2 IDE start screen

The Groot2 IDE (see Fig. 8) is an essential tool for developing robot mission specifications within the CSSR4Africa software ecosystem, as it streamlines the process of creating and modifying behavior trees.

To illustrate how Groot2 can be used to design a robot mission specification by implementing behavior trees, we can use a simple scenario example. In this example, a robot is tasked with picking and placing a ball. The mission is broken down into

individual tasks, such as locating the ball, picking it up, moving it, and finally placing it at a designated spot. The final behavior tree that will be recreated in Groot2 is shown in Fig. 9.

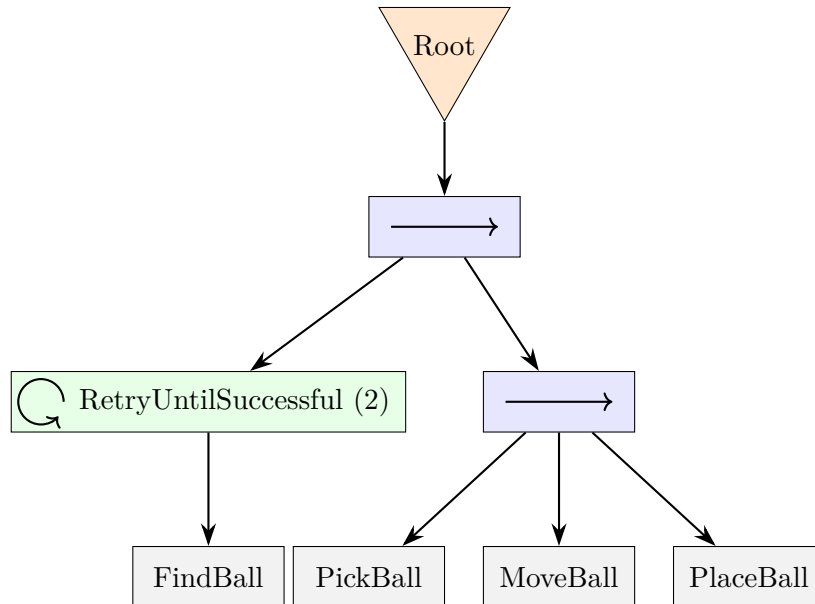


Figure 9: Behavior Tree for the Pick & Place a Ball robot mission

Under the root node, the behavior tree begins with a top-level **Sequence** node. This ensures that every step must succeed for the overall process to succeed. The first step involves a **RetryUntilSuccessful** decorator node that wraps the **FindBall** action node. This decorator attempts to locate the ball up to two times, allowing for a retry if the initial attempt fails; if the ball is not found after two attempts, the entire behavior tree fails. Once the ball is found, the tree proceeds to an inner **Sequence** node that orchestrates the subsequent actions. This node first executes the **PickBall** action to grasp the ball, then the **MoveBall** action node to transport it, and finally the **PlaceBall** action node to deposit it at the target location.

To begin, we open Groot2 and create a new tree file by clicking on the “Create Empty File” button. This will open a new window where we can design our behavior tree. The initial screen is shown in Fig. 10. At this instance, there are no mission nodes defined yet. Only the control-flow nodes that are pre-defined by the IDE are available. So, we need to define the three action nodes that we will use in our behavior tree. To that end, we click on the “+” button at the bottom left section, highlighted in red.

Once all the mission nodes are defined, they can be found under their specific category in the left panel, in this case “Action”. We can drag and drop these nodes onto the canvas to create the behavior tree. These nodes are displayed in blue in Figure 11.

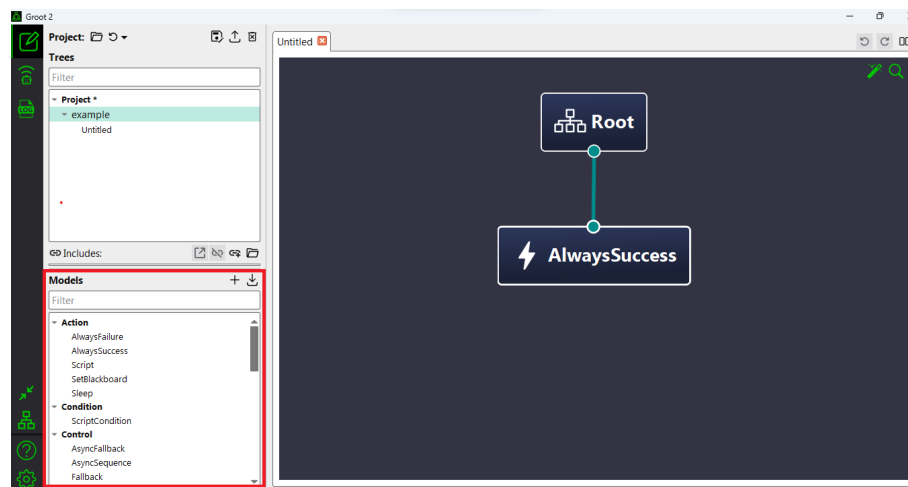


Figure 10: Groot2 New Tree Screen

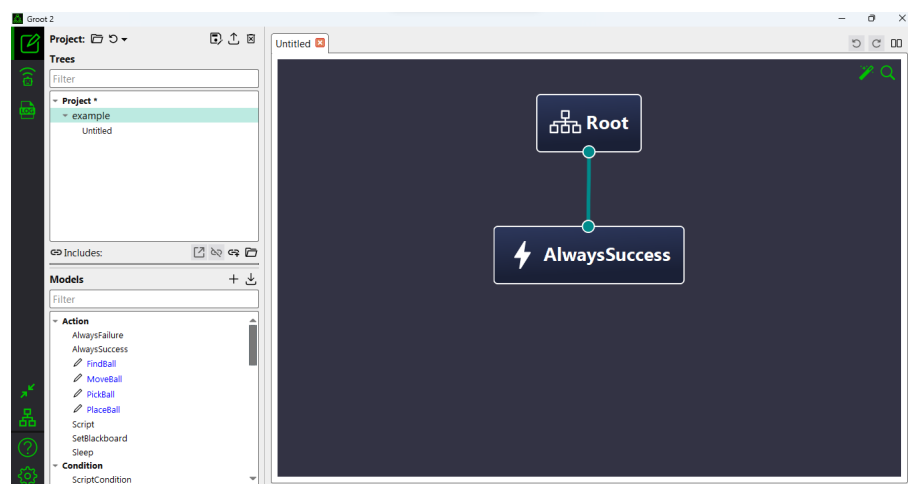


Figure 11: Groot2 Actions Nodes Defined

This will open a new window where we can define a new node; see Fig. 12. We select the type of node we want to create from the drop-down menu. In this case, we select the **Action** node type. We then provide a name for the node, in this case, **FindBall**. We repeat this process for each action node in our behavior tree.

Since our behavior tree starts with a **Sequence** node, we drag and drop a **Sequence** node from the left panel onto the canvas. Next, we add a **RetryUntilSuccessful** node by dragging it from the left panel and placing it under the **Sequence** node. Next, we add another **Sequence** node under the main **Sequence** node. We then add the **PickBall**, **MoveBall**, and **PlaceBall** action nodes under this inner **Sequence** node. The screen after adding these nodes is shown in Fig. 13.

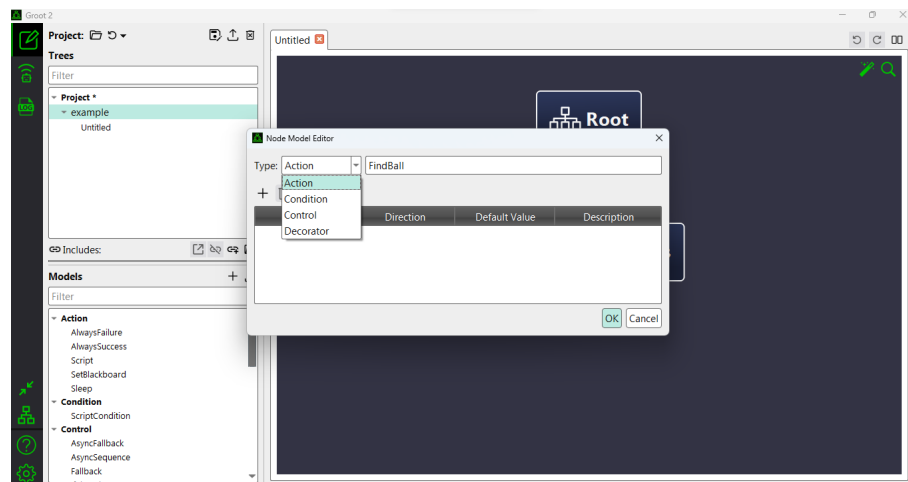


Figure 12: Groot2 New Node

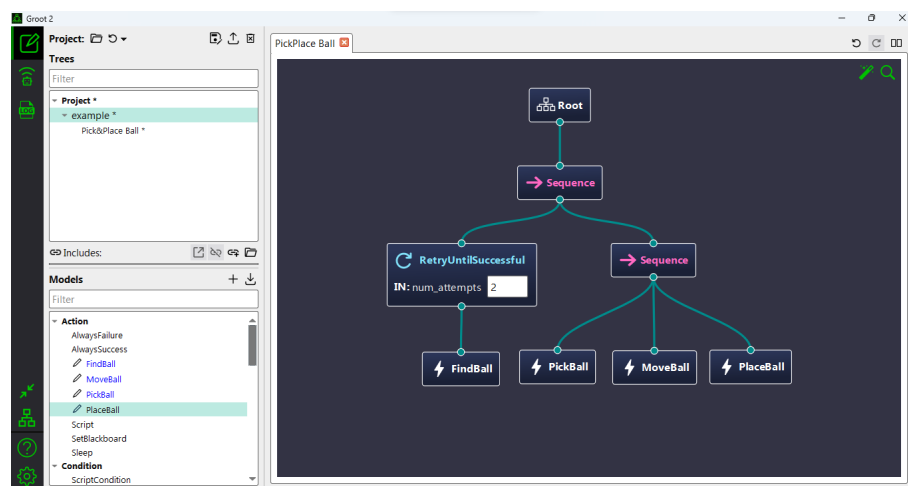


Figure 13: Groot2 Tree Complete

The behavior tree is now complete and ready for use. We can save the tree by clicking on the “Save” button at the top left corner of the screen. The tree can be exported in XML format by clicking on the “Export” button, represented by an arrow pointing upwards from a flat-line. The XML file can then be used in the robot mission interpreter to execute the specified mission.

### 3 Environment Knowledge Ontology and Knowledge Base

Figure 14 presents a simple ontology of environment knowledge. In this ontology, internal nodes in the ontology tree form the key in the environment knowledge base, e.g., `robotLocation`. Leaf nodes represent the data entities and their types. This allows multiple elements in a value for each key, e.g., `robotLocation 3 15.2 9.0 45.0`. The identification number value element associated with each key is the means by which the different elements an environment location — robot location, location description, gesture target, pre-gesture message, post-gesture message — are related. The tour specification identifies the number and sequence of locations to be visited in the tour.

Table 1 lists the key-value pairs, i.e., each key and the associated multiple numeric or alphanumeric elements of the value that encapsulate the environment knowledge. These numeric or alphanumeric values can then be used directly in the robot mission interpreter, i.e., the `behaviorController` ROS node, and passed as arguments in the service requests it issues to the nodes in the system architecture to conduct a tour or provide directions as a response to an enquiry at reception.

The key-value pairs are stored in a file `environmentKnowledgeBaseInput.dat`. This file is read and the value-pairs are accessed using a helper class `EnvironmentKnowledgeBase` described in Section 4.

Environment Knowledge		
Key	Values	Units
<code>robotLocationPose</code>	<code>&lt;IDNumber&gt; &lt;x&gt; &lt;y&gt; &lt;theta&gt;</code>	Metres, degrees
<code>robotLocationDescription</code>	<code>&lt;IDNumber&gt; &lt;text&gt;</code>	String
<code>gestureTarget</code>	<code>&lt;IDNumber&gt; &lt;x&gt; &lt;y&gt; &lt;z&gt;</code>	Metres
<code>preGestureMessageEnglish</code>	<code>&lt;IDNumber&gt; &lt;text&gt;</code>	String
<code>preGestureMessageIsizulu</code>	<code>&lt;IDNumber&gt; &lt;text&gt;</code>	String
<code>preGestureMessageKinyarwanda</code>	<code>&lt;IDNumber&gt; &lt;text&gt;</code>	String
<code>postGestureMessageEnglish</code>	<code>&lt;IDNumber&gt; &lt;text&gt;</code>	String
<code>postGestureMessageIsizulu</code>	<code>&lt;IDNumber&gt; &lt;text&gt;</code>	String
<code>postGestureMessageKinyarwanda</code>	<code>&lt;IDNumber&gt; &lt;text&gt;</code>	String
<code>culturalKnowledge</code>	<code>&lt;IDNumber&gt; &lt;Key1&gt;, &lt;Key2&gt;, ..., &lt;Keyn&gt;</code>	String
<code>tourSpecification</code>	<code>&lt;n&gt; &lt;ID1&gt;, &lt;ID2&gt;, ..., &lt;IDn&gt;</code>	

Table 1: Key-value pairs for specifying environment knowledge actions using the ontology depicted in Figure 14. As noted above, the identification number element of the value associated with each key is the means by which the robot location, the location description, the gesture target, the pre-gesture message, the post-gesture message, and the cultural knowledge keys (from the cultural knowledge ontology) are related. The tour specification identifies the number of locations and the sequence of locations to be visited in the tour.

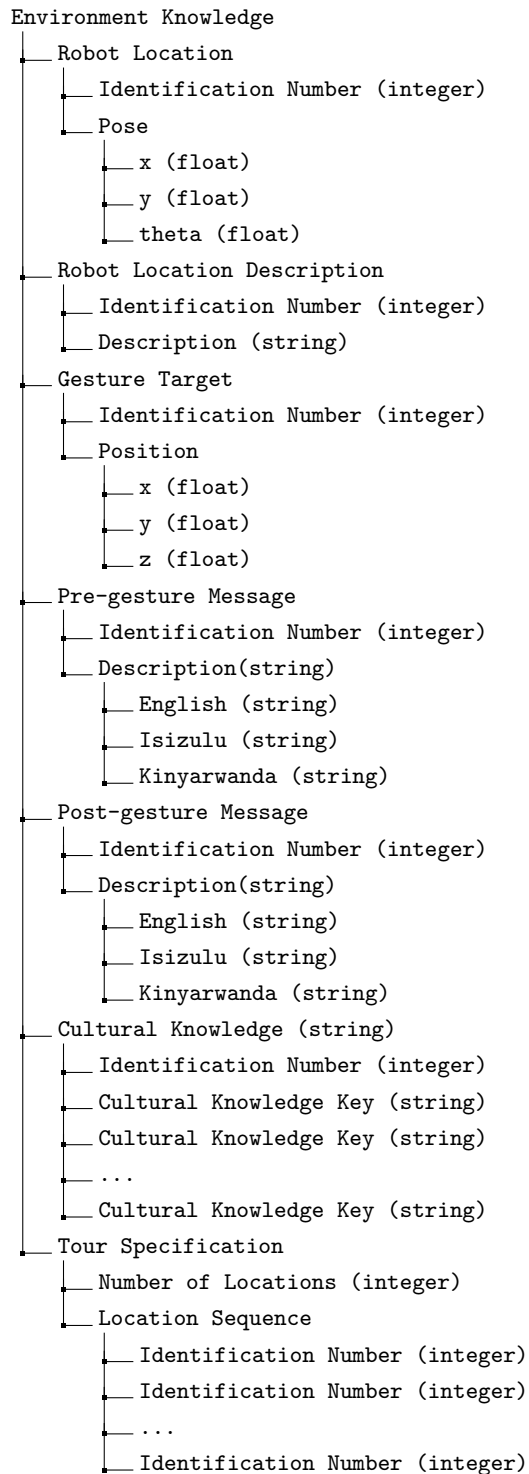


Figure 14: Environment knowledge ontology.

## 4 Environment Knowledge Base Implementation

The key-value pairs listed in Tables 1, comprising an alphanumeric key and associated numeric or symbolic values that encapsulate the environment knowledge, are stored in a file named

`environmentKnowledgeBaseInput.dat`. This file is accessed using a C++ helper class `EnvironmentKnowledgeBase` described in this section. Specifically, a C++ object instantiation of the helper class reads the environment knowledge base file, store the knowledge, and make the knowledge accessible through three public access methods. The remainder of this section details the implementation of this C++ helper class.

### 4.1 File Organization

Since the C++ helper class is intended to be embedded in `behaviorController` ROS node, it is not included as an individual component in the GitHub software repository. That said, the constituent files are organized in several subdirectories in a `utilities` package, as shown in Figure 15.

There are three C++ source code files: `environmentKnowledgeBaseApplication.cpp`, `environmentKnowledgeBaseImplementation.cpp`, and `environmentKnowledge.h`. The implementation file contains the helper class definition. The interface file contains the helper class declaration.

The application file is essentially a unit test to illustrate how the helper class is used and to verify that it works correctly. It instantiates a C++ helper class object which reads the environment knowledge base file, and uses the access method to retrieve values in the environment knowledge base, implemented using a binary search tree dictionary data structure, write them to the terminal.

It is intended that the implementation and interface files, along with the configuration and data files, be integrated in the `behaviorController` ROS node files. The relevant parts of the `behaviorController` software can use the application code as the basis of its implementation of functionality to access the knowledge base.

### 4.2 Configuration File

The population of the knowledge base is determined by the contents of a configuration file `environmentKnowledgeBase.ini` that contain a list of key-value pairs, as shown below in Table 2. The configuration file is named `environmentKnowledgeBaseConfiguration.ini`.

Table 2: Configuration file for the `EnvironmentKnowledgeBase` helper class.

Key	Value	Description
<code>knowledgeBase</code>	<code>environmentKnowledgeBaseInput.dat</code>	Specifies the filename of the file in which the cultural knowledge key-value pairs are stored.
<code>verboseMode</code>	<code>true</code> or <code>false</code>	Specifies whether diagnostic data is to be printed to the terminal.



```
utilities
├── config
│   └── environmentKnowledgeBaseConfiguration.ini
├── data
│   └── environmentKnowledgeBaseInput.dat
├── include
│   ├── utilities
│   └── environmentKnowledgeBaseInterface.h
├── src
│   ├── environmentKnowledgeBaseApplication.cpp
│   └── environmentKnowledgeBaseImplementation.cpp
├── launch
│   └── environmentKnowledgeBaseExample.launch
├── README.md
└── CMakeLists.txt
```

Figure 15: Directory Structure for the EnvironmentKnowledgeBase C++ helper class.

### 4.3 Environment Knowledge Base

The environment knowledge base file comprises a list of key-value pairs as shown in Table 3.

Table 3: Key-value pairs listed in the knowledge base file

environmentKnowledgeBaseInput.dat.

robotLocationDescription	1 Pepper's starting location
robotLocationPose	1 2.6 8.1 -90
gestureTarget	1 0.0 0.0 0.0
preGestureMessageEnglish	1 Welcome the the robotics lab at Carnegie Mellon University Africa
preGestureMessageIsiZulu	1 No message in isiZulu
preGestureMessageKinyarwanda	1 Murakazaneza muri laboratwari ya robotikisi muri kaneji meloni iniverisite yafurika.
postGestureMessageEnglish	1 I will give you a short tour. I hope you enjoy it
postGestureMessageIsiZulu	1 No message in isiZulu
postGestureMessageKinyarwanda	1 Ngiye kubatemberezaho gato, nizereko muribuze kuryoherwa
CulturalKnowledge	1 SymbolicShapeRespect BowExtentRespect
robotLocationDescription	2 The (other) Pepper robot
robotLocationPose	2 2.6 8.1 -45
gestureTarget	2 3.2 8.4 0.82
preGestureMessageEnglish	2 This is the Pepper humanoid robot
preGestureMessageIsiZulu	2 No message in isiZulu
preGestureMessageKinyarwanda	2 Iyi yitwa pepa humanoyidi roboti
postGestureMessageEnglish	2 We use it for research in social robotics and human-robot interaction
postGestureMessageIsiZulu	2 No message in isiZulu
postGestureMessageKinyarwanda	2 Tuyikoresha mubushakashatsi mubijyanye nimibanire hamwe nimikoranire hagati yabantu
CulturalKnowledge	2 ndetse naza robo 2 DeicticShape
robotLocationDescription	3 Lynxmotion
robotLocationPose	3 2.0 6.3 -45
gestureTarget	3 0.6 4.8 0.82
preGestureMessageEnglish	3 This is the Lynxmotion robot
preGestureMessageIsiZulu	3 No message in isiZulu
preGestureMessageKinyarwanda	3 Iyi robo ni likisi moshoni
postGestureMessageEnglish	3 We use it for teaching robot manipulation
postGestureMessageIsiZulu	3 No message in isiZulu
postGestureMessageKinyarwanda	3 Tuyikoresha mukwigisha roboti manipilesihoni
CulturalKnowledge	3 DeicticShape
robotLocationDescription	4 Roomba
robotLocationPose	4 5.0 3.9 110
gestureTarget	4 6.8 4.8 0.82
preGestureMessageEnglish	4 This is the Roomba
preGestureMessageIsiZulu	4 No message in isiZulu
preGestureMessageKinyarwanda	4 Iyiyo yitwa rumba
postGestureMessageEnglish	4 We use it for teaching mobile robotics
postGestureMessageIsiZulu	4 No message in isiZulu
postGestureMessageKinyarwanda	4 Tuyikoresha mukwigisha mobile robotikisi
CulturalKnowledge	4 DeicticShape
robotLocationDescription	5 Pepper's starting location
robotLocationPose	5 2.6 8.1 -90
gestureTarget	5 0.0 0.0 0.0
preGestureMessageEnglish	5 I hope you enjoyed the tour
preGestureMessageIsiZulu	5 No message in isiZulu
preGestureMessageKinyarwanda	5 Nizereko mwaryohewe no gutemberezwa
postGestureMessageEnglish	5 See you again soon
postGestureMessageIsiZulu	5 No message in isiZulu
postGestureMessageKinyarwanda	5 Tuzongere kubonana ubutaha
CulturalKnowledge	5 IconicShape
tourSpecification	5 1 4 3 2 5

### 4.4 Output Data File

There is no output data file for the environment knowledge base helper class.

## 4.5 Class Definition

Instantiating the `EnvironmentKnowledgeBase` class as a C++ object causes the contents of the environment knowledge base file to be read and stored in private dictionary data structure. Diagnostic messages are printed on the screen, depending on the value of `verboseMode` key in the configuration file. The contents of the dictionary are accessed using the identification number. Appendix A provides the full definition of the `EnvironmentKnowledgeBase` class.

### 4.5.1 Constructor

The `EnvironmentKnowledgeBase()` constructor reads the configuration file to determine the mode of operation, the name of the knowledge base value types file, and the name of the knowledge base file. It sets a private data member flag with the mode of operation, initializes the private dictionary data structure with the key-value pairs read from the knowledge base file. If operating in verbose mode, it echoes the keys and values to the terminal.

### 4.5.2 Destructor

The `~EnvironmentKnowledgeBase()` destructor deletes the dictionary data structure and write a diagnostic message if in verbose mode.

### 4.5.3 Private Data

The dictionary is implemented using a binary search tree with an element of type `struct KeyValue`, with eleven fields.

```
typedef struct {
    float x;
    float y;
    float theta;
} RobotLocationType;

typedef struct {
    int numberOfKeys;
    char *key[MAX_CULTURAL_KEYS];
} CulturalKnowledgeType;

typedef struct {
    float x;
    float y;
    float z;
} GestureTargetType;

typedef struct {
    int key; // location identification number
    RobotLocationType robotLocation;
    char robotLocationDescription[STRING_LENGTH];
    GestureTargetType gestureTarget;
    CulturalKnowledgeType culturalKnowledge;
    char preGestureMessageEnglish[STRING_LENGTH];
    char preGestureMessageIsiZulu[STRING_LENGTH];
    char preGestureMessageKinyarwanda[STRING_LENGTH];
    char postGestureMessageEnglish[STRING_LENGTH];
    char postGestureMessageIsiZulu[STRING_LENGTH];
    char postGestureMessageKinyarwanda[STRING_LENGTH];
} KeyValue;
```

The first field `key` is the identification number for this location. The data type is integer. This is the key that is used to access data in the binary search tree dictionary.

The second field `robotLocation` is a structure with three members containing the  $x$ ,  $y$ , and  $\theta$  floating point values that specify the pose of the robot at this location.

The third field `robotLocationDescription` is a description of this robot location. The data type is a C-string, i.e., a null-terminated array of characters.

The fourth field `gestureTarget` is a structure with three members containing the  $x$ ,  $y$ , and  $z$  floating point values that specify the position of the target to which the robot is to gesture.

The fifth field `culturalKnowledge` is a structure with two members, one specifying the number of keys in the cultural knowledge bases that are associated with this location, and the other listing them. These keys can then be used to look up the associated value in the culture knowledge base. At present, the data structure can store a maximum of ten keys. This can be adjusted by changing the value of the `#define MAX_CULTURAL_KEYS` constant.

The sixth, seventh, and eighth fields are messages to be spoken by the robot prior to executing the gesture; there are three versions, one in English, one in isiZulu, and one in Kinyarwanda. The data type is a C-string, i.e., a null-terminated array of characters.

The ninth, tenth, and eleventh fields are messages to be spoken by the robot after executing the gesture; again, there are three versions, one in English, one in isiZulu, and one in Kinyarwanda. The data type is a C-string, i.e., a null-terminated array of characters.

In addition to the binary search tree dictionary data structure, there is also a data structure

`tourSpecification` to specify the tour. This is a structure with two members: an integer specifying the number of robot locations in a tour and an array of integer identification numbers specifying sequence of robot locations that the robot should visit during the tour, in the order in which they are stored in the array.

```
typedef struct {
    int numberOfLocations;
    int locationIdNumber[MAX_NUMBER_OF_TOUR_LOCATIONS];
} TourSpecificationType;
```

There are also a small number of other private utility data fields to store the configuration filename, the configuration data, a `keyValue`, and the verbose mode flag.

#### 4.5.4 Public Access Methods

There are three public methods, one to print the knowledge base to the screen, one to retrieve a key-value pair, given the identification number of the location, and one to retrieve the tour specification. These are `printToScreen()`, `getValue()`, and `getTour()`, respectively.

The `printToScreen()` method does not have any parameters.

The `getValue()` method has two parameters: a key and a value, as follows.

```
bool getValue(int idNumber, KeyValue type *keyValue);
```

The method returns `true` if the key value was successfully retrieved from the knowledge base, `false` otherwise.

The `getTour()` method has one parameter: the tour data, as follows.

```
bool getTour(struct TourSpecificationType *tourSpecification);
```

The method returns `true` if the tour was successfully retrieved from the knowledge base, `false` otherwise.

## 5 Example Application

The example application in `environmentKnowledgeBaseApplication.cpp` illustrates the use of the class to read the environment knowledge base file and print each key-value pair with multiple elements. It also provides examples of how to retrieve the values associated with a robot location given by its identification number, and how to retrieve the sequence of robot locations in a tour.

```
#include <utilities/environmentKnowledgeBaseInterface.h>

int main() {
    KeyValueType      keyValue; // structure with key and values
    TourSpecificationType tour;  // list of tour locations
    int               idNumber; // location id
    int               i;        // counter
    int               k;        // counter

    /* instantiate the environment knowledge base object */
    /* this reads the knowledge value types file and the knowledge base file */
    /* as specified in the environmentKnowledgeBaseConfiguration.ini file */

    EnvironmentKnowledgeBase knowledgebase;

    /* verify that the knowledge base was read correctly */

    printf("main: the environment knowledge base data:\n");
    printf("-----\n\n");

    knowledgebase.printToScreen();

    printf("main: the environment knowledge base tour:\n");
    printf("-----\n\n");

    knowledgebase.getTour(&tour);

    /* query the contents of the knowledge base: */
    /* retrieve all the locations on a tour */
    /* and print them in the order in which they are specified */

    for (i = 0; i <= tour.numberOfLocations; i++) {
        idNumber = tour.locationIdNumber[i];
        if (knowledgebase.getValue(idNumber, &keyValue) == true) {
            printf("main:\n"
                "Key %-4d \n"
                "Location Description %s \n"
                "Robot Location (%.1f, %.1f %.1f)\n"
                "Gesture Target (%.1f, %.1f %.1f) \n"
                "Pre-Gesture Message English %s \n"
                "Pre-Gesture Message isiZulu %s \n"
                "Pre-Gesture Message Kinyarwanda %s \n"
                "Post-Gesture Message %s \n"
                "Post-Gesture Message isiZulu %s \n"
                "Post-Gesture Message Kinyarwanda %s \n",
                keyValue.key,
                keyValue.robotLocationDescription,
                keyValue.robotLocation.x, keyValue.robotLocation.y, keyValue.robotLocation.theta,
                keyValue.gestureTarget.x, keyValue.gestureTarget.y, keyValue.gestureTarget.z,
                keyValue.preGestureMessageEnglish,
                keyValue.preGestureMessageIsiZulu,
                keyValue.preGestureMessageKinyarwanda,
                keyValue.postGestureMessageEnglish,
                keyValue.postGestureMessageIsiZulu,
                keyValue.postGestureMessageKinyarwanda);

            printf("Cultural Knowledge ");
            for (k=0; k<keyValue.culturalKnowledge.numberOfKeys; k++) {
                printf("%s ", keyValue.culturalKnowledge.key[k]);
            }

            printf("\n\n");
        }
    }
}
```

Run the application by entering the following command:

```
roslaunch utilities environmentKnowledgeBaseExample
```

This assumes the existence of a `utilities` package, as shown in Figure 15, and that the package has been built with `catkin_make`.

Screenshots of the output of running this application are shown in Figures 16 and 17.

```

livernongubuntu:~/workspace/ros/src/utilities/src$ rosrn utilities environmentKnowledgeBaseExample
main: the environment knowledge base data:
-----
Key
1
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
2
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
3
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
4
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
5
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda

```

Figure 16: Screenshot of the output of running the example application: invoking `printToScreen()`.

```

main: the environment knowledge base tour:
-----
main:
Key
1
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
main:
Key
4
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
main:
Key
3
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
main:
Key
2
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda
main:
Key
5
Location Description
Robot Location
Gesture Target
Pre-Gesture Message English
Pre-Gesture Message isiZulu
Pre-Gesture Message Kinyarwanda
Post-Gesture Message English
Post-Gesture Message isiZulu
Post-Gesture Message Kinyarwanda

```

Figure 17: Screenshot of the output of running the example application: invoking `getTour()` and invoking `getValue()` successively for each location on the tour.



## Appendix A The EnvironmentKnowledgeBase Class

Note: documentation comments for the private methods have been removed due to space constraints.

```
#define NUMBER_OF_CONFIGURATION_KEYS 2
#define NUMBER_OF_VALUE_KEYS 11
#define MAX_NUMBER_OF_TOUR_LOCATIONS 20
#define MAX_CULTURAL_KEYS 10

typedef char Keyword[KEY_LENGTH];

typedef struct {
    char knowledgeBase[MAX_FILENAME_LENGTH];
    bool verboseMode;
} ConfigurationDataType;

typedef struct {
    float x;
    float y;
    float theta;
} RobotLocationType;

typedef struct {
    int numberOfKeys;
    char *key[MAX_CULTURAL_KEYS];
} CulturalKnowledgeType;

typedef struct {
    float x;
    float y;
    float z;
} GestureTargetType;

typedef struct {
    int numberOfLocations;
    int locationIdNumber[MAX_NUMBER_OF_TOUR_LOCATIONS];
} TourSpecificationType;

typedef struct {
    int key; // i.e., idNumber
    RobotLocationType robotLocation;
    char robotLocationDescription[STRING_LENGTH];
    GestureTargetType gestureTarget;
    CulturalKnowledgeType culturalKnowledge;
    char preGestureMessageEnglish[STRING_LENGTH];
    char preGestureMessageIsiZulu[STRING_LENGTH];
    char preGestureMessageKinyarwanda[STRING_LENGTH];
    char postGestureMessageEnglish[STRING_LENGTH];
    char postGestureMessageIsiZulu[STRING_LENGTH];
    char postGestureMessageKinyarwanda[STRING_LENGTH];
} KeyValueType;

typedef struct node *NodeType;

typedef struct node {
    KeyValueType keyValue;
    NodeType left, right;
} Node;

typedef NodeType BinaryTreeType;

typedef BinaryTreeType WindowType;

class EnvironmentKnowledgeBase {
public:
    EnvironmentKnowledgeBase();
    ~EnvironmentKnowledgeBase();

    bool getValue(int key, KeyValueType *keyValue);
    bool getTour(TourSpecificationType *tour);
    void printToScreen();

private:
    BinaryTreeType tree = NULL;
    TourSpecificationType tourSpecification;
    KeyValueType keyValue;
    ConfigurationDataType configurationData;
    char configuration_filename[MAX_STRING_LENGTH] = "environmentKnowledgeBaseConfiguration.ini";
};
```

```
BinaryTreeType *delete_element(KeyValueType keyValue, BinaryTreeType *tree);
KeyValueType delete_min(BinaryTreeType *tree);
bool get_value(int key, KeyValueType *keyValue, BinaryTreeType *tree);
void initialize(BinaryTreeType *tree);
int inorder_print_to_file(BinaryTreeType tree, int n, FILE *fp_out);
int inorder_print_to_screen(BinaryTreeType tree, int n);
BinaryTreeType *insert(KeyValueType keyValue, BinaryTreeType *tree, bool update);
int postorder_delete_nodes(BinaryTreeType tree);
int print_to_file(FILE *fp_out);
int print_to_file(BinaryTreeType tree, FILE *fp_out);
int print_to_screen(BinaryTreeType tree);
void read_configuration_data();
void read_knowledge_base();
};
```

## References

- [1] R. Ghzouli, T. Berger, E. B. Johnsen, A. Wasowski, and S. Dragul. Behavior trees and state machines in robotics applications. *IEEE Transactions on Software Engineering*, 49(9):4243 – 4267, 2023.
- [2] E. Dortmans and T. Punter. Behavior trees for smart robots practical guidelines for robot software development. *Journal of Robotics*, 2022.
- [3] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wąsowski. Behavior trees in action: a study of robotics applications. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2020)*, pages 196–209, New York, NY, USA, 2020. Association for Computing Machinery.
- [4] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith. A survey of behavior trees in robotics and AI. *arXiv:2005.05842*, 2020.
- [5] Unreal Engine. Behavior Tree Overview. <https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-tree-in-unreal-engine---overview>.
- [6] D. Faconti. Comment on issue #370: “listening to 3–4 topics parallely as SimpleConditions”. <https://github.com/BehaviorTree/BehaviorTree.CPP/issues/370>.
- [7] BehaviorTree.CPP 4.6. <https://www.behaviortree.dev>.

## Principal Contributors

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

Clifford Oyononka, Carnegie Mellon University Africa.

Tsegazeab Tefferi, Carnegie Mellon University Africa.

David Vernon, Carnegie Mellon University Africa.

## Document History

### Version 1.0

Partial version to address the specification of the environment knowledge base and `EnvironmentKnowledgeBase` helper class.

David Vernon.

5 March 2025.

### Version 1.1

Removed the `utilities` sub-directory since the C++ helper classes will be integrated directly in the software for the `behaviorController` node and not be made available independently.

Extended the ontology and added new keys to allow for pre- and post-gesture messages in English, isiZulu, and Kinyarwanda.

David Vernon.

11 March 2025.

### Version 1.2

Updated the ontology to rectify the duplicate robot location internal nodes, changing the second to robot location description.

Changed the `robotLocation` key to `robotLocationPose` in the list of keys and in the example data file.

David Vernon.

13 March 2025.

### Version 1.3

Updated the ontology to add a `culturalKnowledge` key. The value has multiple elements, each being a key in the cultural ontology. Changed the implementation of the C++ helper class and the examples accordingly.

Added Kinyarwanda versions of the pre- and post-gesture descriptions.

David Vernon.

20 March 2025.

### Version 1.4

Added Sections ?? and ?? which discuss the necessary background information and the implementation of the robot mission specification.

Updated the *Introduction* section to include a revised and comprehensive overview of the document.

Tsegazeab Tefferi.

22 March 2025.

**Version 1.5**

Changed environment knowledge base keys to camel case.

David Vernon.

15 April 2025.

**Version 1.6**

Removed Section 3 Implementation. This material has been transferred to Deliverable D6.1 Use Case Implementation. The introduction has been adjusted accordingly to remove reference to this material.

Fixed formatting issues, placement of figures, and typos.

Fixed references.

David Vernon.

18 April 2025.