



**BT-Enabled Cognitive Architecture:  
Internal Affect Sensing-Processing-Acting Cycle/Subsystem**

Rui Duro

UMinho | 2023



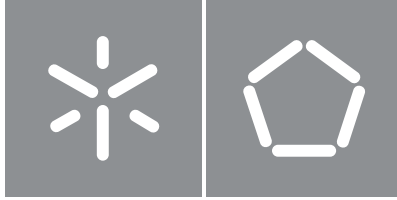
**Universidade do Minho**  
Escola de Engenharia

Rui Miguel Barbosa Pereira Duro

**BT-Enabled Cognitive Architecture:  
Internal Affect Sensing-Processing-Acting  
Cycle/Subsystem**

julho de 2023





**Universidade do Minho**

Escola de Engenharia

Rui Miguel Barbosa Pereira Duro

**BT-Enabled Cognitive Architecture:  
Internal Affect Sensing-Processing-Acting  
Cycle/Subsystem**

Dissertação de Mestrado  
Engenharia Eletrónica Industrial e Computadores  
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do

**Professor Doutor Adriano José da Conceição Tavares**

## **DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS**

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.



**Atribuição-NãoComercial-Compartilhual**  
**CC BY-NC-SA**

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

# Acknowledgements

I would like to take a moment to express my heartfelt appreciation to all those who have supported me throughout this journey. There are many people I am grateful for the invaluable support and encouragement I have received, and to them I would like to express my gratitude.

To Professor Adriano Tavares, my professor and advisor, my deepest gratitude for his invaluable guidance, knowledge, and availability throughout this dissertation. His expertise and insightful advice were pivotal in shaping the direction of this and future work.

I would also like to extend my heartfelt appreciation to Professor Fausto Giunchiglia for the incredible opportunities to work at the DISI in Trento. I am very grateful for the knowledge I have gained through our collaborations.

A special thanks to João Silva, with whom I shared many challenges and projects, including this one that led to the successful completion of our dissertations. I am grateful for all the help and support and look forward to working with you again as we pursue our PhDs.

To my classmates and friends, Master Luís Cunha and Heitor Silva, thanks for the availability, knowledge, and assistance since the first years. I truly appreciate the support and help you have provided throughout our academic journey together.

I am grateful to all the members of Casinha do Povo: the HHBs, Besugas and Tchinelos. Thank you for leading me astray and unnecessarily making my already difficult path even more challenging.

Lastly, but certainly not least, I want to express my deepest gratitude to my parents and my sister. Their unwavering support, unconditional love, and constant presence have been my pillars of strength throughout this entire journey.

## **STATEMENT OF INTEGRITY**

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

# Abstract

## **BT-Enabled Cognitive Architecture: Internal Affect Sensing-Processing-Acting Cycle/Subsystem**

Emotions provide a valuation system that assigns positive or negative affective states to different options, assisting in the assessment of risks and rewards. This emotional component allows cognitive architectures (CAs) to align the choices of an agent with its goals and well-being. Building upon this understanding, this dissertation extends a CA by integrating the internal affect subsystem. This subsystem is structured using Behavior Trees (BTs), taking advantage of the high degree of flexibility and the inherent ease with which BTs enable a modular expansion and adaptation of the CA.

A custom BT engine is leveraged in order to ensure compatibility with other subsystems of the CA and optimize resource usage for an embedded environment. Given that the hierarchical structure of XML is highly suitable for representing BTs, the custom engine is expanded to accept BTs in XML format as input. Furthermore, in order to provide an efficient communication structure between the nodes within the engine, a communication mechanism is incorporated into the engine, the Blackboard.

The internal affect subsystem adheres to the Sensing-Processing-Action (SPA) model of the brain and proposes a simplified model of emotions along with their corresponding memory manipulation processes. In order to enable the subsystem to make contextually appropriate decisions and generate intentions based on the current emotional state, a practical application was devised. This application is focused on guiding an agent from its present location to a desired destination. Within this context, the subsystem aligns the logical choices of the agent with its well-being, prioritizing routes that satisfy its needs. Additionally, the desired location is based on the current drive, which itself is determined by the current emotional state.

The subsystem underwent several validation tests as part of the fully integrated CA. During these tests, a prototype and a controlled environment were developed and employed to validate each memory manipulation process and functionality of the subsystem, ensuring the successful achievement of the goal.

**Keywords:** Emotions, Cognitive architecture, Behavior Trees, XML, Sensing-Processing-Action (SPA) model

# Resumo

## **Arquitetura Cognitiva Ativada por Árvores de Comportamento: Subsistema Interno Afetivo**

As emoções fornecem um sistema de avaliação que atribui estados afetivos positivos ou negativos a diferentes opções, que auxiliam na avaliação de riscos e recompensas. Esta componente emocional permite que arquiteturas cognitivas alinhem as escolhas de um agente com os seus objetivos e bem-estar. Com base nisto, esta dissertação estende uma arquitetura cognitiva ao integrar o subsistema interno afetivo. Este subsistema é estruturado em árvores de comportamento, tirando partido do alto grau de flexibilidade e a facilidade inerente com que as árvores de comportamento facilitam a extensão modular e adaptação da arquitetura cognitiva.

Um motor de árvores de comportamento personalizado foi utilizado para garantir a compatibilidade com outros subsistemas da arquitetura cognitiva e otimizar o uso de recursos para um ambiente embebido. Dado que a estrutura hierárquica do formato XML é adequada para representar árvores de comportamento, o motor personalizado é expandido de forma a aceitar árvores de comportamento no formato XML como entrada. Além disso, de modo a fornecer um meio de comunicação eficiente entre nós, é desenvolvida e adicionada ao motor uma *Blackboard*.

O subsistema interno afetivo baseia-se no modelo do cérebro Sensing-Processing-Action (SPA) e propõe um modelo de emoções simplificado juntamente com os respetivos processos de manipulação de memória. Para permitir que o subsistema tome decisões contextualmente apropriadas e gere intenções com base no estado emocional atual, foi criada uma aplicação prática. Esta aplicação foca-se em guiar um agente da sua localização atual para um destino desejado, selecionado pela motivação atual, que por sua vez é determinada pelo estado emocional atual. Dentro deste contexto, o subsistema alinha as decisões lógicas do agente com o seu bem-estar, priorizando rotas que satisfaçam essa condição.

O subsistema foi submetido a vários testes de validação como parte da arquitetura cognitiva totalmente integrada, de forma a garantir que o objetivo final fosse alcançado. Para os testes foi desenvolvido e utilizado um protótipo num ambiente controlado para validar cada processo de manipulação de memória e funcionalidade do subsistema.

**Palavras-chave:** Emoções, arquiteturas cognitivas, árvores de comportamento, XML, modelo Sensing-Processing-Action (SPA)



# Contents

- I - Introduction ..... 1**
  - 1. Main Goal ..... 1
  - 2. Document Structure ..... 2
  
- II - Background and State of the Art ..... 3**
  - 1. Behavior Trees ..... 3
    - 1.1. Classical Formulation ..... 3
    - 1.2. Similar Technologies ..... 6
  - 2. Related Work ..... 8
    - 2.1. Cognitive Architectures ..... 8
    - 2.2. Modeling Emotion in Cognitive Architectures ..... 11
  
- III - Design ..... 15**
  - 1. Behavior Tree Engine ..... 15
    - 1.1. BehaviorTree.CPP ..... 15
    - 1.2. Behavior Tree Custom Engine ..... 19
  - 2. Cognitive Architecture ..... 23
    - 2.1. Use Case ..... 24
    - 2.2. Memory Model ..... 25
    - 2.3. Subsystems ..... 32
    - 2.4. Physical World Subsystem ..... 33
    - 2.5. Internal Affect Subsystem ..... 34
  
- IV - Implementation ..... 41**
  - 1. Behavior Tree Custom Engine ..... 41

1.1. XML Parser .....	42
1.2. Blackboard .....	47
2. Cognitive Architecture .....	51
2.1. Memory Stores .....	51
2.2. Internal Affect Subsystem .....	60
<b>V - Validation and Results .....</b>	<b>72</b>
1. Physical Structure .....	72
2. Memory Manipulation Processes .....	73
3. Internal Affect Subsystem .....	77
4. Discussion .....	79
<b>VI - Conclusions and Future Work .....</b>	<b>80</b>
<b>Appendix A .....</b>	<b>87</b>
<b>Appendix B .....</b>	<b>92</b>
<b>Appendix C .....</b>	<b>93</b>

# List of Figures

Figure 1: Root node representation. ....	4
Figure 2: Fallback node representation.....	4
Figure 3: Sequence node representation. ....	5
Figure 4: Parallel node representation.....	5
Figure 5: Decorator node representation. ....	5
Figure 6: Illustration of one-way and two-way control transfers. ....	7
Figure 7: Processing cycle of Soar [17]. ....	9
Figure 8: Earlier Structure of Soar [17].....	9
Figure 9: Extended Structure of Soar [17]. ....	9
Figure 10: Structure of ACT-R 7 [19] .....	10
Figure 11: Structure of CLARION [23] .....	11
Figure 12: Lövheim's, Geneva and Plutchik's emotional models, respectively.....	12
Figure 13: NEUCOGAR's mapping of Lovheim's cube of emotions to computing parameters as in [2].	14
Figure 14: Side-by-side comparison of BT and XML representations.....	16
Figure 15: Subtree representation in both BT and XML representations. ....	17
Figure 16: Flowchart describing the parsing algorithm in BehaviorTree.CPP.....	17
Figure 17: Illustration of the Blackboard functionality, adapted from [40]. ....	18
Figure 18: Illustration of port remapping functionality, adapted from [40]. ....	19
Figure 19: Tree structure before (left) and after (right) the removal of the node vector. ....	20
Figure 20: PugiXML's representation of Figure 15 example.....	21
Figure 21: Flowchart describing the parsing algorithm in Behavior Tree Custom Engine. ....	22
Figure 22: R/W Mutex locking mechanism. ....	23

Figure 23: Use case environment.....	25
Figure 24: Atkinson-Shiffrin memory model (left) and designed memory model (right).....	26
Figure 25: Insertion of an element in the STM.....	27
Figure 26: LTM graph structure. ....	27
Figure 27: Sigmoid function to determine the probability of retrieval. ....	28
Figure 28: eSTM structure - mapping elements in eSTM to elements in STM. ....	28
Figure 29: eLTM structure - mapping memory elements (left) to emotional state (right).....	29
Figure 30: Sigmoid function illustrating the relationship between the decay and the emotional score..	29
Figure 31: Flowchart illustrating the processes new information is subjected to.....	31
Figure 32: Flowchart illustrating the relationships between rehearsal, consolidation, and transfer.....	31
Figure 33: Designed cognitive architecture. ....	32
Figure 34: Base cognitive architecture structured using BTs. ....	33
Figure 35: Final physical world subsystem structured using BTs.....	34
Figure 36: Concentrations of cortisol and serotonin mapped to emotions.....	35
Figure 37: Weight distribution if the most recent memory element is index 0 (left) or 3 (right). ....	36
Figure 38: Action nodes added in the sensorial component of the cognitive architecture. ....	37
Figure 39: Additions made to the behavior scheduler of the cognitive architecture. ....	38
Figure 40: Emotional decision added to the cognitive architecture. ....	39
Figure 41: Flowchart illustrating the emotional decision process. ....	40
Figure 42: XMLParser class UML representation. ....	42
Figure 43: Blackboard class UML representation.....	48
Figure 44: e_memory_t UML struct representation. ....	52
Figure 45: eSTM UML class representation. ....	52
Figure 46: eLTM UML class representation. ....	56
Figure 47: Environment used in the validation tests.....	73

Figure 48: Prototype used in the validation tests.....	73
Figure 49: Testing itinerary to validate memory manipulation processes. ....	74
Figure 50: Rehearsal validation.....	74
Figure 51: Association validation.....	75
Figure 52: Consolidation validation. ....	75
Figure 53: Transfer validation. ....	76
Figure 54: Decay validation.....	77
Figure 55: Testing scenario and itinerary to validate the influence in decision-making.....	77
Figure 56: Influence in decision-making validation.....	78
Figure 57: Drive selection validation.....	78
Figure 58: Loud noise detection and avoid danger drive validations. ....	79

# List of Tables

Table 1: BT node types ..... 6

# List of Listings

Listing 1: Verification on the number of arguments.....	41
Listing 2: Scoped block to ensure the destruction of the parser object. ....	42
Listing 3: XMLParser – <i>loadTree()</i> method.....	43
Listing 4: <i>checkFile()</i> function. ....	43
Listing 5: XMLParser – <i>getMainTreeNode()</i> method.....	44
Listing 6: XMLParser – <i>nodeliterator()</i> method. ....	45
Listing 7: XMLParser – <i>parseXMLNode()</i> method.....	46
Listing 8: <i>hash()</i> function. ....	47
Listing 9: Overload of the UDL operator <code>""_</code> . ....	47
Listing 10: Blackboard – <i>getInstance()</i> function.....	49
Listing 11: Alias for mutex and locking mechanisms.....	49
Listing 12: Blackboard – <i>checkEntry()</i> method. ....	50
Listing 13: Blackboard – <i>setEntry()</i> method.....	50
Listing 14: Blackboard – <i>getEntry()</i> method.....	51
Listing 15: Cast of <code>std::any</code> type to its original type. ....	51
Listing 16: eSTM – <i>isIneSTM()</i> method.....	53
Listing 17: eSTM – <i>rehearsal()</i> method. ....	54
Listing 18: eSTM – <i>insertAsRecent()</i> method.....	54
Listing 19: eSTM – <i>getAverage()</i> method.....	55
Listing 20: eSTM – <i>getMostRecent()</i> method. ....	56
Listing 21: eLTM – <i>save()</i> method.....	57
Listing 22: eLTM – <i>load()</i> method. ....	57

Listing 23: eLTM – <i>transfer()</i> method.....	57
Listing 24: eLTM – <i>association()</i> method.....	58
Listing 25: eLTM – <i>consolidation()</i> method.....	58
Listing 26: eLTM – <i>decay()</i> method.....	58
Listing 27: eLTM - <i>checkEntry()</i> method. ....	59
Listing 28: eLTM – <i>getEScore()</i> method. ....	59
Listing 29: eLTM – <i>getHighestESRef()</i> method.....	60
Listing 30: Setup ADC slave I <sup>2</sup> C address for light and decibel sensors. ....	61
Listing 31: ADC configuration for luminosity sensor. ....	62
Listing 32: ADC configuration for decibel sensor.....	62
Listing 33: Read converted value from the ADC and compute hormone concentration.....	63
Listing 34: Store hormone concentration as cortisol in Blackboard.....	63
Listing 35: Store hormone concentration as serotonin in Blackboard. ....	63
Listing 36: Signal and signal handler configuration. ....	64
Listing 37: <i>execute()</i> method of Emotionally stable condition node.....	65
Listing 38: <i>taskFunction()</i> method of Drive selection action node. ....	66
Listing 39: Modification made to the method <i>findShortestPath()</i> . ....	67
Listing 40: Constructor of the Self sensing action node.....	67
Listing 41: Mood and emotional score computation.....	68
Listing 42: First section of the <i>updateMemory()</i> function.....	68
Listing 43: Second section of the <i>updateMemory()</i> function. ....	69
Listing 44: Retrieval of possible paths from the Blackboard. ....	69
Listing 45: Determining of the best path from the vector of possible paths. ....	70
Listing 46: Final decision on the best path and scheduling of the next action to take. ....	71



# Abbreviations, Acronyms, and Initialisms

<b>5-HT</b>	5-HydroxyTryptamine (Serotonin)
<b>ACS</b>	Action-Centered Subsystem
<b>ACT-R</b>	Adaptive Control of Thought-Rational
<b>ADC</b>	Analog-to-Digital Converter
<b>AI</b>	Artificial Intelligence
<b>BT</b>	Behavior Tree
<b>CA</b>	Cognitive Architecture
<b>CLARION</b>	Connectionist Learning with Adaptive Rule Induction ON-line
<b>DA</b>	Dopamine
<b>DR</b>	Data Rate
<b>eLTM</b>	Emotional Long-Term Memory
<b>eSTM</b>	Emotional Short-Term Memory
<b>FSM</b>	Finite-State Machine
<b>GEW</b>	Geneva Emotion Wheel
<b>HFSM</b>	Hierarchical Finite-State Machine
<b>I<sup>2</sup>C</b>	Inter-Integrated Circuit
<b>IDE</b>	Integrated Development Environment
<b>LDR</b>	Light Dependent Resistor
<b>LSB</b>	Least Significant Byte
<b>LTM</b>	Long-Term Memory
<b>MCS</b>	Meta-Cognitive Subsystem
<b>MS</b>	Motivational Subsystem
<b>MSB</b>	Most Significant Byte

<b>NACS</b>	Non-Action-Centered Subsystem
<b>NE</b>	Noradrenaline
<b>NEUCOGAR</b>	Neuromodulating Cognitive Architecture
<b>OCR</b>	Optical Character Recognition
<b>OS</b>	Operating System
<b>RAII</b>	Resource Acquisition is Initialization
<b>RAM</b>	Random-Access Memory
<b>SM</b>	Sensory Memory
<b>SPA</b>	Sensing-Processing-Acting
<b>SPS</b>	Samples-Per-Second
<b>SSH</b>	Secure Shell
<b>STM</b>	Short-Term Memory
<b>UDL</b>	User-Defined Literal
<b>XML</b>	Extensible Markup Language

# I - Introduction

Emotions play a fundamental role in cognitive architectures (CAs), providing them with invaluable capabilities and enhancing their overall functionality. Therefore, the modeling of emotions holds significant importance [1]. Although emotions are often associated with subjective experiences and human behavior, their significance extends beyond our species and is essential for both biological and artificial cognitive systems [2]. Emotions provide a valuation system that assigns positive or negative affective states to different options, assisting in the assessment of risks and rewards. This emotional component allows CAs to align the choices of an agent with its goals and well-being [3]. Additionally, emotions provide the motivational drive required for a goal-directed behavior. Positive emotions, such as joy and excitement, promote approach behaviors toward rewarding stimuli, while negative emotions, such as fear and disgust, elicit avoidance or protective responses [4]. By integrating emotional signals, CAs align the actions of the agent with its goals, increasing persistence and enhancing the likelihood of success.

Behavior Trees (BTs) have gained significant attention in the robotics community in the past decade as a powerful tool for modeling CAs [5]. Originally introduced in the game industry, BTs offer a structured approach for task switching in reactive and fault-tolerant systems [6]. Their modular nature has been particularly advantageous, leading to increased interest and application in various domains [7]. One of the key benefits of BTs is their ability to split a system into smaller, independently developed or reusable BTs, thereby saving valuable time and resources [8]. Additionally, their hierarchical organization, human readability, and reactivity further contribute to their effectiveness [7]. Hence, BTs are a viable approach for the development of CAs.

## 1. Main Goal

This dissertation aims to develop, integrate, and validate the internal affect subsystem within a CA structured in BTs. The goal of this subsystem is to enhance the decision-making capabilities of the CA by integrating emotions into the decision-making process and generating intentions based on the current emotional state of the agent.

The first step toward achieving this goal is to develop a simplified model for emotions within the CA. This model makes certain assumptions and simplifications to reduce the complexity involved in modeling emotions accurately. By simplifying the model, it becomes more feasible to integrate emotions into the

decision-making process and implement it within a BT.

Furthermore, the emotional memory structure and subsequent memory manipulation processes within the internal affect subsystem will adhere to the same model as the co-existing physical world subsystem of the CA. This consistency ensures easy integration with the existing subsystem and maintains coherence within the CA. For the same reason, the internal affect subsystem follows the Sensing-Processing-Acting (SPA) model of the brain. The final step is to integrate both subsystems and validate the subsystem within the context of a fully integrated CA.

## 2. Document Structure

The dissertation follows a well-structured organization based on the Waterfall methodology, with each chapter addressing specific aspects of the research conducted. The document structure is outlined as follows: Chapter II delves into the theoretical fundamentals related to the topics explored in the dissertation. It starts by introducing BTs, discussing their structure, characteristics, and comparing them to earlier technologies. The chapter then delves into the state of the art in CAs and emotion modeling within such architectures. This comprehensive overview lays the foundation for the development of the internal affect subsystem and provides a strong basis for the research conducted. Next, in Chapter III, the design of the adaptation of two core elements for the BT custom engine is presented, namely the Extensible Markup Language (XML) parser and the *Blackboard*. Additionally, the chapter covers the detailed design of the internal affect subsystem within the CA, outlining its core elements and their interactions with the physical world subsystem. Chapter IV focuses on discussing the implementation of the two adapted elements of the BT custom engine and the internal affect subsystem. Next, Chapter V centers around testing and validation procedures, evaluating the impact of the internal affect subsystem on decision-making and validating the memory manipulation processes and intention generation within the CA. Finally, Chapter VI offers conclusions versing on the work developed in this dissertation and identifies possible opportunities for future research and development.

# II - Background and State of the Art

In this chapter, theoretical fundamentals are provided with respect to the topics of interest in this dissertation. It first introduces Behavior Trees (BTs), discussing their structure, characteristics, and a comparison with earlier technologies. Subsequently, the state of the art in cognitive architectures (CAs) and emotion modeling within CAs is covered, serving as the foundation for the development of the internal affect subsystem in this dissertation. This comprehensive overview establishes the necessary context for the subsequent chapters, providing a strong basis for the research conducted.

## 1. Behavior Trees

First introduced in the game industry, BTs are a way to structure the switching between tasks for reactive and fault-tolerant systems [6]. They provide an efficient way to increase modularity in complex systems, a property that is crucial in many applications [7], which led to a growing interest in the robotics community in the last decade [9]. One of the many advantages derived from their high modularity is the ability to split the system into smaller BTs that are developed independently or reused from other BTs [8], which ultimately saves time and resources. Other advantages include their hierarchical organization, human readability, and reactivity [7].

### 1.1. Classical Formulation

The execution of a BT is tick-driven, which is a signal sent from the Root node at a chosen frequency that dictates whenever a node is executed. This signal propagates from the Root node through its children following the rules of different control flow nodes, and upon reaching a leaf node, it executes either an action or condition and returns one of three pre-defined statuses: *success*, *running*, or *failure* [10]. This returned status is then propagated back and forth through the tree until finally one of the return statuses reaches the Root node [11].

As previously mentioned, the nodes from the classical formulation can be divided into control flow nodes (Fallback, Sequence, Decorator, and Parallel) and leaf nodes (Action and Condition). Each node will now be discussed in detail and at the end of this section, a table is presented with all the node principles compiled, see Table 1.

### Root Node

The Root node is the first node in the BT and where the execution of the tree starts and ends. This node can only have one single child and is represented by the symbol “ $\emptyset$ ”, see Figure 1. It is responsible for ticking the tree at a given frequency and receiving the final return status at the end of the execution.



Figure 1: Root node representation.

### Fallback Node

The Fallback node is represented by the symbol “?”, see Figure 2, and can be interpreted as a logic OR function. It ticks its children from left to right until one of them returns either *success* or *running* and returns the received status to its parent. The only scenario where it returns *failure* to its parent is if all its children return *failure*. This node, upon receiving a *running* or *success* return status from one of its children, does not tick the subsequent children [7].

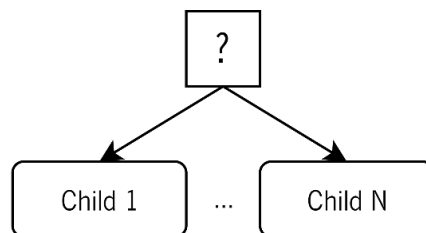


Figure 2: Fallback node representation.

### Sequence Node

The Sequence node is represented by an arrow, see Figure 3, and can be interpreted as a logic AND function as it only returns *success* if all its children return *success*. It ticks its children from left to right and returns *running* or *failure* whenever a child returns respectively *running* or *failure* [6]. Upon receiving the return status *failure* from a child, it keeps ticking all the children to the left of the one that failed but stops ticking the subsequent children.

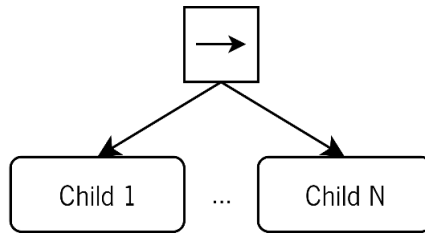


Figure 3: Sequence node representation.

### Parallel Node

The Parallel node is represented by two arrows, as shown in Figure 4, and it is rarely used due to underlying concurrency problems [12]. These problems come from the concurrent execution of its children as it ticks all of them at the same time. The node returns *success* only if the number of children returning *success* surpasses a user-defined threshold  $M$  and *failure* if at least  $N - M + 1$  children return *failure*, where  $N$  is the number of children and  $M \leq N$  is a user-defined threshold. The node returns *running* otherwise [7] [11].

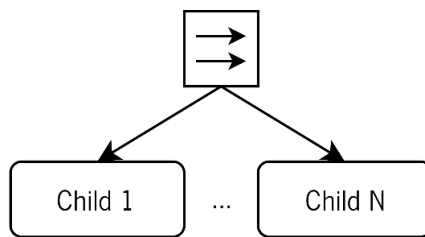


Figure 4: Parallel node representation.

### Decorator Node

The Decorator node, which is represented by a rhombus, see Figure 5, has a single child and it returns a status according to user-defined rules. This node is used to introduce additional semantics, change the return status of a node, or alter the ticking policy [6].

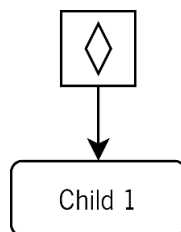


Figure 5: Decorator node representation.

## Action Node



The Action node when ticked performs some user-defined operations and returns *success* whenever the operations are completed successfully, *failure* if the operations cannot be completed and *running* while the operations are being performed. It is represented by a rectangle.

## Condition Node

The Condition node is a subset of the Action node represented by an oval, but it is given a separate category and graphical symbol to increase the readability of the BT. It checks for a user-defined proposition and returns *success* or *failure* depending on if the proposition holds or not [9], and never returns the status *running*.

To recapitulate, all the node types, their symbol, and return status conditions, are compiled into the following table (Table 1).

Table 1: BT node types.

<b>Node Type</b>	<b>Symbol</b>	<b>Returns Success</b>	<b>Returns Failure</b>	<b>Returns Running</b>
Root	$\emptyset$	If tree succeeds	If tree fails	While executing
Fallback	?	If one child succeeds	If all children fail	If one child returns running
Sequence	$\rightarrow$	If all children succeed	If one child fails	If one child returns running
Parallel	$\Rightarrow$	If $\geq M$ children succeed	If $N - M + 1$ children fail	Else
Decorator	$\diamond$	Varies	Varies	Varies
Condition		If true	If false	Never
Action		Upon completion	If impossible to complete	During completion

## 1.2. Similar Technologies

In this section, two well-known task-switching architectures, FSMs and HFSMs, are described and analyzed to better understand how they relate and compare to BTs, i.e., what are the advantages and disadvantages of these earlier technologies and how BTs solve some of the issues with them.

### Finite-State Machines (FSMs)

Finite-State Machines (FSMs) are a common structure to implement task switching in a fast, simple, and intuitive way, and for those reasons they have been the technology of choice for AI in games [8]. However,



FSMs do not scale well, therefore, it becomes difficult to develop complex systems with many transitions and, consecutively, difficult to maintain [13]. This problem results from the nature of the transitions in FSMs, which are one-way control transfers, i.e., the control is transferred from one state to another much like a *goto statement* in assembly. In BTs, however, the transitions are composed of two-way control transfers, i.e., the control after being transferred to the next node, its child, the control always returns to its parent node after it is done executing, as the *call/ret pair* in assembly, see Figure 6.

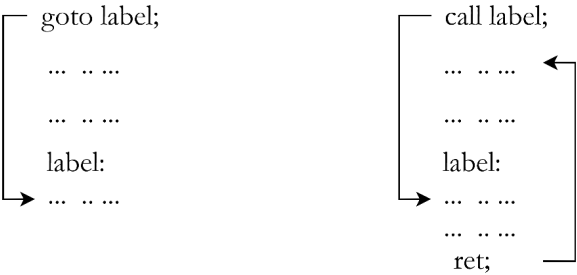


Figure 6: Illustration of one-way and two-way control transfers.

One-way control transfers, in complex systems such as reactive systems, result in an FSM with a high number of transitions, hence the decrease in modularity, since if one component is removed, every transition to that component needs to be revised. Maintainability, scalability, and reusability problems result from the lack of modularity in complex FSMs [7], which is crucial when developing complex systems. Modularity allows the system to be split into building blocks, that can be developed and tested separately, and then integrated into the main system, promoting the reuse of code, incremental design of functionalities, and efficient testing.

Hierarchical Finite-State Machines (HFMSs)

Created to alleviate some of FSMs' disadvantages and ensure extensibility, flexibility, and reusability [14], Hierarchical Finite-State Machines possess increased modularity, as it is possible to split tasks into sub-tasks, and fewer scalability problems due to behavior inheritance [8], which provides different levels of abstraction.

Although HFMSs fixed some critical issues with FSMs, problems in maintainability persisted, as it is still difficult to add or remove states. The newly added hierarchy also brought downsides given that it must be manually created and user-defined, which, in contrast to FSMs and BTs that are simple and intuitive, brings complexity to its development [7].

## 2. Related Work

In this section, the foundation for this dissertation is presented by introducing concepts such as cognitive architectures (CAs) and emotions, alongside some well-known CAs and relevant work on emotion research. As mentioned in the chapter's introduction, the three CAs introduced and discussed are: Soar, ACT-R, and CLARION. For each architecture, a brief description of the concept, its structure, and distinctive characteristics is given, considering its most recent version. Additionally, the effects of emotion on memory and cognition are discussed, and three strategies for emotion modelling are presented: Lövheim's Cube of Emotion, Geneva emotion wheel and Plutchik's Wheel of Emotions.

### 2.1. Cognitive Architectures

Cognitive architecture is a term widely used in modern cognitive science that refers to both a theory about the structure of the human mind and a computational instantiation of such a theory [15], or according to a definition by Anderson [16], is "a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the mind". It ultimately is a fixed structure that supports the acquisition and use of knowledge.

Cognitive architectures distinguish themselves as generally intelligent entities, by having the ability to solve not just a single problem using a specific method but execute a wide variety of tasks using knowledge acquired through experience, in complex and dynamic environments [17]. Their structure comprises memories to store knowledge, processing units that retrieve, select, combine, and store knowledge, and languages to represent the knowledge that is stored.

#### Soar

Soar, one of the best-known existing CAs, has existed since mid-1982. It is a problem-solving architecture, that, like many artificial intelligence (AI) systems, attempts to provide an appropriate organization for intelligent action [18]. At the lowest level, Soar's processing consists of matching and firing rules, which provide a flexible context-dependent representation of knowledge. Unlike most rule-based systems that choose a single rule to fire at a given time, which serves as the point of choice in the system, Soar allows additional knowledge to influence a decision by introducing operators as the point of choice in the system and using rules to propose, evaluate and apply operators [17].

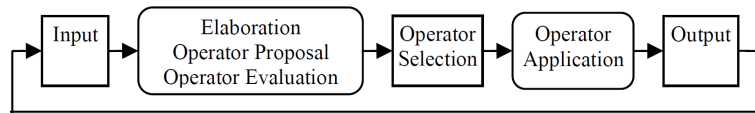


Figure 7: Processing cycle of Soar [17].

Soar's processing cycle, see Figure 7: Processing cycle of Soar [17]. Figure 7, may be split into seven steps. The first step is *Input*, where perception changes are processed and sent to short-term memory. Then, occurs *Elaboration*, *Operator Proposal*, and *Operator Evaluation*, where rules compute entailments of short-term memory, propose operators, and create preferences for each. In the next step, *Operator Selection*, the operator is selected based on the generated preferences. The actions of the operator are performed in the step *Operator Application* and finally, any output commands are passed on to the system in the final step *Output* [18]. Up until recently, Soar's structure, see Figure 8, has consisted of a single long-term memory, which is encoded as production rules, and a single short-term memory, which is encoded as a symbolic graph structure so that objects can be represented with properties and relations. However, to expand the types of knowledge Soar could represent, reason with, and learn, Soar's structure was extended in [17], see Figure 9.

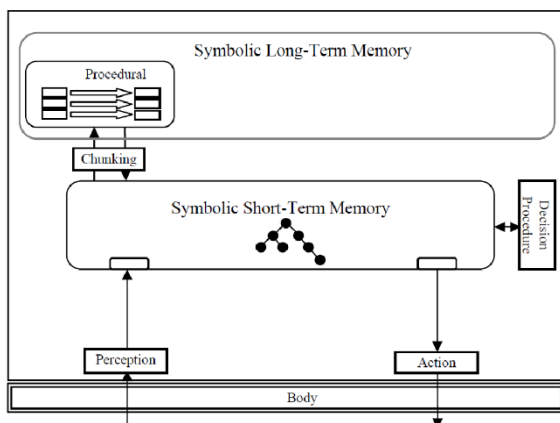


Figure 8: Earlier Structure of Soar [17].

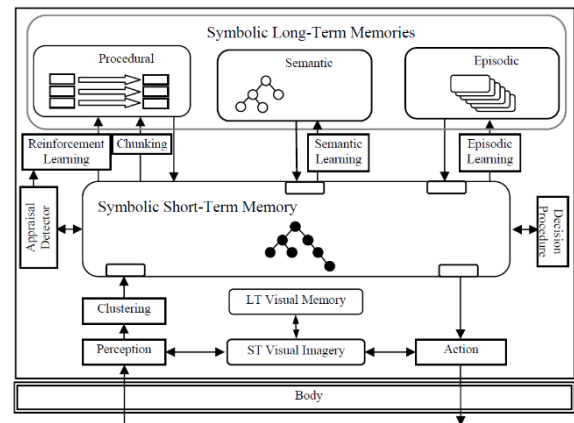


Figure 9: Extended Structure of Soar [17].

This extension, which retained the strengths of the original Soar, added new non-symbolic representations of knowledge and new learning and memory modules. The major additions include working memory activation, reinforcement learning, the appraisal detector, semantic memory, episodic memory, and clustering. Although all these new components have been built, integrated, and run with the traditional Soar components, so far, there is not a single unified system that runs all the components at once [17].

## ACT-R

The history of ACT-R (Adaptive Control of Thought-Rational) began in 1973, and, since then, it has experienced many updates and extensions, being the current version ACT-R 7 updated in 2020 [19]. It is a hybrid CA that is heavily inspired by biology and cognitive psychology and its human alike cognition emerges from the interaction of declarative knowledge, represented by data structures called chunks, and procedural knowledge, which is represented by production rules. Although chunks and production rules are symbolic constructs, the activation process is sub-symbolic, thus being considered a hybrid architecture by having both symbolic and sub-symbolic components [20].

The ACT-R architecture is designed as a production system in which rules are activated when their preconditions are met. The existence of a specific goal is one example of a precondition, while the generation of a sub-goal is an example of produced action.

The structure of ACT-R, see Figure 10, consists of modules and buffers. Modules are the mechanisms that modify and implement the buffers and each module is responsible for processing a different type of information. On the other hand, Buffers store contents visible to the other modules [19].

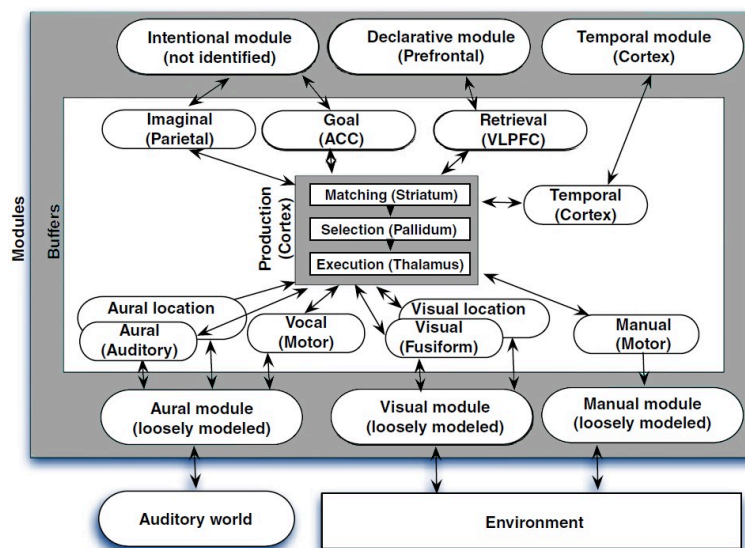


Figure 10: Structure of ACT-R 7 [19] .

ACT-R has four main modules (visual module, goal module, declarative module, and manual module) and a production system that coordinates the communication between them. The central production system has access to the buffers and will update them through the application of rules.

A cycle in ACT-R starts with the buffers holding representations determined by the external world and, using production rules, patterns in these buffers are recognized. A production that matches the working memory is selected and applied, the buffers are updated, and a new cycle starts.

## CLARION

CLARION (Connectionist Learning with Adaptive Rule Induction ON-line) is a hybrid CA in the sense that, it not only combines connectionist and symbolic representations but also implicit and explicit psychological processes [21]. It is composed of four main subsystems: the Action-Centered Subsystem (ACS), the Non-Action-Centered Subsystem (NACS), the Motivational Subsystem (MS), and the Meta-Cognitive Subsystem (MCS), Figure 11. The role of ACS is mainly action decision-making, regardless of whether the actions are for external or internal mental operations. The role of NACS is to maintain general knowledge, i.e., to store declarative and episodic knowledge. The MS is responsible for determining motivational drive levels and providing underlying motivations for perception, action, and cognition. Finally, the MCS has the role of monitoring, directing, and modifying NACS and ACS based on drive levels reported by the MS [22].

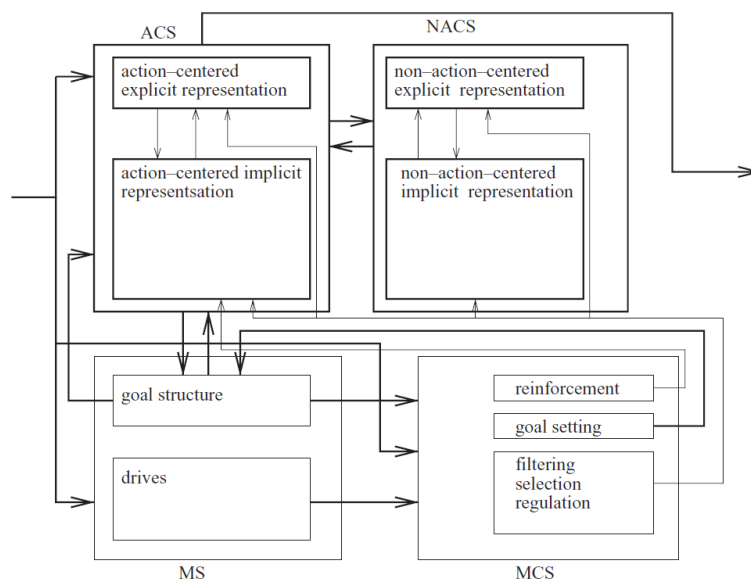


Figure 11: Structure of CLARION [23]

CLARION supports both top-down and bottom-up learning processes but the emphasis on bottom-up learning, i.e., the transformation of implicit knowledge into explicit knowledge, is what distinguishes CLARION from other CAs [22]. This approach to learning is unique as an agent may learn on its own, regardless of whether there is a priori or externally provided domain knowledge. However, the architecture does not exclude innate biases being represented within the architecture [20].

## 2.2. Modeling Emotion in Cognitive Architectures

Although there are many projects modeling emotions [23] [4] [2] [24] [25] [26], there is no consensus on what emotions are and there are many terms in emotion research literature describing what is generally referred to emotions [3] [27] [28]. Despite having different connotations, it has been widely

recognized that emotions play an important role in human life [2]. In the context of CAs, emotions are essential as they provide valuable information for decision-making, allowing systems to prioritize options and make contextually appropriate choices, and provide intention, aligning the system's goals and actions with internal drives and external stimuli. By incorporating emotions, CAs exhibit greater adaptability, decision-making capabilities, and social intelligence.

## Emotions

Emotions are complex and multifaceted phenomena that lack a consensus definition in the field of emotion research, and, within literature, various terms are used to describe them. Generally, emotions pertain to transient states and are characterized by familiar terms such as anger, interest, surprise, disgust, fear, and joy [20] [3]. Several models exist, such as Lövheim's Cube of Emotion [29], Geneva emotion wheel [30] and, Plutchik's Wheel of Emotions [31], applying different strategies, different basic emotions, and different dynamic transitions between them. These are depicted in Figure 12 – high resolution versions can be consulted in Appendix A.

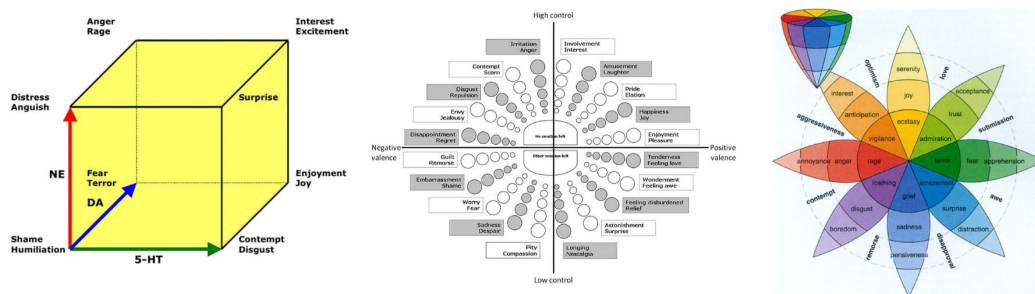


Figure 12: Lövheim's, Geneva and Plutchik's emotional models, respectively.

Lövheim presented a new three-dimensional model for monoamine neurotransmitters and emotions, the Cube of Emotion, left side in Figure 12, with eight basic emotions ordered in an orthogonal coordinate system of the three main monoaminergic axes. The axes represent serotonin (5-HT), dopamine (DA), and noradrenaline (NE), and each end of the arrows represents low and high levels of signaling respectively.

The Geneva Emotion Wheel (GEW) displays emotions methodically linked in a circle with their respective emotion groups. The alignment of the emotion terminology is based on two dimensions: control (low to high) and valence (negative to positive), dividing the emotions into four quadrants: negative/low control, negative/high control, positive/low control, and positive/high control. The numerous reaction choices are represented by "spikes" on the wheel that represent varying degrees of intensity for each emotion family, ranging from low intensity (towards the center of the wheel) to high intensity (towards the perimeter of the wheel).

Plutchik's Wheel of Emotions, depicted in the right in Figure 12, a three-dimensional circumplex model that shows the connections between various emotions, comparable to a color wheel, where emotions are represented by different hues. The model consists of a circle to represent the degree of similarity between emotions and a vertical cone to express intensity. The eight sectors of the wheel, which are arranged as four pairs of opposites, represent the theory's eight basic dimensions of emotions.

### Effects of Emotion in Memory and Cognition

Emotion plays a significant role in shaping our memory and cognitive processes. Numerous studies have explored the effects of emotion on learning, memory formation, and information retrieval. It has been observed that emotional events and stimuli are often better remembered compared to neutral ones, exhibiting higher levels of confidence, vividness, and detail [32].

In the realm of human memory, researchers have discovered that the consolidation of information can be selectively enhanced if it is conceptually related and made salient through an emotional learning experience. This selective consolidation occurs when emotionally relevant information is represented in a common neural substrate [33]. However, when it comes to memory for complex emotional stimuli, a trade-off effect has been observed. Memory for emotionally salient components is enhanced, but at the expense of memory for neutral contextual details [34].

Emotion exerts a powerful influence on attention, modulating the selectivity of attention, and motivating action and behavior [35]. It has been found that some emotions facilitate the encoding of information and aid in efficient retrieval. For instance, when information is presented to someone while experiencing a positive emotion, they are more likely to remember it, than if in a negative emotional state [36].

Understanding and modeling the effects of emotion on attention, perception, and cognitive processing is a critical aspect of studying emotions. Extensive research has focused on the effects of different affective states, such as anxiety, fear, anger, and positive/negative affect, on attention and cognition. These effects manifest in alterations of attentional processing, working memory, perceptual categorization biases, memory encoding and recall, reasoning, judgment, decision-making, and learning. Emotion can influence both low-level processes like attention and working memory capacity, as well as higher-level processes like situation assessment, planning, and judgment [4].

### Integrating Emotion in Cognitive Architectures

As the goal of this dissertation is to modulate and integrate emotions by designing the internal affect

subsystem, two additional CA NEUromodulating COGnitive ARchitecture (NEUCOGAR) and MAMID are presented. Although there are many other projects modeling emotions [21] [22] [23], these may be used to represent the two main approaches to modulation and integration of emotions within cognitive systems, either through neuromodulation or through affect states.

NEUCOGAR's foundation lies in Lovheim's emotion cube, integrating the biochemical influences of monoamines, such as dopamine, serotonin, and noradrenaline, into the computational processes of modern computers [2]. The information on the cube was adapted to the computing domain and extended to include parameters related to computing power, memory, and storage, as seen in Figure 13. It intended to bridge neuroscience and psychology with concepts of computing and AI systems.

In NEUCOGAR, dopamine is associated with increased processor activity and memorized data, reflecting its role in reward processing and motivation. Serotonin influences computing utilization and storage bandwidth, enhancing overall processor activity and connectivity between nodes. Noradrenaline impacts computing and memory distribution, allowing the system to react to specific conditions and alerts, affecting processing and memory configuration.

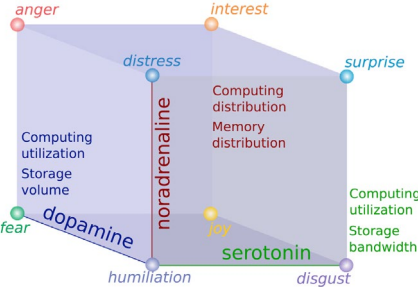


Figure 13: NEUCOGAR's mapping of Lovheim's cube of emotions to computing parameters as in [2].

On the other hand, the MAMID CA models emotion through affect appraisal [4]. It comprises various modules for sensory data processing, attention, situation assessment, expectation generation, affect appraisal, goal selection, and action selection.

The affect appraiser module within MAMID derives the agent's affective state by considering stimulus properties and the agent's internal context. It calculates the valence (positive or negative) and represents emotions such as anxiety, anger, sadness, and happiness. These resulting affective states influence goal and action selection, processing speed, and module capacity within the architecture.



# III - Design

In this chapter, the design of the Behavior Tree (BT) custom engine will be presented, with emphasis on the adaptation of two core elements of the BehaviorTree.CPP engine, the Extensible Markup Language (XML) parser, and the *Blackboard*, to enhance their compatibility with the proposed cognitive architecture (CA) and adapt their resource usage to an embedded environment. The design of the internal affect subsystem of the CA will also be discussed in detail, including an overview of its core elements and the interactions between them and the physical world subsystem.

## 1. Behavior Tree Engine

To fine-tune BehaviorTree.CPP's XML parser and *Blackboard* to this application, it is crucial to first gain a deep understanding of how they work. In the following section, both the XML parser and *Blackboard* will be dissected to identify possible areas of improvement. The structure and functionalities of each component will be analyzed to determine their strengths and limitations and finally, the changes and adaptations that are made to each component will be discussed.

### 1.1. BehaviorTree.CPP

BehaviorTree.CPP [37] is an open-source framework for designing and executing Behavior Trees (BTs). The framework is equipped with a flexible and extensible architecture, enabling the creation of custom nodes and tasks, making it well-suited for a variety of applications.

Its engine generates a BT from an XML representation, either manually written or generated by the BT IDE, GROOT [38]. The hierarchical structure of XML is particularly well suited for representing BTs, as it allows the clear and concise representation of the relationships between the nodes in the tree, making it also a convenient and interoperable format for exchanging BT information between different tools and platforms. This process is accomplished by the BT factory, which is a critical element in the process of generating BTs from an XML representation. It is responsible for creating an instance of the XML parser, which generates node objects for each node in the tree and inserts them into the BehaviorTree.CPP object. The resulting tree structure is organized into a vector of subtrees, each with a vector of nodes and a Blackboard, which enables nodes to communicate and coordinate. This approach enables the BehaviorTree.CPP framework to provide, together with GROOT IDE, a flexible, intuitive, and extensible

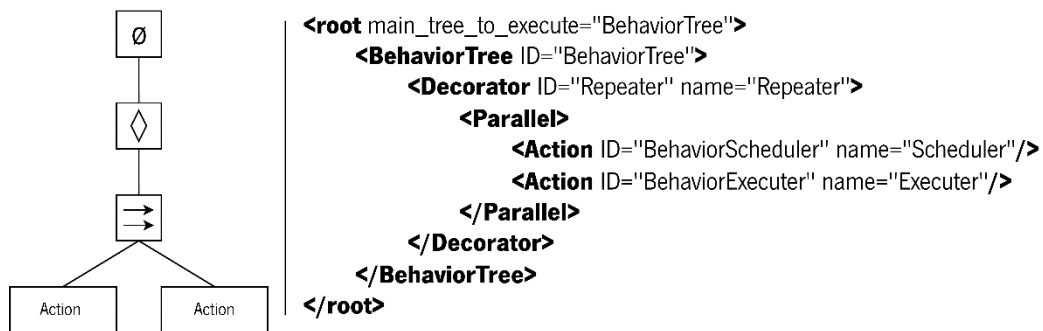
architecture for implementing BTs with custom nodes and tasks.

### XML Representation of Behavior Trees

To better understand how the XML parser works in generating BTs in the BehaviorTree.CPP framework, it is beneficial to first understand how BTs are structured in XML format by the GROOT IDE.

The XML format used by GROOT IDE is based on a set of predefined tags that represent different types of nodes. For example, the `<root>` tag represents the root node of the BT, while the `<sequence>` and `<fallback>` tags represent control flow nodes that determine the flow of the BT.

Default Control flow nodes from the GROOT IDE do not possess any attribute, however, custom Control flow nodes and Leaf nodes are defined by a set of attributes that specify their type and behavior. Custom nodes of each type can be created and marked with the corresponding tag (Decorator, Control, Action, and Condition). In Figure 14, the decorator node is a custom node created in the Decorator type category and therefore has the tag `Decorator`. The main attributes of custom nodes are an ID, which identifies the custom node, and a name that provides means to distinguish between different instances of the same custom node.



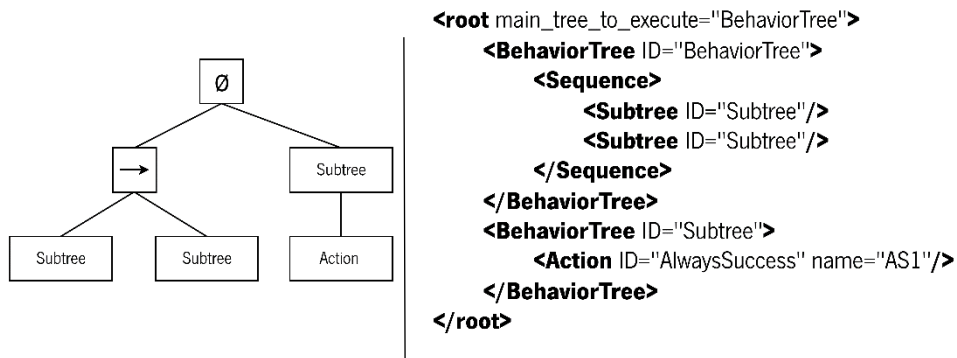


Figure 15: Subtree representation in both BT and XML representations.

### XML Parser

After establishing the structure of BTs in GROOT IDE's XML format, the parser converts the XML representation into the BT structure that can be executed by the BehaviorTree.CPP engine. This process is achieved using the TinyXML2 library [39], which is a lightweight open-source C++ library for parsing and manipulating XML documents.

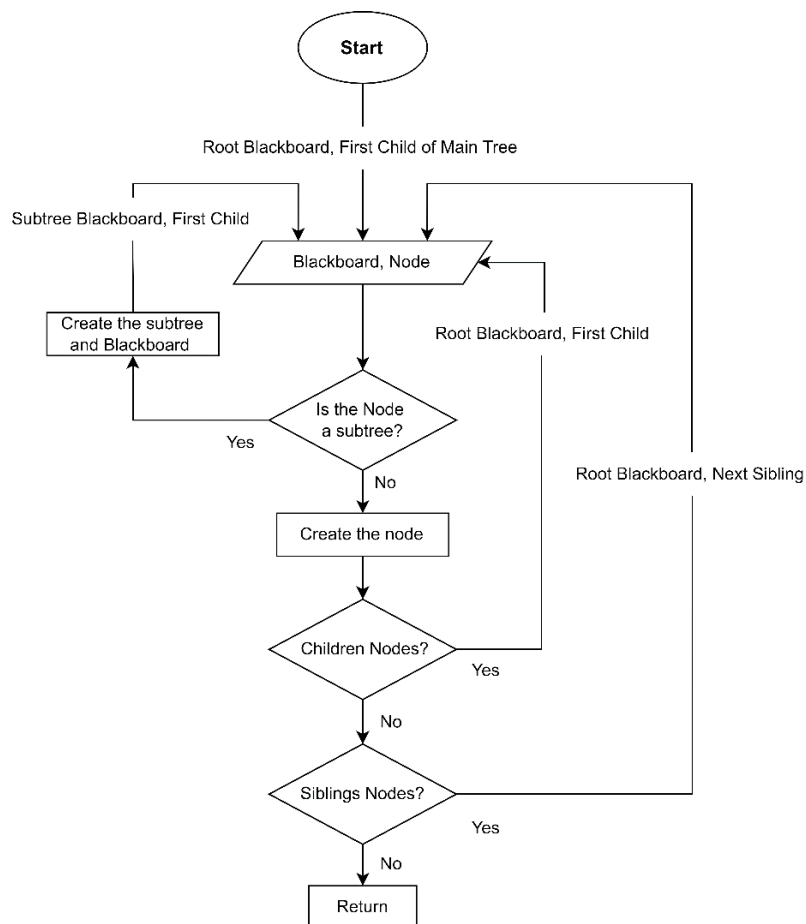


Figure 16: Flowchart describing the parsing algorithm in BehaviorTree.CPP.

As seen in the flowchart in Figure 16, to instantiate a BT, the XML Parser class takes a root *Blackboard*

and the first child of the main tree as input. If a main tree ID is not provided, the parser automatically selects the ID from the *main\_tree\_to\_execute* attribute in the first opened document or from the only registered tree ID. The parsing process involves recursively creating nodes from the XML elements with the root blackboard and the parent node as input. The parser determines whether each node is a subtree or a child node. If the node is a regular node, that is, not a subtree, the node is created, and a recursive call is made with its first child or the next sibling as the argument. However, if the node is a subtree, a new *Blackboard* is created for that subtree and remapping is performed if necessary. The parser then recursively creates the subtree nodes with the new blackboard and the current node as inputs. After this process, the BT is organized into a format that can be executed by the BehaviorTree.CPP engine.

This XML parser also provides detailed error messages when parsing fails, which enables quick detection and resolution of parsing errors. The error-handling feature of the parser helps ensure the robustness and reliability of the BT.

### Blackboard

Before delving into the specifics of the *Blackboard* in the engine BehaviorTree.CPP, it is important to first understand how data is passed between Nodes. The key mechanism for this is through ports, which are similar to the parameters (inputs) and return values (outputs) of functions. This dataflow mechanism between nodes is also used to interface with the *Blackboard*. to read and write entries through the input and output ports, respectively, as illustrated in Figure 17.

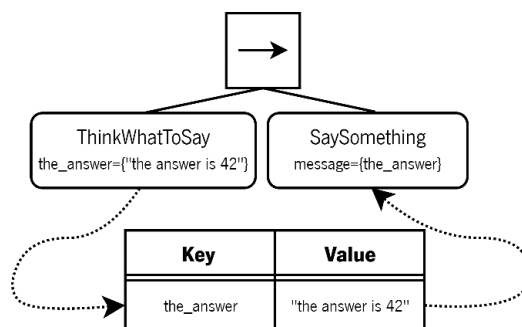


Figure 17: Illustration of the Blackboard functionality, adapted from [40].

The *Blackboard* is a key-value store that is used to share typed data between different nodes in a BT designed to be thread-safe and managing simultaneous accesses from multiple threads.

This *Blackboard* was designed to adhere to the principles of the singleton pattern, ensuring that only one instance of the class exists, with a global point of access provided to that instance. The *Blackboard* class relies on the utilization of an STL container, `std::unordered_map`. This container offers an average time

complexity of  $O(1)$  for both lookups and writes, making it a scalable solution for rapid and efficient data retrieval, independent of the number of elements within the container.

As shown in the previous section, the main tree and any subtree use different instances of the *Blackboard*. Therefore, it is necessary to explicitly connect the ports of a tree to those of its subtrees. When a key-value pair is added to the *Blackboard*, the key is used to map to the corresponding value. However, the key can also be remapped to a different key, allowing for more flexibility in the naming conventions used for the keys. This enables the integration of different Blackboards from different subtrees that may use different naming conventions for the same instance of data. For instance, as seen in Figure 18, the key-value pair which is stored in one *Blackboard* with the key "the\_answer" has "answer" as the key in the subtree that needs to access this data. By using the *Blackboard's* remapping feature, the key "the\_answer" can be remapped to "answer", allowing for seamless integration between the two *Blackboards* without the need for modifying the underlying code.

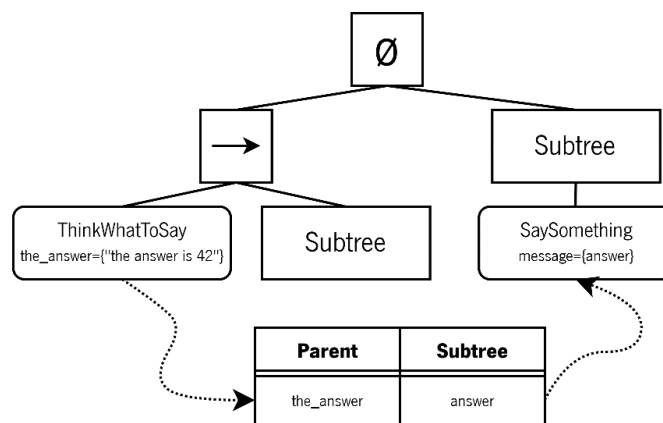


Figure 18: Illustration of port remapping functionality, adapted from [40].

## 1.2. Behavior Tree Custom Engine

A new BT engine was developed from scratch to increase compatibility with the proposed CA and adapt the resource usage to an embedded environment. This granted full control over its functionalities, resulting in the improvement of crucial components and the removal of any unnecessary elements.

For this dissertation, only the design of the XML parser and the *Blackboard* are of relevance, which will be discussed in detail in the following sections. The design of elements such as Action Nodes, Control Flow Nodes, Parallel Nodes, and Decorator Nodes is outside of the scope of this dissertation. Reference [41] can be consulted for further information on these elements.

To better comprehend the algorithm of the new XML parser, it is of relevance to first understand the tree

structure of the BT custom engine. In contrast to the previous implementation, which aimed to serve as a versatile open-source solution for developing BTs, in the BT custom engine, there is no purpose for subtrees. It still supports subtrees generated in the GROOT IDE, however, they are directly inserted into its tree structure. Therefore, the use of subtrees is primarily intended to enhance the organization and simplify the development of the BT within the GROOT IDE.

In the new tree structure, upon creation, all nodes are inserted into a single node vector. Depending on its type, nodes may possess multiple children, and as such, are equipped with a children vector. The children vector contains pointers to the corresponding nodes in the node vector, which are typically the nodes that follow the parent node. After the creation of the tree structure is complete, the node vector, which acts as scaffolding, is destroyed, and a sole pointer to the first node represents the root node and the beginning of the tree. In Figure 19, the representation of the BT shown in Figure 15 is used to illustrate the deletion of the node vector and the direct insertion of the subtrees into the main tree structure.

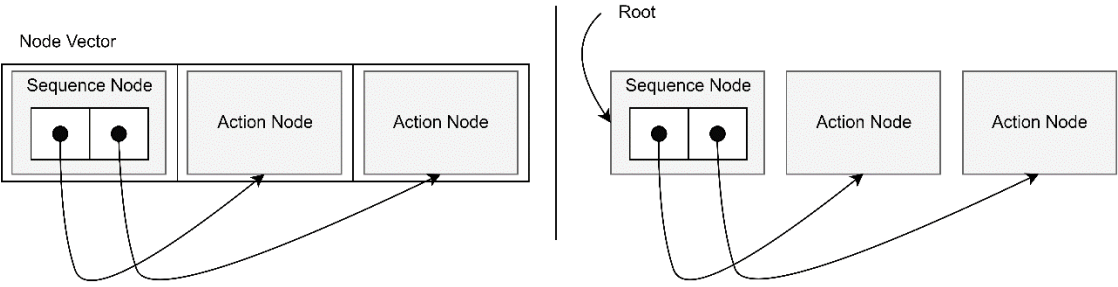


Figure 19: Tree structure before (left) and after (right) the removal of the node vector.

XML Parser

The main improvement made to the XML parser was replacing the TinyXML2 library with PugiXML [42], which yields superior performance and resource utilization. PugiXML’s design philosophy prioritizes minimizing memory usage and maximizing parsing speed, making it an optimal choice for embedded environments with limited resources. For instance, in the benchmark comparison in [43], PugiXML demonstrated significantly lower parsing times over TinyXML2, approximately four times faster, while maintaining a lower peak in memory usage, approximately half.

In addition to the benefits mentioned benefits, further memory optimizations can be achieved with the compact mode feature available in the library. This is particularly relevant as the XML BT representation is markup-heavy, resulting in a tree structure that can occupy more memory than the document itself. By activating the compact mode, the tree structure, on 64-bit architectures, can typically be reduced by a factor of approximately five [44]. Therefore, when handling large markup-heavy documents, the utilization

of compact mode can make the difference between processing of a tree running completely from RAM versus requiring swapping to disk.

The XML hierarchy representation in PugiXML is structured in a tree format that is convenient for data organization. The PugiXML library comprises several node types, with a particular emphasis on `node_element` type which are the content nodes and, in this case, translate to BT nodes. Additionally, the PugiXML library provides an effortless means of navigating and searching through the tree structure, thereby ensuring that all nodes are readily accessible. The PugiXML representation of the example in Figure 15 can be seen in Figure 20. Note that it exclusively displays the `node_element` type nodes and ignores nodes with irrelevant information such as comments and other node types.

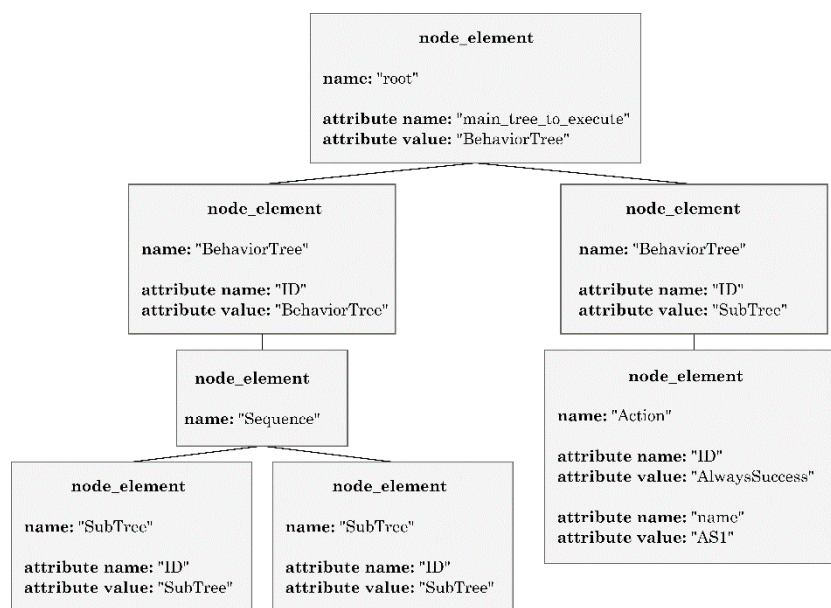


Figure 20: PugiXML's representation of Figure 15 example.

Although the tree structure has been modified to meet the specific requirements of the application at hand, the parsing process used in the BehaviorTree.CPP engine, Figure 16, remains unchanged for the most part.

The modified version, illustrated in Figure 21, depicts the simpler algorithm as subtrees are no longer mapped under the root node and the existence of a single global *Blackboard*. This also removes the need for *Blackboards* with scope limited to the subtree and the need for port remapping. The modified algorithm upon reaching a subtree node, simply finds the subtree under the root node in the PugiXML's tree structure and recursively calls itself with the subtree node as argument passed in the function call. The algorithm will then consider the subtree's children as regular nodes and insert them in the BT custom engine's tree structure in the place of the subtree node.

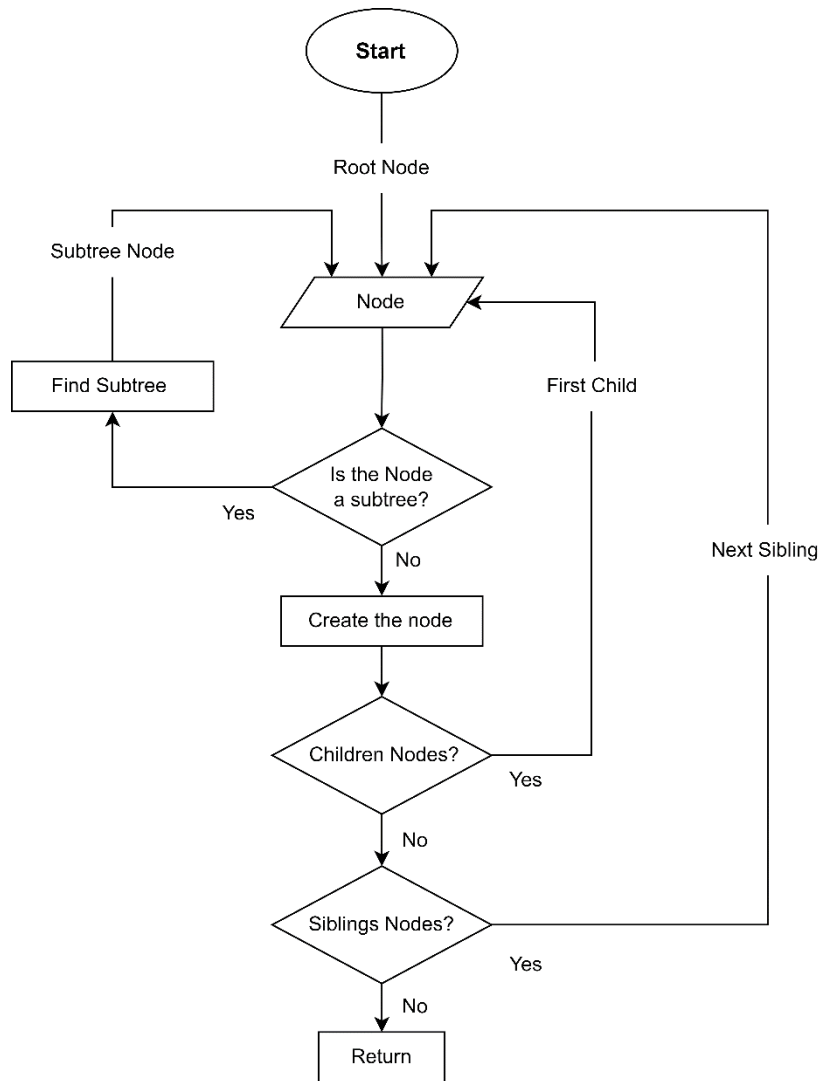


Figure 21: Flowchart describing the parsing algorithm in Behavior Tree Custom Engine.

When a node is created, a node of the corresponding type is generated and inserted into the node vector. If the new node has a parent (all nodes, except the root node, have parents), the node vector is traversed to locate the parent, and a pointer to the new node is added to its children node vector. The primary function of the node vector is to expedite the search for the parent node of the node being created, hence its deletion at the end of this process.

### Blackboard

The *Blackboard* in the BT custom engine has been improved in two main ways. First, as previously mentioned, port remapping checks were removed since subtrees are now directly inserted into the main tree and a single global *Blackboard* exists. Hence, the core structure of the storage container has been modified to eliminate the need for saving port information for each entry. As a result, instead of each entry consisting of a key paired with a construct containing the value and port information, it has been



simplified to a basic key-value pair. Second, the synchronization mechanisms have been redesigned to fully exploit the multithreading capabilities of STL containers. According to the C++ standard reference [45], all *const* member functions do not require synchronization mechanisms. As long as the lookup functions used are *const* member functions, the only need is to prevent simultaneous reading and writing. Therefore, a read/write lock has been implemented, which allows multiple threads to read from the *Blackboard* simultaneously, but only allows a single writer. This works by acquiring the lock with shared ownership for read operations and exclusive ownership for write operations, as illustrated in Figure 22. This implementation is potentially a considerable performance boost considering that the BehaviorTree.CPP's *Blackboard* does not allow simultaneous readings.

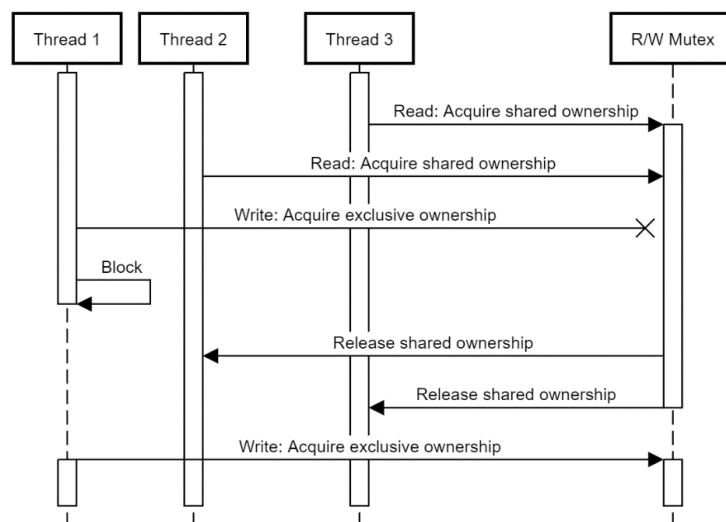


Figure 22: R/W Mutex locking mechanism.

Read heavy applications employing R/W locks may cause write operations to starve due to the prolonged read lock's access time. Writer-priority policies are a common solution to this issue however, the *std::shared\_mutex*, as explained in its official C++ standard implementation reference [46], lacks reader-writer priority policies due to an algorithm credited to Alexander Terekhov [47]. This algorithm lets the Operating System (OS) decide which thread is the next to acquire the lock without differentiating between unique and shared locks. This results in neither reader nor writer starvation.

## 2. Cognitive Architecture

The designed CA consists of a physical world subsystem, an internal affect subsystem and a memory model based on the Atkinson-Shiffrin memory model. In this subsection, the design of the internal affect subsystem and related memory components is presented, alongside a brief introduction to the physical world subsystem and memory components.

Although the focus of this dissertation is the internal affect subsystem, it is crucial to first understand the physical world subsystem and all its components to fully comprehend the designed internal affect subsystem. This is due to the internal affect subsystem being designed to be an extension and, therefore, function as a wrapper around the physical world subsystem that can be easily inserted/removed. It is important to mention that the provided explanation on the physical world subsystem is very superficial and further details are available in reference [41].

It is important to emphasize that the CA presented is intentionally kept of low complexity and does not adhere to all plausible biological concepts. For instance, memory formation and retrieval are highly complex and dynamic processes that involve intricate interactions between various brain regions and molecular mechanisms. In contrast, the memory model in this CA may simplify or abstract some of these biological complexities for the purpose of proof of concept. Additionally, the time scales and capacity of the memory stores in this CA may not be an exact match to the biological counterparts or accepted models.

## 2.1. Use Case

A practical application was devised to provide comprehensive validation of the CA, particularly the internal affect subsystem. The goal is to enable the agent, steered by the designed CA, to navigate from the current to a desired location, by either following a known path or by acquiring directions from the built-in map. This required the development of a physical structure to provide its embodiment and of the use case environment. The physical structure is equipped with moving capabilities, three sensors – a camera, a luminosity sensor, and a decibel sensor – to provide sensory stimuli and another three – two infrared sensors and an ultrasound sensor – to prevent collision. Although Validation and Results subsection offers a comprehensive explanation of its design, it is important to mention that the embodiment relies on a separate control board to handle its movement's processing. This way, the CA can take advantage of the full processing power of the target platform, a Raspberry Pi 4, and simply send I<sup>2</sup>C commands to the control board.

To reduce the number of variables influencing the use case, the environment is closed, controlled, and simplified to a single matrix drawn on the floor in which a character is randomly assigned to each square, as illustrated in Figure 23. However, upon validation of this use case, serving as a proof of concept, both the CA and the use case can be scaled to encompass higher complexity levels, with higher resemblance of real-world scenarios.

O	I	T	W	G
S	D	A		U
H	X	K	V	N
C	M	F	R	E
P	Q	L	B	J

Figure 23: Use case environment.

The physical structure is able to move in four directions: up, down, right, and left, based on commands received in the control board of the prototype from the CA. These movements are predetermined and precisely executed to traverse the appropriate distance required to transition from the central point of one reference to another. Using the onboard camera, it detects the letters in the matrix, to know its current position, and the luminosity and decibel sensors to compute the emotional score associated with it. It is important to highlight that this use case serves as a proof of concept for validation purposes and despite relying solely on values from the luminosity and decibel sensors as inputs for the emotional subsystem, other sensors can be integrated.

## 2.2. Memory Model

The memory model is based on influential Atkinson-Shiffrin memory model [48]. This model distinguishes three memory stores: a sensory register, a short-term store, and a long-term store. As depicted in Figure 24, the sensory register is a very short-lived store which temporarily holds incoming sensory information while it is being initially processed and transferred to the short-term store. The short-term store provides a working memory in which manipulations of information may take place on a temporary basis. Finally, the long-term store is a permanent repository of information. [49]

For this architecture, however, the sensory register is referred to as sensory memory (SM), the short-term store as short-term memory (STM) and the long-term store as long-term memory (LTM). Additionally, as seen in Figure 24, it includes two extra memory storages that are specifically related to the internal affect subsystem, namely emotional short-term memory (eSTM) and emotional long-term memory (eLTM). These additional memory storages are used to store recent emotional states, to determine the current mood, and to store all the experienced emotional states, respectively.

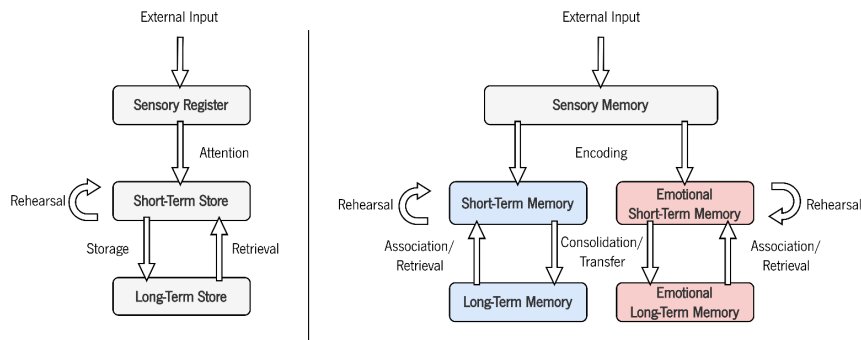


Figure 24: Atkinson-Shiffrin memory model (left) and designed memory model (right).

### Memory Stores Structure

To implement the Sensory Memory (SM), considering it requires a scalable implementation that also allows variables of any type to be stored, the *Blackboard* is used. These aspects are crucial for this type of memory to be able to store the large number of different types of outputs from the Sensory Module. Information stored in this memory store is raw and is yet to be encoded, and therefore does not yet constitute memory elements.

There are two distinct structures that capture different dimensions of our recollections: physical memory elements and emotional memory elements. Physical memory elements can either be references (characters within the matrix) or connections (actions and other information necessary to transition from one reference to another). References are constituted by a tag that delineates their matrix reference, a vector that encompasses all their connections, and the hot index. Each connection retains the tag of its target node, pertinent connection-specific details, such as distance, and the hot index.

Emotional memory elements complement the physical memory elements by incorporating the emotional aspect of the stored information. Each element is directly linked to a specific physical memory element, through its corresponding tag, and includes an emotional score, which represents the emotional state associated with the referenced physical memory. This emotional score adds a layer of subjective and affective value to the memory, reflecting the emotional impact or importance attached to the underlying physical memory element.

The STM is designed to be a circular static array of exactly seven elements since, according to George A. Miller [50] [51], there is a finite span of immediate memory which is about seven items in length. Due to its circular design, the most recent element is tracked to allow the insertion of new elements. Memory elements are inserted in the array in the index of the most recent element plus one, as illustrated in Figure 25.

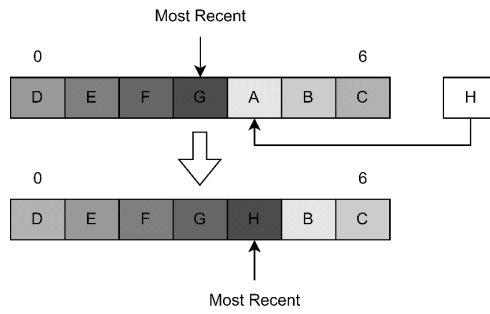


Figure 25: Insertion of an element in the STM.

The LTM is designed as a weighted and directed graph, where the edges contain relevant information for their connections, and each vertex has a vector of connections that enables bidirectional connections. For instance, vertex C can be connected to A, but A might not be connected to C, as shown in Figure 26. Considering the use case, edges are references in the matrix and vertices correspond to connections between the references. The implementation of the graph in the LTM employs an unordered map, which is a scalable container offering constant time complexity irrespective of its size. This quality makes it an ideal choice for implementing the LTM's graph structure, enabling efficient storage and retrieval of information.

The ability to save the current state of the LTM between different power cycles and restore a previous state without the need for repeated learning or knowledge acquisition is crucial. It leads to significant time and effort savings during development and testing, thus the LTM's graph structure is periodically serialized and saved into a binary file. At boot, if a previously saved state is present, this state is loaded, otherwise a new graph structure is created.

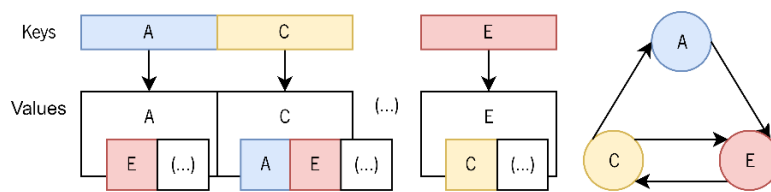


Figure 26: LTM graph structure.

Each memory element is assigned a hot index, ranging from 0 to 100, which serves as an indicator of its recognition frequency, playing a crucial role in determining the outcome of the retrieval process. Following a successful recognition of a memory trace, the probability of its retrieval is computed using a sigmoid function, seen in Figure 27, resulting in older, frequent memories being highly likely to be retrieved while new memory elements, recently transferred from the STM, possessing a very low probability of retrieval.

Every occurrence of either association or rehearsal results in an increase of the hot index by a fixed amount, although the actual increment takes place during consolidation. This amount differs from

references to connections relatively to the total possible quantity for each. Furthermore, when new information is introduced to the LTM, it triggers the decay of older memories, decreasing the hot index of all existent memory traces by a small, fixed amount. More information on these physical memory stores and their interactions can be found in [41].

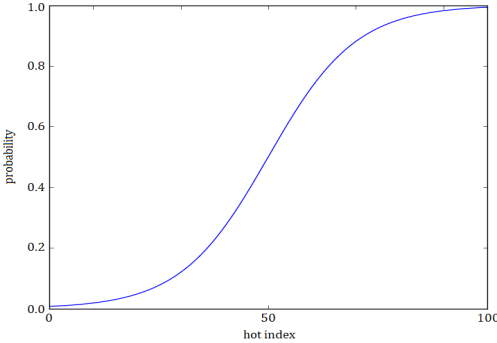


Figure 27: Sigmoid function to determine the probability of retrieval.

Shifting the focus to the emotional memory stores, eSTM and eLTM, they share structural and functional similarities with their physical counterparts, the STM and LTM respectively, and could have been designed as part of them. However, as previously mentioned, the goal is to create the internal affect subsystem as a wrapper to the physical world subsystem. Thus, the emotional memory stores are designed separately to facilitate their complete and effortless removal.

The eSTM is designed as an exact copy of its physical counterpart, however, instead of storing physical memory elements, it holds emotional memory elements and, therefore, a numeric representation of the emotional state associated with the memory element in the same position in the physical STM. The numeric representation of the emotional state is referred to as emotional score.

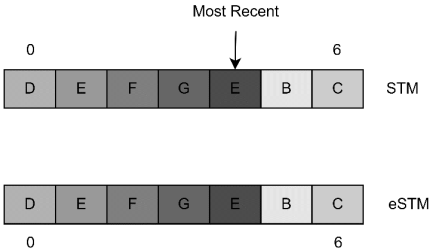


Figure 28: eSTM structure - mapping elements in eSTM to elements in STM.

Similarly, the eLTM provides a permanent store for the emotional score associated with a certain memory element. However, instead of a graph, the eLTM is designed as a hash map using an unordered map, to efficiently map a physical memory element to its emotional counterpart, thus mapping the physical memory element to its emotional score. Figure 29, complemented by Figure 26, illustrates this.

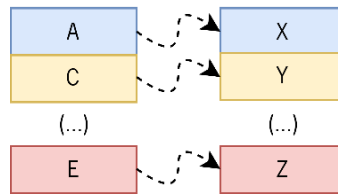


Figure 29: eLTM structure - mapping memory elements (left) to emotional state (right).

The emotional score of memories is not fixed in eLTM. Positive emotionally charged memories tend to be remembered more positively, a phenomenon known as rosy retrospection. Conversely, negatively emotionally charged memories tend to be remembered more negatively, referred to as blue retrospection [52]. While neutral memories remain unchanged, the emotional score of positive memories, with an emotional score greater than 50, increases according to the sigmoid function shown in Figure 30. Similarly, memories with an emotional score lower than 50 decrease following the same function. This phenomenon happens whenever new information is incorporated into the eLTM, resembling a process akin to the well-known interference theory [53], thereby taking place during consolidation and transfer.

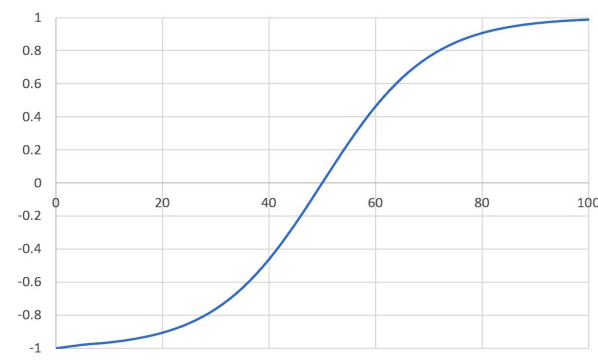


Figure 30: Sigmoid function illustrating the relationship between the decay and the emotional score.

Given the intention for the eLTM to function as a permanent repository parallel to its physical counterpart, the restoration of the corresponding eLTM state becomes imperative whenever an LTM saved state is reinstated. To ensure this, the eLTM undergoes periodic serialization into a binary file. Consequently, there is the possibility of loading the serialized eLTM file during boot, guaranteeing the consistency and persistence of the eLTM in parallel with its physical representation.

### Memory Manipulation

The management of the stored information involves various memory manipulation processes, as depicted in Figure 24, enabling the transfer of information between different memory stores. Focusing on the processes in the emotional memory stores, which mirror those of their physical counterparts, the processes are:

- Encoding – convert sensed data into a memory element that can be stored and processed by the memory system. During this process, the emotional score is computed, based on the concentration of certain hormones – stored in the *Blackboard*, this is discussed in detail in Internal Affect Subsystem – and associated with the corresponding physical memory element.
- Rehearsal – maintain relevant information in the eSTM. Due to eSTM having such limited capacity, a strategy, such as rehearsal, is required to have the right information available at the right time. This process assigns the corresponding memory element as the most recent in eSTM and triggers either consolidation or transfer, depending on whether recognition occurs or not.
- Association - link new information with existing knowledge. During this process, a memory element in the eSTM is updated with information from the corresponding memory trace present in the eLTM, allowing for the integration of new information with previously stored knowledge. The hot index from the corresponding physical memory is employed to determine the weight of the existing knowledge, with its range of 0-100 serving as a percentage. This percentage is then used to allocate the weight of the remaining percentage to the new information.
- Consolidation – update existing knowledge with new information. This process involves updating the information in the eLTM by incorporating data from the corresponding memory element in the eSTM. Similar to association, the hot index from the corresponding physical memory is employed to determine the weight of the existing knowledge.
- Transfer – transfer of information from a memory element in the eSTM to eLTM. During this process, the information of a memory element in the eSTM is inserted into a new memory element in the eLTM for permanent storage.

These processes occur when specific conditions are met. The relationships between memory manipulation processes are illustrated in the flowcharts in Figure 31 and Figure 32. The encoding process initiates whenever new information is detected, and stored in the SM. As illustrated in the flowchart shown in Figure 31, if the new information is found encoded in the eSTM, it indicates that it has recently been processed or retrieved to the eSTM. In such cases, the information is designated as the most recent in eSTM through rehearsal, thus refreshing the information in the short-term memory.



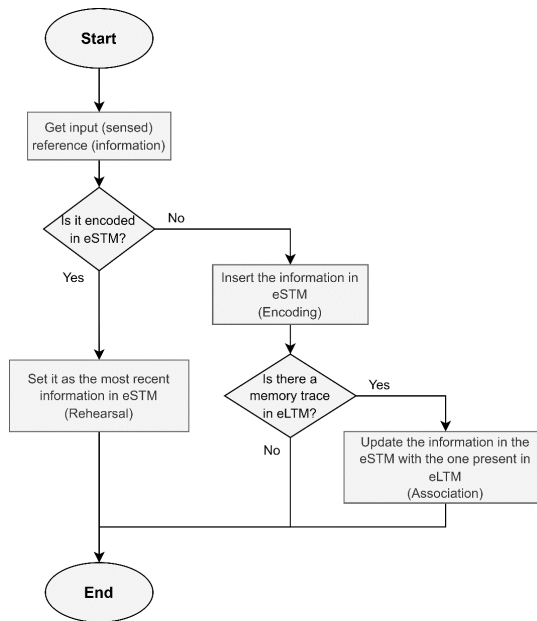


Figure 31: Flowchart illustrating the processes new information is subjected to.

On the other hand, if the information is not encoded in eSTM, it undergoes the encoding process, and the eLTM is searched for the presence of a memory trace. If no memory trace is found in the eLTM, it suggests that the memory has not yet been transferred from the eSTM. However, if detected, the information of the memory present in the eSTM is updated with the corresponding information stored in the eLTM.

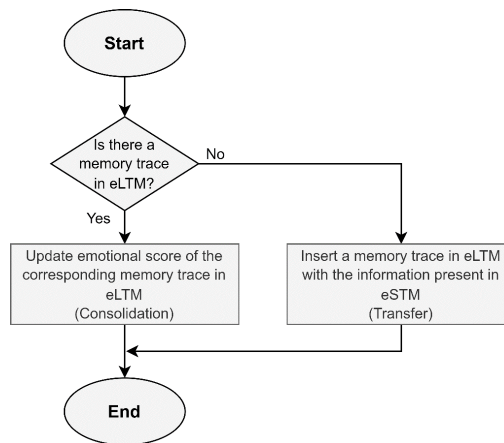


Figure 32: Flowchart illustrating the relationships between rehearsal, consolidation, and transfer.

As the flowchart in Figure 32 illustrates, every occurrence of rehearsal triggers either consolidation or transfer. Consolidation takes place when, during rehearsal, a memory trace in the eLTM is successfully recognized, requiring the memory trace to be updated with the rehearsed information from the eSTM. Transfer, on the other hand, occurs if no memory traces are found and, therefore, the information of the memory element is transferred to a brand-new memory element in the eLTM, thus creating a memory trace.

## 2.3. Subsystems

The physical world subsystem follows the sensing-processing-acting (SPA) computational model. This cycle starts by capturing stimuli from the environment (sensing), deciding how to act based on the information gathered (processing) and the current intention and, finally, executing the action scheduled by the processing stage (acting). This computational model can be observed in Figure 33, where the sensory module performs the sensing of the environment, the scheduling module schedules the next action, and the execution module executes the scheduled action.

Similarly, the internal affect subsystem follows the same SPA computational module. Starts by acquiring the sensed data from the sensory memory, which, together with the previous emotional states and the emotional state associated with the current memory from the physical world subsystem, will be used to determine the emotional state. Much like Clarion's motivational subsystem, the current emotional state is used to determine the drive and, consequently, the intention, influencing the scheduling process. The action picked by the scheduler is then evaluated, weighing emotional value into the decision and possibly selecting a more pleasant action.

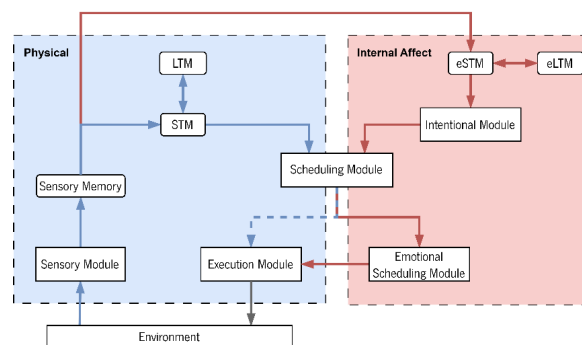


Figure 33: Designed cognitive architecture.

The physical world subsystem is structured using BTs as they not only provide a visually appealing modular representation, while facilitating the parallelization of the different modules, but also enable a reactive design approach for the CA. The internal affect subsystem is also structured using BTs, taking advantage of the high degree of flexibility and the inherent ease with which BTs facilitate modifications and extensions to the CA, to extend the physical world subsystem. The base structure of the CA using BTs can be seen in Figure 34 and as expected, it is composed of the senses to provide the sensory stimuli, the behavior scheduler, the behavior executer, and the emotional scheduler added by the internal affect subsystem. All these components execute in parallel, therefore, the arrangement of the modules in the BT is merely aligned with the SPA model for visual consistency.

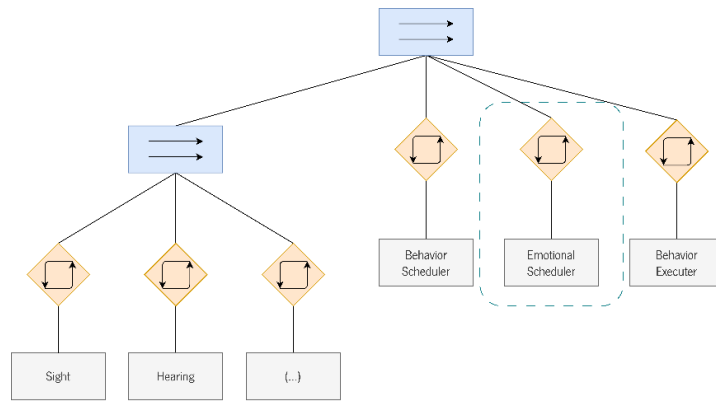


Figure 34: Base cognitive architecture structured using BTs.

## 2.4. Physical World Subsystem

The BT seen in Figure 35 depicts the final physical world subsystem. The sensory module implements a rudimentary version of the sense of sight through the camera present in the physical structure, that detects and identifies the characters on each label via an Optical Character Recognition (OCR) algorithm. The identified character is stored in the SM as a new reference. Upon detecting a new reference in the SM, the information is either encoded into the STM or the corresponding memory element rehearsed, following similar flows as depicted in Figure 31 and Figure 32. Then, a random intention is generated from all the possible characters in the matrix, which is the intended destination reference, and will determine the subsequent action to be taken (Up, Down, Left or Right). To determine this, the shortest path to the destination is traced through the memories (references and connections) in the LTM for which retrieval successfully has occurred. If a complete path cannot be traced using memories from the LTM, the built-in map, which contains all the references, connections, and possible paths, is used. As can be understood by reading reference [41], this is intended to emulate the mechanism of learning by being told. The action associated with the connection between the current reference and the next reference in the shortest path is scheduled for execution. When the execution module detects a new scheduled action, it verifies the path in that direction for any obstacles. If there are no obstructions present along the path, the action is executed.

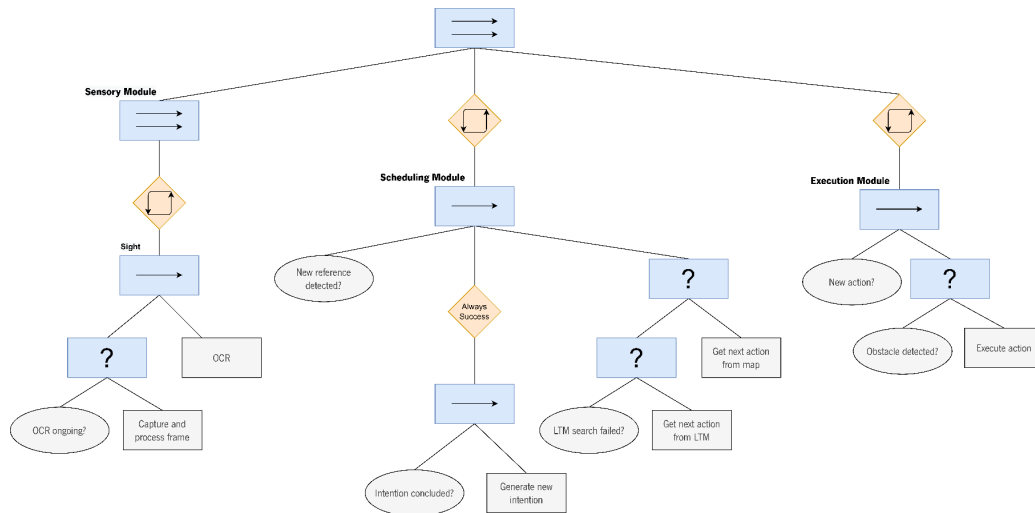


Figure 35: Final physical world subsystem structured using BTs.

## 2.5. Internal Affect Subsystem

So far, with the physical world subsystem, the CA, although functional, pursues random intentions and is incapable of taking contextually appropriate decisions. For instance, it will consistently follow the shortest path known, disregarding other factors such as lighting conditions. While a longer path may sometimes be faster due to better lighting conditions, it prioritizes the shortest path known regardless. The internal affect subsystem plays a crucial role in decision-making by assigning emotional value to memories. This enables the system to prioritize certain decisions and make contextually appropriate choices. In the specific use case mentioned, the internal affect subsystem determines the quality of each reference it encounters and utilizes this information to determine the optimal path towards the destination. It takes into account not only the distance but also factors like the quality of the path taken, ensuring a comprehensive evaluation of the available options.

Another important role of the internal affect subsystem is the motivational role, providing an intention for every action of the CA. It accomplishes this by determining drives based on the system's current emotional state. The emotional state can be categorized into three possibilities: positive, neutral, and negative. A positive emotional state, corresponding to an emotional score of above 60, will produce the “Explore” drive. This indicates a feeling of confidence and motivation, generating a random unknown reference from the map as its intention, which will enable the exploration of the map and further knowledge acquisition. On the other hand, a negative emotional state, corresponding to an emotional score of below 40, will result in the “Lift mood” drive, which focuses on lifting its current emotional state. To achieve this, the intention is changed to a known reference with a highly positive emotional score. Lastly, the “Neutral”

drive results from a neutral emotional state and simply maintains the current intention unchanged.

The current emotional state is determined by the mood, which can be computed from the concentrations of two hormones. In this design, an approach similar to Lövheim’s cube of emotions is used alongside a few assumptions and simplifications to reduce complexity. Cortisol and serotonin are selected to model stress, depression, confidence, and anxiety due to their association with these feelings. Elevated cortisol levels are typically related to the physiological response to stress [54], while cortisol can also be linked to symptoms of depression [55]. Therefore, cortisol is chosen to represent these two. On the other hand, higher levels of serotonin are often associated with a sense of confidence [56], while lower levels can contribute to feelings of anxiety [57]. Thus, serotonin is chosen to model confidence and anxiety. These assumptions are broad and simplified, considering that the hormone system is complex and not easily characterized in such a straightforward manner. However, for the purpose of this proof of concept, they are adequate.

The mood, whether positive, neutral, or negative, is determined by the concentrations of cortisol and serotonin, which range from 0 to 100 as shown in Figure 36. It is entirely dependent on the external environment and is not stored in memory, as it is computed for each moment in time.

These hormone concentrations are stored in the *Blackboard*, and the mood is calculated based on the readings from two sensors: decibels and luminosity. The decibel sensor measures the intensity of sound, with high values indicating loud noises that increase the concentration of cortisol, resulting in heightened stress. Conversely, low decibel values indicate a quiet environment, which reduces the cortisol concentration and can lead to feelings of depression. On the other hand, the values of the luminosity sensor directly impact serotonin levels. In a well-lit environment, the serotonin levels increase, promoting a sense of confidence. Conversely, in a poorly lit environment, serotonin levels decrease, which contributes to feelings of anxiety. The values of the sensors are calibrated to the default ambience noise and lighting corresponding to neutral hormonal concentrations.

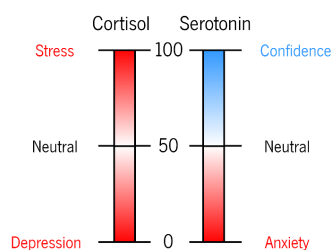


Figure 36: Concentrations of cortisol and serotonin mapped to emotions.

The mood is determined by the sum of the two concentration values, with a few modifications. When

computing the mood, the cortisol's concentration value is adjusted to range from 0 to 50 and back to 0, rather than the standard 0 to 100 range, and the concentration of serotonin is simply divided in half. This adjustment broadly results in the following mood categorizations: (1) a positive mood when serotonin levels are high and cortisol concentration is neutral, (2) a neutral mood when both hormone concentrations are neutral, and (3) a negative mood when cortisol levels are either high or low, accompanied by a low concentration of serotonin.

Whenever a saliency occurs, indicating a persistent change in mood, the emotional state undergoes a shift. While the possible emotional states remain the same (positive, neutral, and negative), the current emotional state is computed by considering two factors: the current mood and the weighted average of emotional states in the eSTM. In the weighted average, the emotional states of more recent memories in the eSTM carry greater significance, as they have a stronger impact on the current emotional state. This means that the emotional state is influenced not only by the current mood but also by the collective influence of recent emotional states weighted by their recency in the eSTM. The weights follow a decreasing exponential function, as seen in Figure 37.

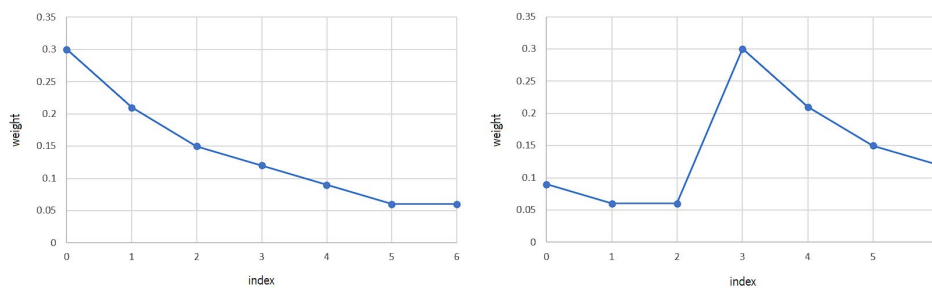


Figure 37: Weight distribution if the most recent memory element is index 0 (left) or 3 (right).

To incorporate the internal affect subsystem, nodes were introduced into the physical world subsystem, Figure 35, and some nodes required modification. Since the internal affect subsystem functions as a wrapper, these modifications do not disrupt the flow of the physical world subsystem, rather, they provide supplementary information for the internal affect subsystem. As the whole BT cannot fit into a single page, the design is segmented into 3 parts. The full BT can be consulted in Appendix A.

As depicted in Figure 38, two action nodes were introduced in the sensorial component of the CA. The action node "Measure sound intensity" conducts decibel sensor measurements during each execution tick of the BT. Moreover, it generates a signal when a significantly loud noise is detected. On the other hand, the action node "Measure light intensity" serves as a supplementary function to the existing sense of sight by reading measurements from the luminosity sensor. The readings obtained from these two sensors are normalized based on the ambient light or noise levels to correspond to the neutral

concentration of both cortisol and serotonin. The lowest and highest sensor readings are mapped to the minimum and maximum concentrations of these hormones, respectively. The concentrations of the hormones are stored in the SM.

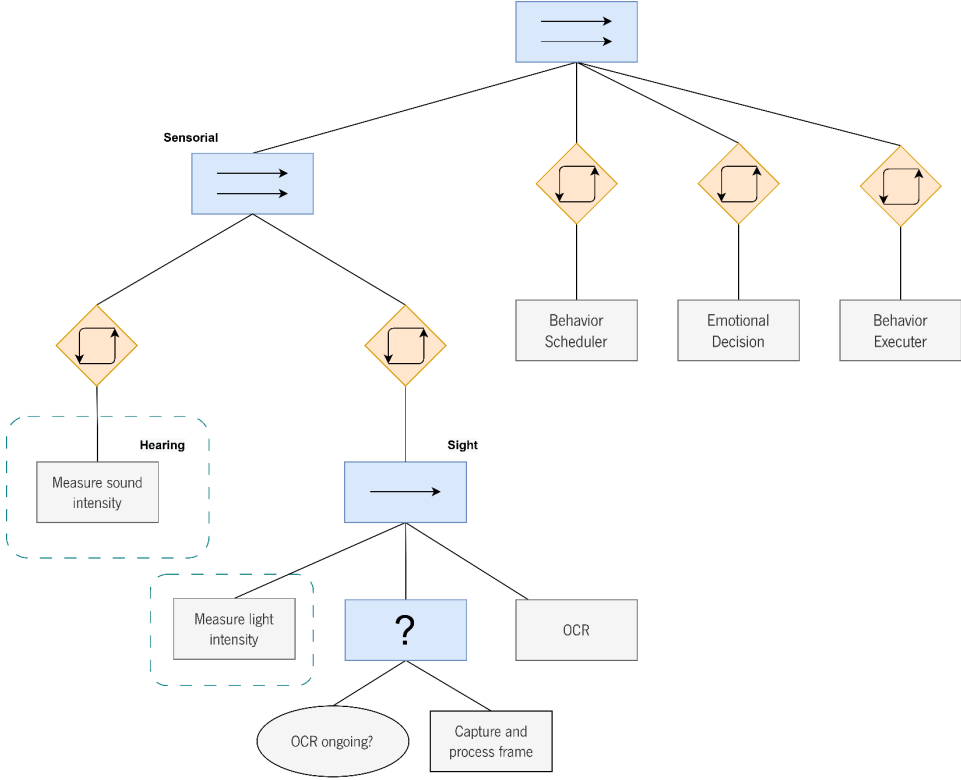


Figure 38: Action nodes added in the sensorial component of the cognitive architecture.

The additions made in the behavior scheduler, as seen in Figure 39, provide the motivational component to the CA. The role of these additions is to determine the drive in accordance with the current emotional state. This drive will generate an intention so that the decided action aligns with it.

A new sequence node is added before the existing random intention generator and three nodes are added at the left of it, to ensure their execution prior to the intention generation. The two added nodes, the condition node “Emotionally stable?” and the action node “Drive selection”, marked in Figure 39, are added under a fallback node in this order so that the action node only executes if the condition returns failure. The condition checks whether the emotional score is “negative” and if a signal was sent from the sensorial component. These scenarios result in a failure return from the condition node leading to the “Drive selection” action node execution and the subsequent selection of the adequate drive. The intention generator driver, next in the sequence node, then executes and selects an intention based on the current drive, instead of a random intention.

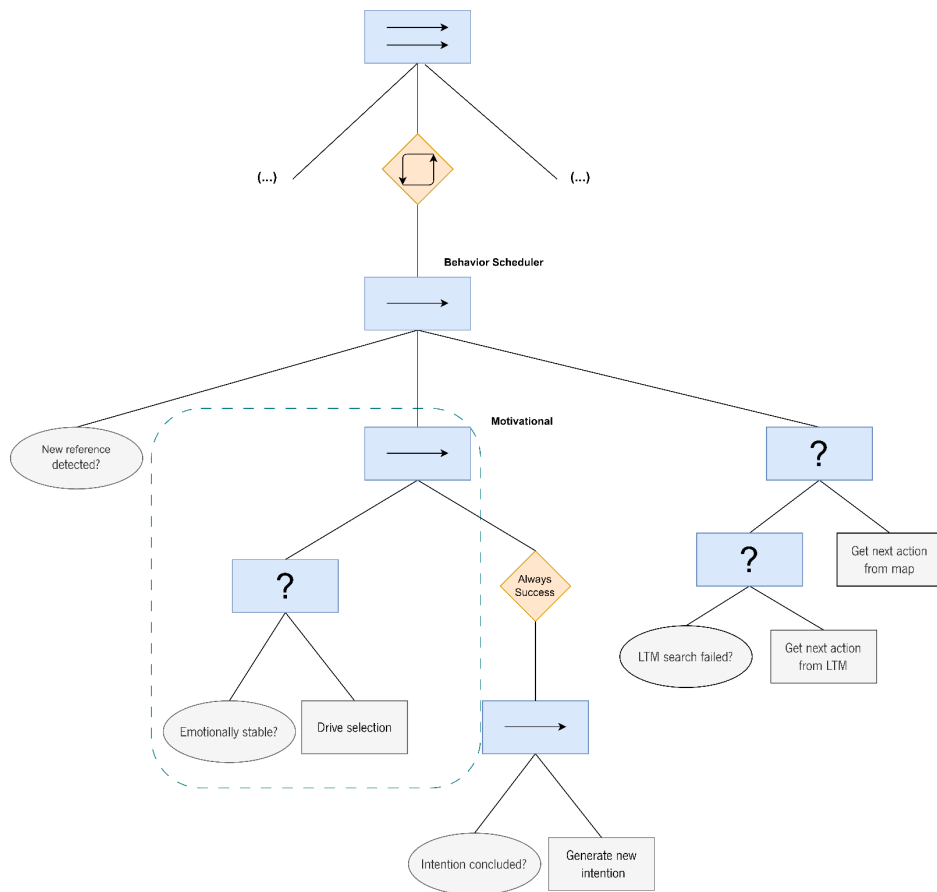


Figure 39: Additions made to the behavior scheduler of the cognitive architecture.

Following the motivational component comes the decision-making process. Here, based on the selected intention, an action is scheduled either from known knowledge or obtained from the map. This underwent modifications so that, instead of outputting a single scheduled action, it would output all the known paths to the location either from memory or from the map. This is required so that in the emotional decision module all possible paths can be evaluated, weighing in their emotional value.

The final additions to the CA's BT compose the emotional scheduling module. This module first executes self-sensing to compute its current emotional state and then influences the decision making of the physical world subsystem.

The first action node, "Self sensing", retrieves the hormone levels from the SM and determines the current mood and emotional state, as previously explained at the beginning of this subsection. It subsequently updates the memory stores based on the most recent reference in the SM, following the memory manipulation processes outlined in the subsection Memory Manipulation. Once the current emotional state is calculated, the action node "Influence decision-making" is executed. This node considers all the potential paths determined by the physical decision-making process and selects the path that is, in theory, expected to result in a higher emotional state.



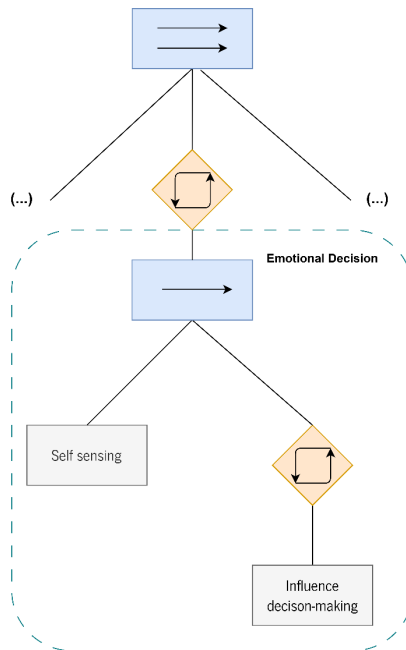


Figure 40: Emotional decision added to the cognitive architecture.

To accomplish this, as illustrated in Figure 41, a vector of potential paths is utilized. For each path, each reference is retrieved, and its corresponding emotional score is added to calculate the total sum for that particular path. Subsequently, the total sum is divided by the number of references to determine the average score per path. However, it is essential to consider the distance factor to avoid favoring longer paths disproportionately. Hence, the average score per path is once again divided by the number of references, as longer paths will naturally have more references. This yields the average score per reference. The index of the path with the highest average score per reference is saved, and the next action from that path is scheduled for execution.

If the vector of paths provided originates from the LTM and the highest average score per reference falls below the threshold value of 25, the path is deemed unsuitable, and the action node returns failure. The decorator node positioned above this action node does not adhere to the conventional repeater decorator node behavior. This repeater node will trigger a second execution of the action node below only if the action node returns failure and the path vector originated from the LTM. This ensures that a new path is computed from the map. During computation using paths determined by the map, there is a possibility that a reference has never been visited, and therefore does not have an associated emotional score. In such cases, a neutral value (50) is assigned to the reference.

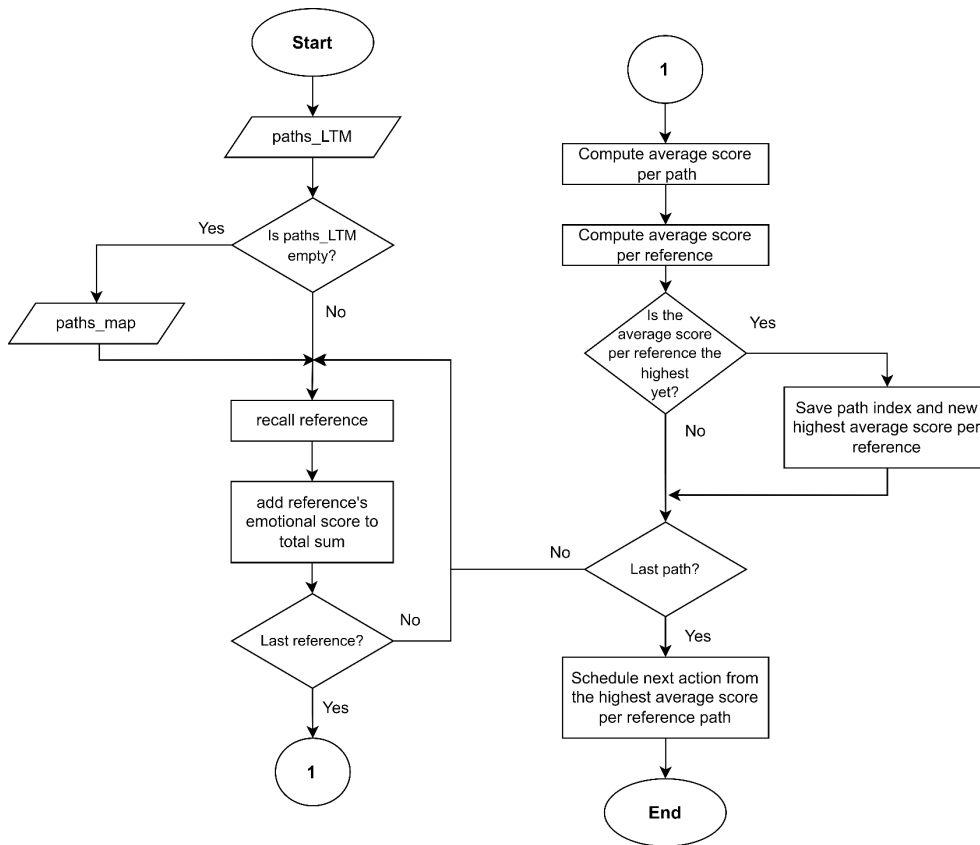


Figure 41: Flowchart illustrating the emotional decision process.

# IV - Implementation

In this chapter, it is first presented the implementation of the Behavior Tree (BT) custom engine focused on the adaptation of the two critical components explored in the Design chapter. The implementation of each of the components is reviewed in detail to exhibit the steps taken to achieve the improvements discussed in the design. Furthermore, the implementation of the internal affect subsystem of the cognitive architecture (CA) is examined alongside the required modifications to the physical world subsystem.

## 1. Behavior Tree Custom Engine

The BT custom engine is entirely implemented in C++, more specifically, C++17 as it is the latest fully supported version by g++ at the time of writing this dissertation. The structure of the BT in this engine relies heavily on pointers, however, while raw pointers provide a lightweight solution, they require manual memory management and can lead to memory leaks, dangling pointers, and other memory-related bugs. To address these issues, smart pointers, since C++11, provide automatic memory management through the Resource Acquisition Is Initialization (RAII) technique [58], ensuring that resources are properly managed and reducing the likelihood of memory-related bugs.

The execution of the engine requires a BT, and therefore, requires an Extensible Markup Language (XML) file as argument. At the very start of the execution, the number of arguments is verified to ensure that, besides the name of the program, only one is provided and returns otherwise. As seen in Listing 1, the number of arguments must be two, as the name of the program is always passed in *argv[0]*.

```
if(argc != 2){
    std::cout << "Expected 1 argument." << std::endl;
    return(-1);
}
```

Listing 1: Verification on the number of arguments.

Then, as seen in Listing 2, a smart shared pointer of *TreeNode*s is declared, which purpose is to point to the first node in the BT and, therefore, hold the entrance point and start of the BT. The *TreeNode* class defines the base type of all the nodes in the BT custom engine and is thoroughly explained in [41].

Another step taken to optimize memory management in the BT generation process is to destroy the parser object at the end of its task, which can be accomplished using a scoped block, as illustrated in Listing 2. This ensures that the parser object is created, performs its function, and is subsequently destroyed at the end of the scope, freeing up memory that is no longer required. Hence, a parser object is created and

the name of the XML file, stored in *argv[1]*, is passed to the parser as an argument of the *loadTree()*. At the end of the parsing process, the Root pointer is assigned to the first element of the generated BT.

```
//Declare a smart pointer to hold the root node of the tree structure
std::shared_ptr<TreeNode> Root;

{
    XMLParser b_tree;           //Create a X parser object
    b_tree.loadTree(argv[1]);   //Load the BT into the XML parser
    Root = b_tree.getTree();    //Assign the root pointer to the first element of the tree structure
}
```

Listing 2: Scoped block to ensure the destruction of the parser object.

## 1.1. XML Parser

In this subsection, the implementation of the XML parser and the steps taken to achieve the designed functionalities are presented. To promote code organization and maintainability, the XML parser is implemented as a class, the *XMLParser* class seen in Figure 42. It encapsulates all the details and intricacies of the algorithm and provides at its interfaces only two public methods: an input method to provide the name of the XML file containing the BT, *loadTree()*, and an output method to return a reference to the first element of the generated BT, *getTree()*.

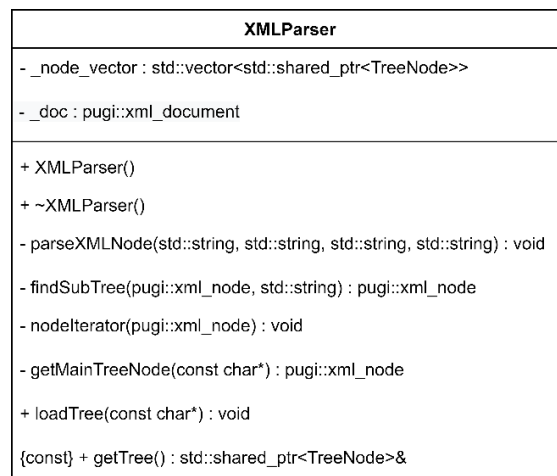


Figure 42: XMLParser class UML representation.

The *loadTree()* method, depicted in Listing 3, starts by checking if the provided file does not exist, in which case prints an error message and calls *exit()* to perform termination procedures and terminate the process. However, if the file does exist, the *getMainTreeNode()* method is called to acquire the node of the main tree which is then fed to the *nodeIterator()* method. This function will perform the algorithm discussed in the design phase and presented in Figure 21.

```

void XMLParser::loadTree(const char* xmlFile)
{
    //Confirm that the provided file can be opened before proceeding
    if(!checkFile(xmlFile)){
        std::cout << "File \"\" << xmlFile << \"\" cannot be opened or does not exist." << std::endl;
        exit(-1);
    }

    //Retrieve the first node of the main tree to execute
    pugi::xml_node main_tree = getMainTreeNode(xmlFile);

    //Construct the tree structure starting from the first node of the main tree to execute
    nodeIterator(main_tree);
}

```

Listing 3: XMLParser – *loadTree()* method.

The inline function *checkFile()*, shown in Listing 4, is used to confirm the existence of the file and that it can be opened for reading. This is achieved by attempting to open the file using the *std::ifstream* class and checking the outcome of the operation. If the operation was successful, the static cast to *bool* of a well-formed *std::basic\_ios* object, which is the base class of *std::ifstream* class, results in the value *true*. Otherwise, the *failbit* flag in the stream state will be set and the static cast will result in the value *false*. This is a very simple and efficient way of checking if a file can be opened, however, it does not provide any information on the specific reason why the file could not be opened. For instance, insufficient read privileges or sharing violation are two different reasons for the constructor of *std::ifstream* to fail, but for this application it is only important to know whether the file can be opened or not, and not the specific reason for the failure.

```

inline bool checkFile(const char *file_name) {
    //Attempt to open file_name
    //return the result of the static cast to bool of the constructed std::ios_basic object
    return static_cast<bool>(std::ifstream(file_name));
}

```

Listing 4: *checkFile()* function.

To acquire the node of the main tree, which holds the first node of the main tree as its child, the method *getMainTreeNode()* uses the following content-based traversal functions provided by the PugiXML library:

- *load\_file()* - destroys any existing document tree and attempts to load a new tree from the specified file;
- *child()/attribute()* - returns the first child/attribute with the specified name;
- *find\_child\_by\_attribute()* - returns the first child node with the specified attribute and value.

The method, presented in Listing 5, starts by loading the XML file into the PugiXML's document structure using the function *load\_file()*. The return value is then checked for errors, in which case an error message is displayed, and *exit()* is called to perform termination procedures and terminate the process. Upon a successful loading of the BT, the root node is first acquired by invoking the method *child()* on the

document with root as its argument. Then, the name of the main tree is retrieved by obtaining the value of the attribute *main\_tree\_to\_execute* in the root node. This name, as explained in the Design chapter, is used to distinguish the main tree from the subtrees. Having the name of the main tree, the next step is to find the node which has the name of the tree as the value of its attribute ID. To accomplish this, the method *find\_child\_by\_attribute()* is invoked with ID and the name of the main tree as arguments. Finally, the desired node is returned to the callee.

```

pugi::xml_node XMLParser::getMainTreeNode(const char* xmlFile) {
    //Load XML document
    if (!_doc.load_file(xmlFile)) {
        std::cout << "Failed loading file: " << xmlFile << std::endl;
        exit(-1);
    }

    //Get root node
    pugi::xml_node root_node = _doc.child("root");

    //Get ID of the main tree to execute
    std::string maintree = root_node.attribute("main_tree_to_execute").value();
    const char* maintree_c = maintree.c_str();

    //Get the node of the main tree to execute
    pugi::xml_node maintree_node = root_node.find_child_by_attribute("ID", maintree_c);
    return maintree_node;
}

```

Listing 5: XMLParser – *getMainTreeNode()* method.

The method presented in Listing 6, as implied by its name, iterates through all the nodes below the provided node, recursively. It guarantees that the *parseXMLNode()* method, Listing 7, is called for each node and therefore the corresponding node object is created. It starts by declaring and initializing an iterator that will iterate the received node's direct children. However, for each child that also has children, the method will execute a recursive call with the current node as argument. This process will repeat until a leaf node, i.e., a node without children, is found, which is the recursion break condition.

For each node, certain information is collected such as its node type, name, ID, and its parent's name and ID. This information is then used to create the corresponding node object by passing it as argument to the *parseXMLNode()* method. Then, as discussed in the Design phase, if the node is a Subtree tag, the corresponding Subtree must be located and inserted into the primary tree structure. To achieve this, the attributes of the detected Subtree tag node are iterated until the attribute ID is found and its value obtained. This ID is later used as argument in the function *find\_child\_by\_attribute()* to acquire the corresponding node object, which in this context, is the first child of the Subtree. The method is then called with the first child of the Subtree as the argument, and subsequently, iterate through all its children. In contrast, for all the other node types, the method *parseXMLNode()* is called to create the node object, and if the current node still has children, the current node is passed as argument to the recursive call.

```

void XMLParser::nodeIterator(pugi::xml_node node)
{
    //Iterate node's children
    for(pugi::xml_node_iterator it = node.begin(); it != node.end(); ++it)
    {
        //Collect current node's relevant information
        std::string node_type = it->name();
        std::string node_name = it->attribute("name").value();
        std::string node_id = it->attribute("ID").value();
        std::string parent_name = it->parent().attribute("name").value();
        std::string parent_id = it->parent().name();

        //Determine if current node is a Subtree
        if(node_type == "SubTree"){
            //Iterate current node's attributes and search for ID attribute
            for(pugi::xml_attribute_iterator ai = it->attributes_begin(); ai != it->attributes_end(); ++ai)
            {
                std::string attr_name = ai->name();
                if (attr_name == "ID")
                {
                    //Retrieve the ID of the Subtree and search for the Subtree under the root node
                    std::string subtree_id = ai->value();
                    const char* st_id_c = subtree_id.c_str();
                    pugi::xml_node st_node = _doc.child("root").find_child_by_attribute("ID", st_id_c);

                    //recursively call itself with the first node of the Subtree as argument
                    nodeIterator(st_node);
                }
            }
        }
        //For all the other nodes provide required node information to construct the node object
        else parseXMLNode(node_name, node_id, parent_name, parent_id);
        //Recursively call itself with the current node as argument
        if (!it->empty()) nodeIterator(*it);
    }
}

```

Listing 6: XMLParser – *nodeIterator()* method.

The method *parseXMLNode()*, as depicted in Listing 7, is responsible for creating the corresponding node in the parsing process. It does so by evaluating the ID of the node within a switch statement, constructing the appropriate node object, and assigning it to a previously declared shared pointer. Subsequently, the shared pointer, that now holds the newly created node, is inserted into the scaffolding node vector. This vector is referred to as "scaffolding" as its sole purpose is to facilitate the search for the parent node, and it is destroyed at the conclusion of the parsing process when the parsing object goes out of scope, as discussed in the Design section.

To locate the parent node in the scaffolding node vector, the parent node's name and ID, received from the callee, are utilized. Once the parent node is identified, the newly created node is added to the parent node's children vector. However, this requires the use of a *dynamic\_pointer\_cast* in order to access the *addChild()* method, as the pointers being iterated are of the base class type and this method is exclusive to the derived class *ControlFlowNode*. This method is not part of the base class as not all nodes in a BT can have children, and thus it is not a member of the base class. It is worth noting that the decorator node is a special case, as it can only have a single child. Therefore, instead of adding the node to its children vector, it simply maintains a pointer to its child node.

```

void XMLParser::parseXMLNode(std::string node_name, std::string node_id,
                             std::string parent_name, std::string parent_id) {
    //Declare a smart pointer to hold the node
    std::shared_ptr<TreeNode> node;

    //Create the corresponding node
    switch(hash(node_id)){
        //Control
        case "SequenceNode"_: node = std::make_shared<SequenceNode>(node_name); break;
        case "ParallelNode"_: node = std::make_shared<ParallelNode>(node_name); break;
        (...)

        //Actions
        case "SelfSystem"_: node = std::make_shared<A_SelfSystem>(node_name); break;
        case "SchedEM"_: node = std::make_shared<A_Sched_EM>(node_name); break;
        (...)

        //Conditions
        case "NewRef"_: node = std::make_shared<C_Sched_New_Ref>(node_name); break;
        case "NewAction"_: node = std::make_shared<C_Exec_New_Action>(node_name); break;
        (...)

        //Decorators
        case "Repeater"_: node = std::make_shared<D_Repeater>(node_name); break;
        case "AlwaysSucceed"_: node = std::make_shared<D_Always_Success>(node_name); break;
        (...)
    }

    //Insert the node into the scaffolding vector of nodes
    _node_vector.push_back(node);

    //If the node has a parent, set the node as child for that parent
    if (!parent_name.empty()) {
        for (auto it : _node_vector) {
            if (it->getName() == parent_name) {
                if(parent_id == "Decorator")
                    std::dynamic_pointer_cast<SingleChildControlFlowNode>(it)->addChild(node);
                else std::dynamic_pointer_cast<ControlFlowNode>(it)->addChild(node);
            }
        }
    }
}

```

Listing 7: XMLParser – *parseXMLNode()* method.

Prior to C++17, the usage of `std::string` as a controlling expression in a switch statement was not possible. However, with the introduction of *constexpr* hashing in C++17, `std::string` is now allowed as a controlling expression in switch statements, as seen in Listing 7. By using the keyword *constexpr*, the hash values generated by the *hash()* function, depicted in Listing 8, are determined at compile-time for the string literals used in the case statements of the switch statement. At runtime, when the switch statement is executed, the hash value of the input *node\_id* string is computed using the *hash()* function and its value is then used to match against the case labels in the switch statement. This means that the case labels are effectively constant expressions that are evaluated at compile-time and the hashing is only performed at runtime for the input string, therefore its time complexity can be considered  $O(1)$ . In contrast, the other possible approach, an if/else chain, would require multiple string comparisons at runtime, which, in the worst case, has a linear time complexity of  $O(n)$ , where  $n$  is the number of cases.

The *hash()* function, in Listing 8, takes a single parameter, *data*, which is a `std::string_view`. A



`std::string_view` is a non-owning, read-only view of a string that allows for efficient string manipulation without the need for memory allocations or copying.

The function employs the djb2 hash algorithm created by Daniel J. Bernstein [59]. It is a common hash algorithm known for its simplicity and efficiency in providing good distribution and minimal collisions for short strings. The function begins by initializing a variable `hash` with an initial value of 5381. This value was picked by the author, as testing showed that it results in fewer collisions and better avalanching. Next, the function utilizes a range-based for loop to iterate over each character in the received string view. The loop uses a reference to access each character in the string view without making a copy. Inside the loop, the hash value is updated using the djb2 hash algorithm by multiplying it by 33 and then adding the ASCII value of the character, as shown in [60]. It is important to mention that left-shifting the hash value by 5 bits, which is equivalent to multiplying it by 32, and then adding it to the original hash value results in a multiplication by 33.

```
constexpr uint32_t hash(const std::string_view data) noexcept
{
    uint32_t hash = 5381;

    // Iterate through each character in the string view
    for (const auto &e : data)
        // Update the hash using the FNV-1a hash algorithm
        hash = ((hash << 5) + hash) + e;

    return hash;
}
```

Listing 8: `hash()` function.

However, as seen in Listing 7, the function `hash()` is not invoked for each case to perform the compile-time hashing of its string literals. Instead, a User-Defined Literal (UDL) operator overload, shown in Listing 9, allows the creation of a user-defined suffix which calls the `hash()` function. This provides a more intuitive and convenient means of computing the hash values from string literals by simply adding an underscore to the end of the string. This subtle detail promotes readability and maintainability, as it does visibly impact the switch case and makes it appear as if it is working with integral types.

```
constexpr inline unsigned int operator "" _(char const * p, size_t) {
    //Return the hash value calculated by the hash() function
    return hash(p);
}
```

Listing 9: Overload of the UDL operator `""_`.

## 1.2. Blackboard

In this subsection the implementation of the *Blackboard* is presented alongside the improvements

discussed in the Design section. The *Blackboard* is implemented as a black box container which provides methods to manipulate its storage, hence its implementation as a class. As seen in its UML class representation in Figure 43, its constructor is private to prevent the construction of a different instance of the *Blackboard*, enforcing the singleton pattern. Its public methods provide means to obtain the instance of the *Blackboard* and to insert, retrieve, erase, and check the existence of entries.

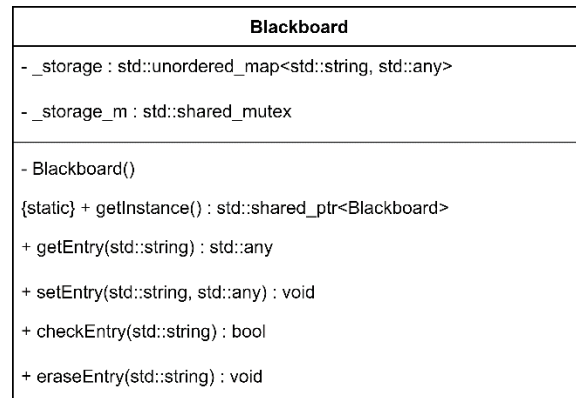


Figure 43: Blackboard class UML representation.

The chosen storage container is an *std::unordered\_map* that maps a key of type *std::string* to a value of type *std::any*. The utilization of *std::unordered\_map* is justified by its scalability and efficiency in handling key-value pair lookup and insertion operations, rendering it a suitable choice for managing unordered data. To enhance the flexibility of the *Blackboard* and enable it to effectively store and manage diverse data types, the implementation employs *std::any* as the value type, which can accommodate single values of any copy constructible type.

In order to enforce the singleton pattern, the constructor is kept private and the public method *getInstance()* is used instead. As depicted in Listing 10, this static function returns a shared pointer to the singleton instance of the *Blackboard* class. If the instance does not exist, it is created using a new *Blackboard* object, which is created only once due to the static storage allocation. Subsequent calls to this function will return the same instance of the *Blackboard*, ensuring that only one instance of the class is created and shared across the nodes.

Although *std::make\_shared*, is a more efficient alternative to *new* when creating *std::shared\_ptr* objects, as it allows for better memory management and performance optimizations, it requires the constructor of the class being instantiated to be public. This poses a limitation when creating a singleton instance hence, *new* is used.

```

static std::shared_ptr<Blackboard> getInstance() {
    // Create a static shared_ptr instance of Blackboard class
    // Initialized with a new Blackboard object, created only once
    static std::shared_ptr<Blackboard> instance(new Blackboard());
    return instance;
}

```

Listing 10: Blackboard – *getInstance()* function.

As discussed during the Design phase, a read/write lock is implemented to synchronize the access to the *Blackboard*. This is achieved by taking advantage of the *std::shared\_mutex* which can be locked using two different mechanisms, providing both shared and unique ownership. When using *std::lock\_guard* to lock the shared mutex, it acquires exclusive ownership of the mutex, thereby blocking all other threads including those attempting to acquire read-only locks, from accessing the shared resource. On the other hand, when using *std::shared\_lock* to lock the shared mutex for read-only operations, it allows multiple threads to concurrently acquire read locks, enabling efficient concurrent read access to the shared resource without blocking other threads.

Listing 11 shows the aliases *read\_only\_lock* and *write\_lock* that are created to provide a simplified and intuitive usage of the corresponding locking mechanisms with the *std::shared\_mutex*. These aliases also facilitate the readability of the code, making it clearer and more concise when utilizing the shared and unique ownership of the mutex for read-only and write operations, respectively. The lock mechanisms being used, *std::lock\_guard* and *std::shared\_lock*, are scoped locks that automatically release the mutex when control leaves the scope in which the lock object was created, ensuring proper resource management. To ensure minimal contention for the mutex and promote efficient concurrent access to the shared resource, the scope in which the lock objects are being created is kept as small as possible. This allows other threads to acquire the lock as soon as it is no longer needed, minimizing potential contention.

```

// Define an alias for std::shared_mutex
using mutex_type = std::shared_mutex;
// Define an alias for std::shared_lock and std::lock_guard.
using read_only_lock = std::shared_lock<mutex_type>;
using write_lock = std::lock_guard<mutex_type>;

```

Listing 11: Alias for mutex and locking mechanisms.

To determine the existence of an entry, the *checkEntry()* method is provided. This method, as shown in Listing 12, returns a *boolean* value, true if the entry with the given key exists in the *\_storage* container, and false otherwise. It begins by acquiring the mutex with shared ownership to allow concurrent access by multiple threads for read operations. Subsequently, the *find()* member function of the *std::unordered\_map* is utilized to search the container for the provided key. The *find()* function returns an iterator to the element with the corresponding key, or an iterator to the element following the last

element of the bucket, if no such element is found. The return value is verified, and the method returns true if a valid iterator is obtained, and false otherwise, indicating the presence or absence of the element in the container, respectively.

```
bool Blackboard::checkEntry(std::string key)
{
    // Acquire a read-only lock on the mutex
    read_only_lock lck (m);

    // Check if the entry exists in the storage
    if (storage_.find(key) != storage_.end())
    {
        return true; // Return true if entry exists
    }

    return false; // Return false if entry does not exist
}
```

Listing 12: Blackboard – *checkEntry()* method.

The *setEntry()* method, presented in Listing 13, takes a key and a value as arguments, and sets the value for the corresponding key in the *\_storage* container. It starts by checking if the entry already exists in the container using the previously discussed *checkEntry()* method. If the entry already exists, it acquires exclusive ownership of the mutex with *write\_lock* to ensure exclusive access to the container and then updates the value for the key. On the other hand, if the entry does not exist, after acquiring the mutex with *write\_lock* it uses the *std::unordered\_map*'s member function *emplace()* to insert a new key-value pair into the container. The use of *emplace()* is preferred over *operator[]* because it avoids unnecessary value copying if the key already exists.

```
void Blackboard::setEntry(std::string key, std::any value)
{
    // Check if the entry already exists
    if(checkEntry(key)) {
        write_lock lck (m); // Lock the mutex with exclusive ownership
        storage_[key] = value; // Update the value of the existing entry
    }
    else {
        write_lock lck (m); // Lock the mutex with exclusive ownership
        storage_.emplace(key, value); // Add a new entry with the provided key-value pair
    }
}
```

Listing 13: Blackboard – *setEntry()* method.

The *getEntry()* method, shown in Listing 14, shares similarities with the previously discussed *checkEntry()* method. Both methods acquire shared ownership of the mutex using *read\_only\_lock* to enable concurrent read access by multiple threads, and both methods use the *find()* member function of *std::unordered\_map* to check for the existence of a key in the *\_storage* container. The key difference is the return type. Instead of a *boolean* type return indicating the existence of a key/value pair, the *getEntry()* method returns the value for the provided key. An exception is thrown if a non-existing entry is requested.

```

std::any Blackboard::getEntry(std::string key)
{
    // Acquire shared ownership of the mutex
    read_only_lock lck (m);

    // Check if entry exists in the container
    if (storage_.find(key) != storage_.end()){ //entry exists

        // Return the value associated with the key
        return storage_[key];
    }

    // Throw an exception if entry doesn't exist
    throw std::logic_error("Entry doesn't exist");
}

```

Listing 14: Blackboard – *getEntry()* method.

As expected, the return type is *std::any* and must be casted to the appropriate type using *std::any\_cast* when assigned to a variable of its original type. For instance, consider the key/value pair `<std::string, int>`. The value variable, which is of type *integer*, is stored as *std::any* in the container and therefore must be casted to *integer* type when retrieved, as seen in Listing 15.

```

int value = std::any_cast<int>(blackboard->getEntry("key"));

```

Listing 15: Cast of *std::any* type to its original type.

## 2. Cognitive Architecture

This section focuses on the practical implementation of the memory stores and BTnodes within the developed subsystem. This section addresses two key aspects: the implementation of the memory stores that were previously discussed in the design phase, and the implementation of the BT nodes integrated into the overall CA. By providing insights into the implementation of both the memory stores and BT nodes, this section offers a comprehensive understanding of how the CA was translated from the design phase to practical implementation. It showcases the technical considerations and decisions made to ensure the efficient functioning and cohesive integration of these components with the physical world subsystem.

### 2.1. Memory Stores

The design chapter has provided a detailed framework outlining the structure, functionality, and integration of these memory stores within the CA. This subsection focuses on elucidating the methodologies and mechanisms employed to accomplish the implementation, providing a comprehensive understanding of the underlying processes, translating design concepts into tangible, functional components in C++.

It is important to outline that all mutexes and their corresponding aliases utilized in the memory stores remain consistent with the description provided in the previous section, *Blackboard*. Consequently, to avoid redundancy, further elaboration on them will be omitted.

In accordance with the design outlined in the Memory Model section, each emotional memory element needs a tag variable for storing the corresponding physical memory element's tag, as well as a variable to store the associated emotional score. Figure 44 depicts the *e\_memory\_t* struct, which includes a char-type variable for the tag, a float-type variable for the emotional score, and a function to handle object serialization and deserialization.

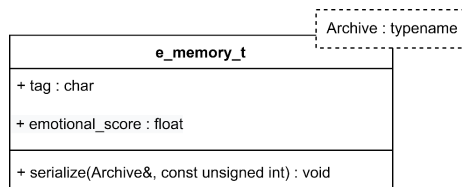


Figure 44: *e\_memory\_t* UML struct representation.

## eSTM

The eSTM is implemented as a singleton class, seen in Figure 45, to encompass all the memory manipulation processes discussed while ensuring a single instance exists. As a singleton class, the constructor is private and a *getInstance()* method provides an instance, similarly to the one described in Listing 10. As seen in the class diagram in Figure 45, the eSTM class has an array of 7 elements, the primary storage, a variable, *\_mem\_index*, to track the most recent element - as discussed in the Memory Model design subsection - and a shared mutex for synchronization.

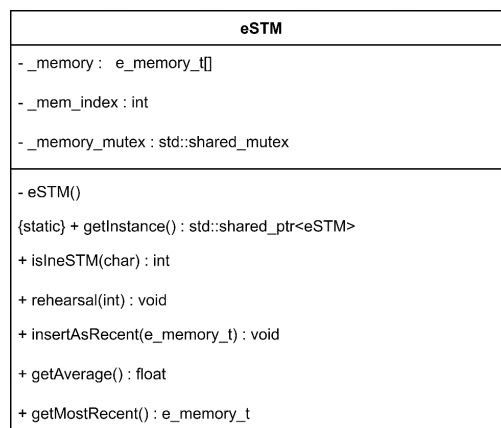


Figure 45: eSTM UML class representation.

The first method depicted in the UML class representation is the *isIneSTM()* method, illustrated in Listing 16. This method receives a tag and, after acquiring a read-only lock, iterates through the whole eSTM

looking for a match. If a match is found, the index of the corresponding element is returned, otherwise “-1” is returned to denote no match was found.

```
int eSTM::isIneSTM(char tag)
{
    //acquire a read-only lock on the mutex
    read_only_lock lck (_memory_mutex);

    //iterate whole array
    for(int i = 0; i < 7; ++i)
    {
        //return index if the desired memory element tag matches
        if(tag == _memory[i].tag) return i;
    }
    //return -1 to indicate the element was not found
    return -1;
}
```

Listing 16: eSTM – *isIneSTM()* method.

The *rehearsal()* method implements the rehearsal memory manipulation process discussed in the design phase. As illustrated in Listing 17, first an exclusive ownership lock is acquired to prevent data races, then the memory element to be rehearsed – to set as the most recent within the eSTM – is saved in a backup variable. This allows the element to be restored later after the shifts are completed. The number of shifts required is determined by the index of the most recent element and the index of the element to be rehearsed. This is achieved by subtracting the index of the element to be rehearsed, *index*, from the index of the most recent element, *\_mem\_index*, and then adding the size of the circular memory array to ensure a positive or zero result. Taking the modulus of this sum by 7 restricts the value to the range of 0 to 6, as the modulus operator wraps the result around if it exceeds the range.

A loop then executes the determined number of shifts times, shifting the elements within the memory array. The variable *curr* determines the index where the next element will be shifted to, during the rehearsal operation. It is calculated considering the starting index, *index*, and the iteration counter *i*, allowing the target position to be determined for each element to be shifted. The modulus operation ensures that the resulting index remains within the valid range for the circular memory array of size 7. This ensures the elements are shifted in a circular manner while maintaining their recency order.

After all the necessary shifts are completed, the previously saved element, *backup*, is inserted back into the memory array at the *\_mem\_index* position, making it the most recent element.

```

void eSTM::rehearsal(int index)
{
    //acquire a lock with exclusive ownership
    write_lock lck (_memory_mutex);

    //save the element to be rehearsed
    e_memory_t backup = _memory[index];

    //compute the number of shifts required
    int it = (_mem_index - index + 7) % 7;

    //iterate the number of shifts required
    for(int i = 0; i < it ; ++i)
    {
        //determine the index to where the next element will be shifted to
        uint8_t curr = (index+i)%7;
        //shift the curr+1 element to curr
        _memory[curr] = _memory[(curr+1)%7];
    }

    //insert the previously saved element as the most recent
    _memory[_mem_index] = backup;
}

```

Listing 17: eSTM – *rehearsal()* method.

The `insertAsRecent()` method in Listing 18, as the name suggests, inserts the memory element received as argument in the eSTM as the most recent element. To achieve this, the method starts by acquiring exclusive ownership of the mutex. It then increments the index `_mem_index`, which keeps track of the most recent memory element, and uses the modulus operator with the array size to ensure it wraps around within the valid range of the array. Finally, the received element is inserted in the `_mem_index` position, thus replacing the oldest element, which is transferred to the eLTM.

```

void eSTM::insertAsRecent(e_memory_t to_insert)
{
    //acquire a lock with exclusive ownership
    write_lock lck (_memory_mutex);

    //increment the index keeping track of the most recent element
    //ensuring the value remains within the size of the array
    _mem_index = (_mem_index+1) % 7;

    //prevent transfer of empty memory elements when eSTM is not full
    if(memory[_mem_index].emotional_score != 0)
    {
        //transfer element being removed of eSTM
        eltm->transfer(memory[_mem_index].tag, memory[_mem_index]);
        eltm->decay();
    }

    //insert the received element in the _mem_index position - as the most recent
    _memory[_mem_index] = to_insert;
}

```

Listing 18: eSTM – *insertAsRecent()* method.

As discussed in the design chapter, the weighted average emotional score of all the memory elements in the eSTM is required to compute the current emotional state. To achieve this, the method `getAverage()` in Listing 19, uses values taken from the sigmoid function shown in Figure 37 to weight each memory element in the eSTM.



It starts by acquiring a shared lock on the mutex as it will only perform read operations but still needs to prevent concurrent writes. Then, the value of the variable `_mem_index` is evaluated to determine if the eSTM is empty. In which case, the default value for neutral, 50, is returned.

In the other hand, if the eSTM is not empty, a variable named `total_sum` is declared and initialized to 0. This variable will accumulate the sum of emotional scores multiplied by their corresponding weights. Next, the variable `it_idx` is set to the value of `_mem_index`, which represents the index of the most recent memory element. This variable, `it_idx`, will be used to iterate through the elements in the memory array in an ordered manner, from the most recent to the oldest. Within the subsequent loop, the code iterates over all the elements in the eSTM array and, during each iteration, the emotional score of the memory element at index correspondent to the index `it_idx` is multiplied by the corresponding weight from the weights array. As it starts from the most recent memory element, it uses the loop index to iterate the weight array, which is sorted by recency, starting with the weight correspondent to the most recent element. The resulting value is added to the `total_sum` variable. After calculating the sum for the current memory element, `it_idx` is updated to the next element in a circular manner, going back in recency.

```
float weights[7] = {0.3, 0.21, 0.15, 0.12, 0.09, 0.06, 0.06};

int eSTM::getAverage()
{
    //acquire a read-only lock on the mutex
    read_only_lock lck (m);

    //_mem_index value bellow 0 indicates an empty eSTM
    if(_mem_index < 0) return 50;

    //declare and initialize total sum as 0
    int total_sum = 0;
    int it_idx = _mem_index;

    //iterate all elements in the eSTM
    for(int i = 0; i < 7; ++i){
        //add each element's emotional score multiplied by its correspondent weight
        total_sum += _memory[it_idx].emotional_score * weights[i];
        //increment the circular index
        it_idx = (it_idx + (7 - 1)) % 7;
    }

    //return the summed total
    return total_sum;
}
```

Listing 19: eSTM – `getAverage()` method.

The method `getMostRecent()`, as the name dictates, simply returns the most recent memory element. As the variable `_mem_index` tracks the most recent memory element, it simply returns the element in index `_mem_index`. Naturally, a read-only lock is acquired to prevent concurrent writes when retrieving the memory element.

```

e_memory_t eSTM::getMostRecent()
{
    //acquire a read-only lock on the mutex
    read_only_lock lck (_memory_mutex);

    //return the most recent memory element by _mem_index
    return _memory[_mem_index];
}

```

Listing 20: eSTM – *getMostRecent()* method.

## eLTM

The UML class diagram provided in Figure 46 corresponds to the class eLTM and aligns with the discussions presented in the Memory Model section in the Design chapter. The main storage container, the *\_graph* variable, is an unordered map to provide an efficient and flexible means of accessing memory elements from the corresponding physical memory’s tag. It stores pairs of keys and values, where the keys are of type *char* and represent the tag of the corresponding physical memory element, and the values are of type *e\_memory\_t*, representing the emotional memory element associated with each tag. The eLTM class is implemented as a singleton class (see discussion around Listing 10).

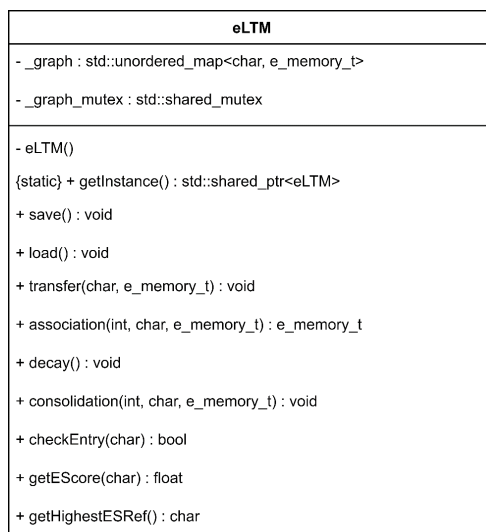


Figure 46: eLTM UML class representation.

To serialize the eLTM, the method *save()* is used. As seen in Listing 21, it starts by opening a file stream for output operations with a flag specifying that the file is treated as a binary file, meaning data is handled in a raw byte-by-byte format without any specific text encoding. This mode is suitable for storing non-textual information or structured data. Additionally, a flag is used to truncate the file if it already exists. This operation erases any existing content within the file, ensuring a clean starting point for writing new data. This output stream is used to construct a binary archive to serialize the data. Finally, the mutex is locked with exclusive ownership and the serialized unordered map is written to the file using the binary archive.

```

void eLTM::save()
{
    //Open output stream for binary file
    std::fstream wf("EM.dat", std::ios::out | std::ios::binary | std::ios::trunc);

    //Create a text output archive from the output stream
    boost::archive::binary_oarchive os(wf);

    //acquire a lock with exclusive ownership
    write_lock lck (_graph_mutex);

    //Serialize and write the unordered_map to the file
    os << _graph;
}

```

Listing 21: eLTM – *save()* method.

Similarly, the *load()* method in Listing 22 opens an input binary file stream and creates a binary input archive. This binary input archive is utilized to deserialize data from the file into the unordered map. However, before loading the file, a check is performed at the beginning to ensure that a serialized file exists and can be opened in the directory. Details on the *checkFile()* function are discussed in Listing 4.

```

void eLTM::load()
{
    //check if the provided file can be opened before proceeding
    if(checkFile("EM.dat")){

        //open input stream for binary file
        std::fstream rf("EM.dat", std::ios::in | std::ios::binary);

        //Create a text input archive for deserialization of unordered_map from the input stream
        boost::archive::binary_iarchive is(rf);

        //Load from file to LTM's graph
        is >> _graph;
    }
}

```

Listing 22: eLTM – *load()* method.

The method *transfer()*, seen in Listing 23, implements the memory manipulation process transfer. To accomplish this, the mutex is locked with exclusive ownership and the received memory element is emplaced into the graph structure using the received tag as key. The *emplace()* member function allows the new element to be constructed in-place while avoiding unnecessary copy or move operations, provided there are no elements with the provided key in the container.

```

void eLTM::transfer(char tag, e_memory_t ememory)
{
    //acquire a lock with exclusive ownership
    write_lock lck (_graph_mutex);

    //insert the value associated with the key to the eLTM
    _graph.emplace(tag, ememory);
}

```

Listing 23: eLTM – *transfer()* method.

Another memory manipulation process is association. This process links new information, received as argument, with existing knowledge in the eLTM. For this, as explained in the Memory Model section and

seen in Listing 24, the existing emotional score is weighted by the corresponding hot index in a weighted average. The remaining weight is attributed to the new emotional score, received as argument. The received memory element is updated and returned.

```
e_memory_t eLTM::association(int hot_index, char tag, e_memory_t ememory)
{
    //weight for current emotional_score
    float trace_weight = (hot_index/100);

    //weight for new emotional score
    float weight = 1 - trace_weight;

    //emotional score from memory trace
    int memory_trace = _graph[tag].emotional_score;

    //acquire a lock with exclusive ownership
    write_lock lck (_graph_mutex);

    //compute weighted average
    ememory.emotional_score = (memory_trace * trace_weight) + (ememory.emotional_score * weight);

    //return associated memory element
    return ememory;
}
```

Listing 24: eLTM – *association()* method.

Similarly, the consolidation() method depicted in Listing 25 applies the same concept as the association method, however, instead of returning the updated memory element it updates the corresponding memory element in the eLTM.

```
void eLTM::consolidation(int hot_index, char tag, e_memory_t memory){
    //weight for current emotional_score
    float trace_weight = (hot_index/100);

    //weight for new emotional score
    float weight = 1 - trace_weight;

    //emotional score from memory trace
    int memory_trace = _graph[tag].emotional_score;

    //acquire a lock with exclusive ownership
    write_lock lck (_graph_mutex);

    //compute the weighted average and update the emotional score in the eLTM
    graph[tag].emotional_score = (memory_trace*trace_weight) + (memory.emotional_score*weight);
}
```

Listing 25: eLTM – *consolidation()* method.

The method depicted in Listing 26 implements the decay process. It iterates through all memory elements in the unordered map and applies the decay function shown in Figure 30.

```
void eLTM::decay()
{
    //iterate the whole graph
    for(auto &element : _graph)
        //apply the decay for each memory element's emotional score
        element.second.emotional_score += (2/(1+exp(-0.2*(element.second.emotional_score-50)))-1);
}
```

Listing 26: eLTM – *decay()* method.

The *checkEntry()* method, seen in Listing 27, is responsible for checking whether a given tag exists in the eLTM and returns a *boolean* value, indicating the result. To achieve this, after acquiring a read-only lock on the mutex, it iterates through all elements in the eLTM. The *find()* member function of the container unordered map returns an iterator to the element with key equivalent to the provided key. If no such element is found, past-the-end iterator – same iterator returned by *end()* - is returned. This is used to determine if the element is present or not and the appropriate *boolean* value is returned.

```
bool eLTM::checkEntry(char tag)
{
    //acquire a read-only lock on the mutex
    read_only_lock lck (_graph_mutex);

    //iterate through all elements
    if (_graph.find(tag) != _graph.end()) {
        return true;
    }
    return false;
}
```

Listing 27: eLTM - checkEntry() method.

Listing 28 depicts the *getEScore()* method. This method is a simple getter function that returns the emotional score associated with the given tag. To be thread-safe and prevent concurrent writes, a read-only lock is acquired at the beginning. The method then iterates through the elements until the desired element is found. If the function is called for a non-existent memory element, it throws an appropriate exception to handle the situation.

```
float eLTM::getEScore(char tag)
{
    //acquire a read-only lock on the mutex
    read_only_lock lck (_graph_mutex);

    //iterate through all elements
    if (_graph.find(tag) != _graph.end())
    {
        //return the emotional score associated with the received tag
        return _graph[tag].emotional_score;
    }

    //throw exception if the element does not exist, which should never happen
    throw std::logic_error("[eLTM] Entry does not exist");
}
```

Listing 28: eLTM – getEScore() method.

The *getHighestESRef()* method, shown in Listing 29, is responsible for returning the reference with the highest emotional score. This method is specifically invoked when the current emotional state is significantly low, aiming to generate an intention which will lift the mood. It starts by locking the mutex with shared ownership. Then, it proceeds to iterate through each memory element in the eLTM. Within the loop, if the emotional score of the current reference is greater than the *highest\_score* encountered so far, the *highest\_score* variable is updated with the new value and the current reference is assigned to

*highest\_es\_ref*. The loop continues until all memory elements in eLTM have been processed. After the loop finishes, the method returns the reference with the highest emotional score encountered during the iteration.

```
char eLTM::getHighestESRef()
{
    //variable to store the highest emotional score reference
    char highest_es_ref;

    //variable to store the highest emotional score
    int highest_score = 0;

    //acquire a read-only lock on the mutex
    read_only_lock lck (m_graph);

    //iterate through all memory elements
    for(auto const& ref : graph)
    {
        //check if the current emotional score is higher than the previous highest score
        if(ref.second.emotional_score > highest_score)
        {
            //update the highest score and reference variables
            highest_score = ref.second.emotional_score;
            highest_es_ref = ref.first;
        }
    }

    //return the reference with the highest emotional score
    return highest_es_ref;
}
```

Listing 29: eLTM – getHighestESRef() method.

## 2.2. Internal Affect Subsystem

The creation of nodes in the BT engine falls outside the scope of this dissertation, therefore, the creation of leaf nodes has not been mentioned thus far. Without delving into excessive details, the key concept to understand is that all condition nodes inherit directly from the base class *TreeNode*, while action nodes inherit from the class *ActionNode*, which itself inherits from the *TreeNode* class. The *TreeNode* class has an *execute()* method which must be overridden since it is the method called for each execution tick of the BT. The *ActionNode* class spawns a thread in the constructor, which, when notified by the overridden *execute()* method, invokes the overridden method *taskFunction()*. The purpose of this method is to execute the intended functionality for the action node. Consequently, to be executed, all condition nodes provide an overridden *execute()* method, and all action nodes have an overridden *taskFunction()* method.

### Measure Sound Intensity and Measure Light Intensity Action Nodes

The two action nodes added in the sensorial module, "Measure light intensity" and "Measure sound intensity" are very similar. Their main difference lies in the ADC configuration, specifically adjusting the gain and the channel for reading. To avoid redundancy, only the code from the action node "Measure

light intensity” will be shown, along with the configuration details from the "Measure sound intensity" node.

Considering the sensors are connected to the same ADC in different channels, the readings must be synchronized to prevent simultaneous access to the ADC with conflicting configurations. As seen in Listing 30, a flag on the *Blackboard* is used, which is verified at the start. The value of this flag dictates if the action node returns failure or the execution proceeds by setting the flag to false immediately. To configure the ADC via I<sup>2</sup>C, the device driver for the corresponding I<sup>2</sup>C bus is opened, and the resulting file descriptor is stored in a variable. This variable is then used in the *ioctl()* system call to establish the slave address of the ADC.

```
void A_ADC_Luminosity::taskFunction()
{
    //check adc synchronization flag
    if(!std::any_cast<bool>(blackboard->getEntry("adc_sync")))
    {
        //node returns Failure
        setStatus(NodeStatus::FAILURE, action_pusher);
        return;
    }
    //acquire exclusive ownership of adc by setting flag to false
    blackboard->setEntry("adc_sync", false);

    int file_descriptor;           //declare variable for file descriptor
    int addr = ADC_SLAVE_ADDRESS; //declare and initialize variable with adc slave address
    char buf[3];                  //declare buffer to send and receive data to and from the adc

    //open the device driver for the I2C bus
    if((file_descriptor = open("/dev/i2c-1", O_RDWR)) < 0)
    {
        //print adequate error message and node returns Failure
        perror("Failed to open the bus: ");
        setStatus(NodeStatus::FAILURE, action_pusher);
        return;
    }

    //set the ADC slave address
    if(ioctl(file_descriptor, I2C_SLAVE, addr) < 0)
    {
        //print adequate error message and node returns Failure
        perror("Failed to acquire bus access and/or talk to slave: ");
        setStatus(NodeStatus::FAILURE, action_pusher);
        return;
    }
}
//----- (... ) -----
```

Listing 30: Setup ADC slave I<sup>2</sup>C address for light and decibel sensors.

As depicted in Listing 31 and Listing 32, to configure the ADC, first the address of the configuration register is sent, followed by the 16-bit desired configuration. The most significant byte (MSB), among other functions, allows the configuration of the input multiplexer to select the channel to read from, and to program the programmable gain amplifier. As the luminosity sensor is connected to pin A0, the bits 14:12 are set to 0x04.

```

#define ADC_MUX_LIGHT 0x04 //A0
#define PGA_FSR_LIGHT 0x01 //4.096V
//-----

//config register
buf[0] = 0x01;

//bits 15-8 of config register (MSByte)
buf[1] = 0x81|((ADC_MUX_LIGHT<<4)|(PGA_FSR_LIGHT<<1));

//bits 7-0 of config register (LSByte)
buf[2] = 0x83;

//send the configuration to the ADC
if(write(file, buf, 3) != 3)
{
    //print adequate error message and node returns Failure
    perror("[1] Failed to write to the i2c bus: ");
    setStatus(NodeStatus::FAILURE, action_pusher);
    return;
}

```

Listing 31: ADC configuration for luminosity sensor.

Similarly, for the decibel sensor connected to pin A1, in Listing 32, the same bits are changed to 0x05. The programmable gain amplifier is chosen individually for each sensor based on their output voltage. This ensures that the values do not reach their maximum limit while still allowing significant and measurable changes to be detected. The least significant byte (LSB) is configured with 0x83 for both sensors, which corresponds to the default configuration for data rate and comparator mode.

```

#define ADC_MUX_SOUND 0x05 //A1
#define PGA_FSR_SOUND 0x02 //2.048V
//-----

//config register
buf[0] = 0x01;

//bits 15-8 of config register (MSByte)
buf[1] = 0x81|((ADC_MUX_SOUND<<4)|(PGA_FSR_SOUND<<1));

//bits 7-0 of config register (LSByte)
buf[2] = 0x83;

//send the configuration to the ADC
if(write(file, buf, 3) != 3)
{
    //print adequate error message and node returns Failure
    perror("[1] Failed to write to the i2c bus: ");
    setStatus(NodeStatus::FAILURE, action_pusher);
    return;
}

```

Listing 32: ADC configuration for decibel sensor.

According to the ADC datasheet, the conversion time is determined by  $1/DR$ . Assuming the default data rate of 128 SPS, the thread needs to be interrupted for 8 milliseconds to allow for the conversion to complete. As seen in Listing 33, the thread is interrupted for 9 milliseconds instead to account for possible fluctuations. After that, the address of the conversion register, which holds the result of the most recent conversion, is transmitted. The result is then read and the MSB and LSB combined into a single value.



Since the ADC is 16-bit and can handle negative values, the obtained result is scaled from 0 to 100 by multiplying it by 100 and dividing it by 32767 (the range for 15-bit positive values). This value is the concentration of the corresponding hormone. Subsequently, the file descriptor is closed, and the synchronization flag is set to true, indicating to the other action node that the ADC is available for use.

```
//----- (...) -----
//wait for conversion
std::this_thread::sleep_for(std::chrono::milliseconds(9));

//conversion register
buf[0] = 0x00;

//send the conversion register address
if(write(file, buf, 1) != 1)
{
    //print adequate error message and node returns Failure
    perror("[2] Failed to write to the i2c bus: ");
    setStatus(NodeStatus::FAILURE, action_pusher);
    return;
}

//read the conversion result
if(read(file, buf, 2) != 2)
{
    //print adequate error message and node returns Failure
    perror("[3] Failed to read from the i2c bus: ");
    setStatus(NodeStatus::FAILURE, action_pusher);
    return;
}

//combine the two bytes of the conversion result
int16_t result = (buf[0] << 8) | buf[1];

//compute hormone concentration 0-100
float horm_conc = (result*100)/32767;

//close file descriptor
close(file);

//release ADC
blackboard->setEntry("adc_sync", true);
//----- (...) -----
```

Listing 33: Read converted value from the ADC and compute hormone concentration.

As each hormone's concentration is determined by the readings of the corresponding sensor, the hormone concentration is stored as cortisol in the "Measure sound intensity" action node, and as serotonin in the "Measure light intensity" action node, as shown in Listing 34 and Listing 35, respectively. Both action nodes then return success.

```
//store cortisol concentration in BB
blackboard->setEntry("cortisol", horm_conc);

//node returns Success
setStatus(NodeStatus::SUCCESS, action_pusher);
```

Listing 34: Store hormone concentration as cortisol in Blackboard.

```
//store serotonin concentration in BB
blackboard->setEntry("serotonin", horm_conc);

//node returns Success
setStatus(NodeStatus::SUCCESS, action_pusher);
```

Listing 35: Store hormone concentration as serotonin in Blackboard.

## Emotionally Stable Condition Node

As previously mentioned, condition nodes inherit directly from the `TreeNode` base class and, therefore, need only the constructor and an overridden `execute` method. In the constructor for the “Emotionally stable?” condition node, seen in Listing 36, the signal and signal handler callback function are set.

The real-time signal used for communication is `SIGRTMIN+7`. This signal is transmitted from the device driver, which monitors a GPIO pin connected to the decibel sensor. The sensor features a potentiometer that allows configuring a threshold. When the decibel level exceeds this threshold, the corresponding pin is toggled. At this point, the device driver sends a signal from the kernel space to the user space, indicating that the decibel levels have reached a critical level. It should be emphasized that signals in the kernel space and user space may have different numbering schemes. For instance, the `pthread` library uses two real-time signals, incrementing `SIGRTMIN` in the user space. This results in `SIGRTMIN`'s value corresponding to a value of 34 in the user space, while in the kernel space, `SIGRTMIN` being equal to 32.

The signal handler simply sets the `loud_noise_flag` when the defined signal is caught. As signals are software interrupts, this flag is defined as `volatile` to ensure that any changes to its value are immediately reflected in memory, preventing any optimization that may lead to inconsistent or incorrect behavior in signal handling. For similar reasons, the flag is defined as `sig_atomic_t`, a type specifically indicated for variables that are accessed and modified within signal handlers as it guarantees that read and write operations on it are atomic.

```
//set signal number considering SIGRTMIN is 34 in userspace
#define SIG_LOUD_NOISE (SIGRTMIN+7)

//declare and initialize flag for the signal handler
volatile sig_atomic_t loud_noise_flag = 0;

//signal handler callback function
void signal_catcher(int signo, siginfo_t *info, void *context){
    if(info->si_signo == SIG_LOUD_NOISE) loud_noise_flag = 1;
}

C_Sched_Safe::C_Sched_Safe(std::string name) : TreeNode (name)
{
    struct sigaction act;           //structure to configure signal and signal handler
    act.sa_flags = (SA_SIGINFO);    //sigaction (3 arg) is to be used instead of sa_handler (1 arg)
    act.sa_sigaction = signal_catcher; //handler function
    sigaction(SIG_LOUD_NOISE, &act, NULL); //register signal
}
```

Listing 36: Signal and signal handler configuration.

The overridden `execute()` method's purpose is to determine if the drive must change based on the current drive and current emotional state. As depicted in Listing 37, it starts by checking (1) if the `loud_noise_flag` is set, (2) if the emotional score is notably low, and (3) if the current drive is “Lift mood” and the emotional

score is reasonably high. In each case, flags are set to indicate the desired drive selection to the "Drive selection" action node, which execution is triggered by setting the condition node's return status as failure. If none of these cases verifies, then the return status is success, which prevents a new drive from being selected.

```

NodeStatus C_Sched_Safe::execute()
{
    //get emotional score and current drive from blackboard
    float emotional_score = std::any_cast<float>(blackboard->getEntry("emotional_score"));
    Drives drive = std::any_cast<Drives>(blackboard->getEntry("drive"));

    //Loud noise detected
    if(loud_noise_flag)
    {
        loud_noise_flag = 0;
        blackboard->setEntry("panic", true);
        blackboard->setEntry("scared", true);
        setStatus(NodeStatus::FAILURE);
        return NodeStatus::FAILURE;
    }
    //negative emotional state
    else if(emotional_score < 25.0)
    {
        blackboard->setEntry("negative", true);
        setStatus(NodeStatus::FAILURE);
        return NodeStatus::FAILURE;
    }
    //emotional state has improved
    else if((drive == Drives::LIFT_MOOD) && (emotional_score > 60))
    {
        blackboard->setEntry("positive", true);
        setStatus(NodeStatus::FAILURE);
        return NodeStatus::FAILURE;
    }
    //emotionally stable
    else
    {
        setStatus(NodeStatus::SUCCESS);
        return NodeStatus::SUCCESS;
    }
}

```

Listing 37: *execute()* method of Emotionally stable condition node.

### Drive Selection Action Node

The *taskFunction()* of this action node, as seen in Listing 38, selects a new drive based on the flags set by the previous condition node. It retrieves the flags from the *Blackboard* and sets the appropriate drive based on them. If the panic flag is set, the appropriate drive is to flee and avoid danger. In the case of a negative emotional state, the drive is changed to lift mood. On the other hand, if the emotional state has improved from negative to positive or is already positive, the drive is changed to explore as it feels confident to do so. Then, it stores the new drive in the *Blackboard* and sets a flag to signal that a new drive has been selected. This flag will trigger the calculation of a new intention in the "Generate new intention" action node based on the new drive.

```

void A_Sched_Danger::taskFunction()
{
    //retrieve flags from blackboard
    bool panic = std::any_cast<bool>(blackboard->getEntry("panic"));
    bool negative = std::any_cast<bool>(blackboard->getEntry("negative"));
    bool positive = std::any_cast<bool>(blackboard->getEntry("positive"));
    Drives drive;

    //loud noise?
    if(panic)
    {
        blackboard->setEntry("panic", false);
        drive = Drives::AVOID_DANGER;
    }
    //negative mood?
    else if(negative)
    {
        blackboard->setEntry("negative", false);
        drive = Drives::LIFT_MOOD;
    }
    //has mood improved?
    else if(positive)
    {
        blackboard->setEntry("positive", false);
        drive = Drives::EXPLORE;
    }
    //emotionally stable
    else drive = Drives::EXPLORE;

    //set new drive in blackboard
    blackboard->setEntry("drive", drive);

    //set flag to signal a new drive was set
    blackboard->setEntry("new_drive", true);

    setStatus(NodeStatus::SUCCESS, action_pusher);
}

```

Listing 38: *taskFunction()* method of Drive selection action node.

## Get Next Action from LTM and Get Next Action from Map Action Nodes

As mentioned in the design chapter, some changes were implemented in these action nodes. However, these modifications were made with caution to ensure that the physical world subsystem remains operational even without the internal affect subsystem. The primary modification involves storing a vector containing all possible paths in the blackboard, in addition to scheduling the action to be performed. Thus, in Listing 39 the *findShortestPath()* method now saves not only the shortest path but also all paths that lead to the desired reference. These paths are stored in a vector of possible paths, along with their respective distances. The vector of paths is then sorted by distance and stored in the *Blackboard*.

```

//----- (...) -----
//found the destination?
if(graph[start_node].connections[i].destination == dest) {
    //save reference in path vector
    it_path.push_back(std::make_pair(graph[start_node].connections[i].destination,
                                    graph[start_node].connections[i].action));

    //sum distance
    it_distance += graph[start_node].connections[i].distance;

    //save path
    paths.push_back(path_t(it_path, it_distance));
}

```

```

//destination not yet found
else
{
    //save reference in path vector
    it_path.push_back(std::make_pair(graph[start_node].connections[i].destination,
                                     graph[start_node].connections[i].action));

    //sum distance
    it_distance += graph[start_node].connections[i].distance;

    //recursive call to continue searching for destination
    findShortestPath(graph[start_node].connections[i].destination, it_path, it_distance, dest);
}

//path didnt lead to destination, remove reference from vector and subtract distance
it_path.pop_back();
it_distance -= graph[start_node].connections[i].distance;
//----- (...) -----

```

Listing 39: Modification made to the method *findShortestPath()*.

### Self Sensing Action Node

The *taskFunction()* of the "Self sensing" action node is responsible for computing the mood in conjunction with the current emotional state, represented by its emotional score. In the constructor shown in Listing 40 instances of the memory stores are obtained, namely the eLTM, eSTM, and LTM.

```

A_SelfSystem::A_SelfSystem(std::string name) : ActionNode(name)
{
    eltm = eLTM::getInstance(); //obtain instance of eLTM
    ltm = LTM::getInstance(); //obtain instance of LTM
    estm = eSTM::getInstance(); //obtain instance of eSTM
};

```

Listing 40: Constructor of the Self sensing action node.

As discussed during the design phase, the mood is computed based on the concentrations of serotonin and cortisol, thus they are acquired from the *Blackboard*. Then, the emotional score is determined by combining the obtained average of emotional scores from the memory elements present in the eSTM with the current mood, as seen in Listing 41. As a loud noise triggers a feeling of insecurity, it heavily affects the current emotional state. To implement this, if the flag *scared* is set, the emotional score is decreased by 25.

The function *updateMemory()* is then called to process the new information using the memory manipulation mechanisms with most recent emotional score. At the end, the final emotional score is updated on the *Blackboard*.

```

void A_SelfSystem::taskFunction()
{
    //Self sense
    float serotonin = std::any_cast<float>(blackboard->getEntry("serotonin"));
    float cortisol = std::any_cast<float>(blackboard->getEntry("cortisol"));

    //compute mood and average emotional score present in eSTM
    mood = ((cortisol>50) ? (100-cortisol) : cortisol ) + (serotonin/2);
    int average_es = estm->getAverage();
}

```

```

//compute emotional score
emotional_score = (mood+average_es)/2;

//Loud noise heavily affects the current emotional state
if(std::any_cast<bool>(blackboard->getEntry("scared"))) {
    blackboard->setEntry("scared", false);
    (emotional_score > 25) ? emotional_score -= 25 : 0;
}

//update memory stores
updateMemory();

//save current emotional score in blackboard for other nodes to access
blackboard->setEntry("emotional_score", emotional_score);

setStatus(NodeStatus::SUCCESS, action_pusher);
}

```

Listing 41: Mood and emotional score computation.

The *updateMemory()* function can be divided into two sections based on whether the new information in the SM is present in the eSTM or not. As seen in Listing 42, it begins by retrieving the most recent reference in the sensory memory and searching for it in the eSTM. If the reference is found in the eSTM, its index is returned, and rehearsal is performed, making this memory element the most recent in the eSTM. Subsequently, the eLTM is searched for a match with this memory element. If a match is found, consolidation takes place, otherwise transfer occurs. Decay affects the memory elements in the eLTM after both consolidation and transfer processes.

```

void A_SelfSystem::updateMemory(){
    //obtain most recent reference from SM
    char reference = std::any_cast<char>(blackboard->getEntry("reference"));

    //search for the reference in eSTM
    int index = estm->isIneSTM(reference);

    if( index >= 0){
        //rehearsal
        estm->rehearsal(index);
        e_memory_t ememory = estm->getMostRecent();

        //recognition?
        if(eltm->checkEntry(reference))
        {
            //get most recent reference's hot index
            reference_t c_reference = ltm->getReference(reference);
            int hot_index = c_reference.hot_index;

            //consolidation
            eltm->consolidation(hot_index, reference, ememory);
            eltm->decay();
        }
        else
        {
            //transfer
            eltm->transfer(reference, ememory);
            eltm->decay();
        }
    }
}
//----- (...) -----

```

Listing 42: First section of the updateMemory() function.

On the other hand, as shown in Listing 43, if the reference is not found in the eSTM, encoding takes place using the current emotional score. Using its tag, the memory element is searched in the eLTM and, if the element is found in the eLTM, it indicates that the previous knowledge must be incorporated in the newly encoded memory element, thus association is performed. Regardless of association occurring or not, the newly encoded element is inserted into the eSTM as the most recent memory element.

```
//----- (...) -----
else
{
    //encoding
    e_memory_t ememory = { .tag = reference, .emotional_score = emotional_score};

    //recognition?
    if(eltm->checkEntry(ememory.tag))
    {
        //get most recent reference's hot index
        reference_t c_reference = ltm->getReference(reference);
        int hot_index = c_reference.hot_index;

        //association
        ememory = eltm->association(hot_index, reference, ememory);

        //set new emotional state
        emotional_score = ememory.emotional_score;
    }

    //insert as recent in eSTM
    estm->insertAsRecent(ememory);
}
}
```

Listing 43: Second section of the updateMemory() function.

### Influence Decision Making Action Node

The method *taskFunction()* of this action node, sectioned in Listing 44, Listing 45 and Listing 46, retrieves a vector of possible paths from the Blackboard. It first attempts to fetch the vector of possible paths originated from the LTM. If this vector is empty, it indicates the physical world subsystem could not put a path together to the destination using references in the LTM and, therefore, the vector of paths originated from the map must be used instead.

```
void A_Sched_EM::taskFunction()
{
    //Get vector of possible paths from BB originated from LTM
    std::vector<path_t> paths = std::any_cast<std::vector<path_t>>(blackboard->getEntry("paths_ltm"));

    //or from MAP
    if((std::any_cast<bool>(blackboard->getEntry("es_threshold"))) || (paths.empty()))
        paths = std::any_cast<std::vector<path_t>>(blackboard->getEntry("paths_map"));
    //----- (...) -----
}
```

Listing 44: Retrieval of possible paths from the Blackboard.

In the next section of the function, Listing 45, first, some variables are initialized to hold the highest index path, its corresponding emotional score, and the average score of the best path. Then, a loop iterates

over all the paths contained in the paths vector. Within this loop another loop is nested to iterate through each reference within the current path. For each reference, it checks if there is an associated emotional score in the eLTM. If a score exists, it is added to the *score\_sum* variable, which represents the sum of emotional scores for the current path, otherwise a neutral score of 50 is assigned. After processing all the references in the path, the average emotional score for the path is calculated by dividing the sum of the emotional scores of the path by the number of references. Additionally, it computes the emotional score per reference by dividing the average emotional score for the path by the number of references in the path, thus taking the distance factor into account.

At this point, the *score\_sum* variable is reset to 0 in preparation for the next iteration. The current emotional score per reference is then checked if it is greater than the current highest emotional score per reference. If this condition is true, the *highest\_score* is updated with the new value, and the index of the current path is assigned to *highest\_idx*, to keep track of the highest scoring path in the vector. This process continues until all paths have been iterated through. At the end, the variables *highest\_idx* and *highest\_score* will contain the index and emotional score, respectively, of the path with the highest score per reference.

```
//----- (...) -----
//variables to store highest index path, its emotional score and average
int highest_idx = 0;
float highest_score = -1, best_path_average_score = 0;

//iterate all paths
for(int i = 0; i < paths.size(); ++i)
{
    int j, score_sum = 0;
    float path_average_score = 0;

    //iterate all the references in the path
    for(j = 0; j < paths[i].path.size(); ++j)
    {
        char reference = paths[i].path[j].first;
        //retrieve the corresponding emotional score or attribute a neutral score
        score_sum += (eltm->checkEntry(reference)) ? eltm->getEScore(reference) : 50;
    }

    //Compute the average for the path and then for each reference
    path_average_score = (score_sum/j);
    float score_per_ref = path_average_score/j;
    score_sum = 0;

    //keep track of the highest average score and its index
    if(score_per_ref > highest_score) {
        highest_score = score_per_ref;
        highest_idx = i;
    }
}
//----- (...) -----
```

Listing 45: Determining of the best path from the vector of possible paths.

The final section of the function, shown in Listing 46, starts by determining if the average score of the selected path is above a certain threshold. If not the case, it means that this path may negatively affect



the emotional state. Therefore, the action returns failure and a flag is set to signal the occurrence. This flag will indicate the decorator node above to re-execute the action node, and, as seen in Listing 44, at this instant, the possible paths generated by the map will be used to calculate the best path.

If the path is considered good, the action is retrieved from the first element of the selected path and scheduled for execution in the *Blackboard*.

```
//----- (...) -----  
//determine if the average score of the best path is above the threshold  
if((best_path_average_score<25) && !(std::any_cast<bool>(blackboard->getEntry("es_threshold"))))  
{  
    //set the flag to trigger path selection from map  
    blackboard->setEntry("es_threshold", true);  
    setStatus(NodeStatus::FAILURE, action_pusher);  
    return;  
}  
  
//get next action from selected path and update on BB  
Actions next_action = paths[highest_idx].path[0].second;  
blackboard->setEntry("next_action", next_action);  
  
//new action available and path's average score is above the threshold  
blackboard->setEntry("new_action", true);  
blackboard->setEntry("es_threshold", false);  
  
setStatus(NodeStatus::SUCCESS, action_pusher);  
}
```

Listing 46: Final decision on the best path and scheduling of the next action to take.

# V - Validation and Results

This chapter presents the testing and validation procedures conducted to assess the performance and functionality of the developed cognitive architecture (CA), with a particular focus on the internal affect subsystem. The primary objective of the testing phase is to not only evaluate its influence in the decision making, but to also validate the memory manipulation processes and the generation of intentions within the CA. The internal affect subsystem plays a pivotal role in these aspects by assigning emotional values to memories, enabling the system to prioritize decisions and make contextually appropriate choices. By evaluating the performance of the internal affect subsystem, the aim is to demonstrate its effectiveness in enhancing decision-making capabilities within the CA.

To conduct the tests, the complete CA was first structured using the GROOT IDE, resulting in an Extensible Markup Language (XML) file that can be found in Appendix C. This XML file was then loaded into the BT engine, enabling the execution of the CA. A BT monitoring tool was utilized to facilitate the validation tests and ensure accurate monitoring and tracking of the CA's execution, which through a ZeroMQ connection with the BT engine, allowed real-time tracking of the nodes return status. The complete CA being monitored in this tool may be consulted in Appendix A, and more information on the BT monitoring tool can be found in [41].

## 1. Physical Structure

To facilitate the validation process, a physical structure was used to embody the CA. As this prototype has been equipped with object detection and mobility capabilities, specifically designed to validate the physical world subsystem [41], it is deemed well-suited to validate the internal affect subsystem with minimal modifications.

The Raspberry Pi 4 serves as the primary processing unit to host the CA. To ensure optimal performance, the BT engine operates within a customized Buildroot [61] image specifically tailored for this application. It is equipped with the necessary packages and libraries to provide essential features for the prototype such as, OpenCV, I<sup>2</sup>C, Wi-Fi, and SSH, enabling image processing, wired communication and wireless control of the prototype. For more detailed information, refer to [38].

To ensure the CA takes advantage of the full processing power of the target platform, the physical structure is controlled by a separate control board, a Romi32U4. It performs tasks such as powering the

Raspberry Pi, acquiring data from the wheel quadrature encoders, controlling the wheel motors, and correcting possible deviations. The control board establishes a direct I<sup>2</sup>C connection with the Raspberry Pi, enabling it to receive movement commands from the execution module. These commands are executed by the control board, resulting in the appropriate rotations and forward movements, taking into account the standardized distance of 24cm between references within the environment. Figure 47 depicts a photograph of the prototype within the environment.

To validate the internal affect subsystem, the prototype, seen in Figure 48 and Appendix A, incorporates two additional sensors: a decibel sensor and a luminosity sensor. These sensors play a crucial role in providing the necessary data for the internal affect subsystem to compute hormonal concentrations. Since these sensors, SparkFun LMV324 and Light Dependent Resistor (LDR) module, operate on analog signals, an Analog-to-Digital Converter (ADC) is essential to convert the analog inputs from these sensors into valid digital inputs. The 16-bit ADC used in this setup, ADS1115 JOY-IT KY-053, facilitates the configuration and reading of converted data through I<sup>2</sup>C communication. It is worth noting that as the Buildroot image already includes the I<sup>2</sup>C package, which is necessary for communication with the control board, no additional packages were added. The complete list of components may be consulted in Appendix B.

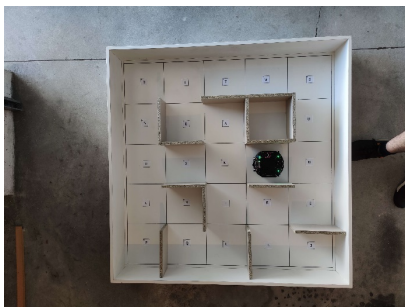


Figure 47: Environment used in the validation tests.

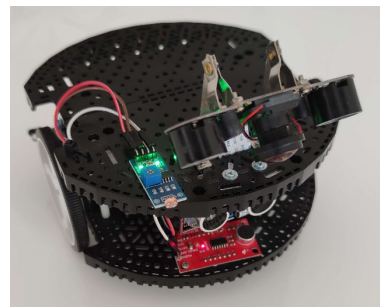


Figure 48: Prototype used in the validation tests.

## 2. Memory Manipulation Processes

To evaluate and validate the memory manipulation processes within the CA, a series of tests were conducted. These tests aimed to access the encoding, rehearsal, association, consolidation, and transfer processes in the emotional memory stores (eSTM and eLTM). Put differently, the goal was to determine if the internal affect subsystem could encode new information, maintain relevant information in the eSTM, link new information with existing knowledge, update existing knowledge with new information, and transfer information from the eSTM to the eLTM.

To conduct these tests a predetermined itinerary was set, as depicted in Figure 49. This was

accomplished by setting the starting reference of the agent to 'P' and explicitly coding its intention to first go to reference 'A' and then to reference 'P'. This sequence of references would cause the agent to repeat certain references, thereby triggering multiple memory manipulation processes. As a result, this test scenario was considered appropriate. Due to the absence of a more efficient method for displaying the contents of data variables and structures, the relevant data is printed via wireless SSH (using Wi-Fi) for each SPA cycle, and screenshots are provided accordingly.

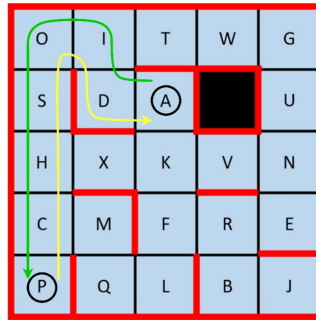


Figure 49: Testing itinerary to validate memory manipulation processes.

### Rehearsal

To validate the rehearsal process, the eSTM was printed in every SPA cycle. The most recent memory element was printed first, with the corresponding integer on the left representing its position in the array. As shown in Figure 49, the agent was positioned in the reference 'P', and its intention is to reach the reference 'A'. As the agent progresses through the references, each new reference is added as the most recent element in the eSTM. Rehearsal does not occur until a reference that already exists in the eSTM is encountered. When the agent reaches the reference 'A', its intention changes to reference 'P', and the agent starts revisiting references that are already in the eSTM, thus rehearsal is triggered. In Figure 50 (middle), the reference 'D' is rehearsed and becomes the most recent element. Similarly, in the same figure (right), the reference 'I', which was the third most recent, reappears and undergoes rehearsal, becoming the most recent element. It is important to mention that Figure 50, and subsequent figures, are divided into sections but represent a sequence.

```

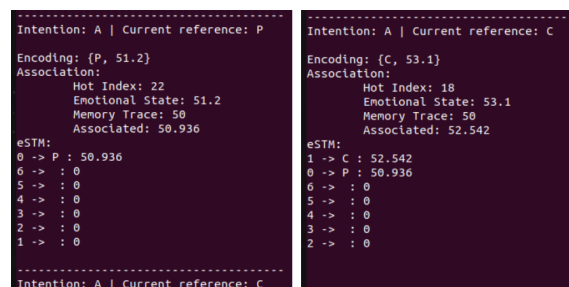
-----
Intention: P | Current reference: A
eSTM:
0 -> A : 64
6 -> D : 62.5
5 -> I : 60
4 -> O : 57
3 -> S : 54
2 -> H : 50.5
1 -> C : 46.5
-----
Intention: P | Current reference: D
Rehearsal: 'D'
eSTM:
0 -> D : 62.5
6 -> A : 64
5 -> I : 60
4 -> O : 57
3 -> S : 54
2 -> H : 50.5
1 -> C : 46.5
-----
Intention: P | Current reference: I
Rehearsal: 'I'
eSTM:
0 -> I : 60
6 -> D : 62.5
5 -> A : 64
4 -> O : 57
3 -> S : 54
2 -> H : 50.5
1 -> C : 46.5
-----
Intention: P | Current reference: D
Intention: P | Current reference: I

```

Figure 50: Rehearsal validation.

## Association

To validate the association process, the same itinerary was utilized. As the agent begins to move and references are added to the eSTM, they are searched in the eLTM. In Figure 51 (left), when the agent is located at the starting reference 'P', the memory trace associated with that reference has an emotional score of 50.0, and the corresponding physical memory possesses a hot index of 22. The resulting emotional score is determined, as discussed in the Design chapter, by the weighted average between the memory trace and the current emotional state. Being the weight for the memory trace given by the hot index. Similarly, when the agent progresses to the next reference, 'C', it is encoded and inserted into the eSTM, triggering an association.

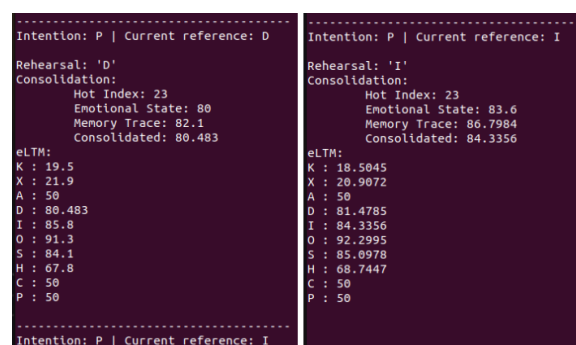


```
-----
Intention: A | Current reference: P
Encoding: {P, 51.2}
Association:
  Hot Index: 22
  Emotional State: 51.2
  Memory Trace: 50
  Associated: 50.936
eSTM:
0 -> P : 50.936
6 -> : 0
5 -> : 0
4 -> : 0
3 -> : 0
2 -> : 0
1 -> : 0
-----
Intention: A | Current reference: C
Encoding: {C, 53.1}
Association:
  Hot Index: 18
  Emotional State: 53.1
  Memory Trace: 50
  Associated: 52.542
eSTM:
1 -> C : 52.542
0 -> P : 50.936
6 -> : 0
5 -> : 0
4 -> : 0
3 -> : 0
2 -> : 0
-----
Intention: A | Current reference: C
```

Figure 51: Association validation.

## Consolidation

As the consolidation process occurs after rehearsal if a memory trace is present, the conditions for the test are similar to those used to validate rehearsal. When this situation arises, the physical hot index of the corresponding physical memory is retrieved to determine the weight of the memory trace in the weighted average. The new value of the emotional score is computed and updated in the eLTM. In Figure 52 (left), this process is depicted as the agent follows the same selected itinerary. The agent encounters the reference 'D' while it is still in the eSTM, and the same occurs when the agent encounters the reference 'I', on the right side of the figure below.



```
-----
Intention: P | Current reference: D
Rehearsal: 'D'
Consolidation:
  Hot Index: 23
  Emotional State: 80
  Memory Trace: 82.1
  Consolidated: 80.483
eLTM:
K : 19.5
X : 21.9
A : 50
D : 80.483
I : 85.8
O : 91.3
S : 84.1
H : 67.8
C : 50
P : 50
-----
Intention: P | Current reference: I
Rehearsal: 'I'
Consolidation:
  Hot Index: 23
  Emotional State: 83.6
  Memory Trace: 86.7984
  Consolidated: 84.3356
eLTM:
K : 18.5045
X : 20.9072
A : 50
D : 81.4785
I : 84.3356
O : 92.2995
S : 85.0978
H : 68.7447
C : 50
P : 50
-----
Intention: P | Current reference: I
```

Figure 52: Consolidation validation.

## Transfer

To validate the transfer process, the file containing the eLTM was deliberately removed from the directory. Consequently, during rehearsal, no memory trace was available to initiate consolidation, resulting in the activation of the transfer process instead. As shown in Figure 53, when returning through the same path after reaching the reference 'A', the memory elements correspondent to references encountered along the way are still present in the eSTM. This occurrence prompts rehearsal, ultimately leading to the transfer of these memory elements to the eLTM.

```
-----
Intention: A | Current reference: P
-----
Intention: A | Current reference: C
-----
Intention: A | Current reference: H
-----
Intention: A | Current reference: X
-----
Intention: A | Current reference: K
-----
Intention: P | Current reference: A
-----
Intention: P | Current reference: K

Rehearsal: 'K'
eSTM:
5 -> K : 25.7
4 -> A : 54.8
3 -> X : 26.4
2 -> H : 65.9
1 -> C : 53.8
0 -> P : 51.4
6 -> : 0

Transfer: {K, 25.7}
eLTM:
K : 25.7
-----
Intention: P | Current reference: X

-----
Intention: P | Current reference: X

Rehearsal: 'X'
eSTM:
5 -> X : 26.4
4 -> K : 25.7
3 -> A : 54.8
2 -> H : 65.9
1 -> C : 53.8
0 -> P : 51.4
6 -> : 0

Transfer: {X, 26.4}
eLTM:
X : 26.4
K : 24.7154
-----
Intention: P | Current reference: H

Rehearsal: 'H'
eSTM:
5 -> H : 65.9
4 -> X : 26.4
3 -> K : 25.7
2 -> A : 54.8
1 -> C : 53.8
0 -> P : 51.4
6 -> : 0

Transfer: {H, 65.9}
eLTM:
H : 65.9
X : 25.4177
K : 23.728
```

Figure 53: Transfer validation.

## Decay

As memory elements only decay with the addition of new knowledge to the eLTM, to validate the decay process either consolidation or transfer must occur. The same scenario utilized for the validation of the consolidation process is applied, however, in this case, the eLTM is printed before and after the decay occurs, enabling visual observation of the changes. In the testing scenario, when going from reference 'A' to reference 'P', rehearsal will be triggered in all references as it is moving through the same path. This will trigger either transfer or consolidation mechanisms based on the presence or absence of memory traces. As depicted in Figure 54, the memory trace for the references 'D' and 'I' are present. Consequently, the emotional scores of all other memory elements undergo positive or negative decay, as described in the Memory Model subsection in the Design chapter.

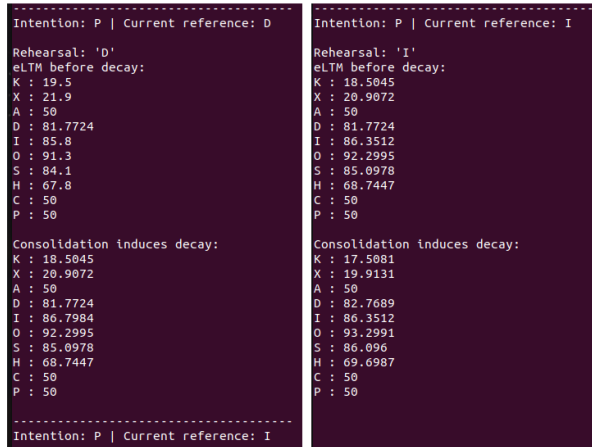


Figure 54: Decay validation.

### 3. Internal Affect Subsystem

To validate the influence of the internal affect subsystem on the decision-making process, a test scenario was designed. The sound intensity level in the environment was meticulously adjusted to maintain a neutral cortisol concentration of approximately 50. This deliberate setting allowed for the manipulation of the emotional score solely through the variation in light intensity.

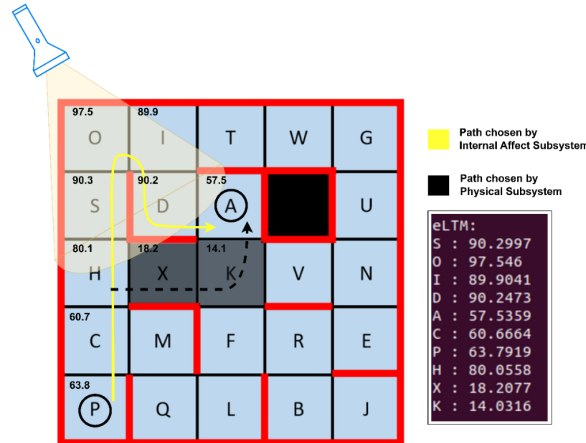


Figure 55: Testing scenario and itinerary to validate the influence in decision-making.

A test scenario was then developed, involving the positioning of a flashlight towards the left top corner, as illustrated in Figure 55. To create a low light intensity setting, the references 'X' and 'K' were deliberately covered. The agent was first intentionally set to move through the relevant references to retrieve the respective emotional scores for each of them and subsequently store them in the eLTM. The final values can be consulted in Figure 55.

Once the eLTM was established, the agent was positioned at reference 'P' with its intention hardcoded to

reach reference 'A'. This choice was intentional since the shortest path from 'P' to 'A' clearly involved passing through references 'X' and 'K'. However, as depicted in Figure 56, while the physical world subsystem, in reference 'H', scheduled an action to move right, the internal affect subsystem overrode this decision and changed the action to move upwards. This alternate path, indicated in yellow in Figure 55, was longer in terms of distance but offered significantly higher emotional scores per reference, making it a preferable route.

Intention: A   Current reference: P [Physical Subsys] Action Scheduled: UP [Internal Affect] Action Scheduled: UP [Exec] Action Executed: UP	Intention: A   Current reference: H [Physical Subsys] Action Scheduled: RIGHT [Internal Affect] Action Scheduled: UP [Exec] Action Executed: UP	Intention: A   Current reference: O [Physical Subsys] Action Scheduled: RIGHT [Internal Affect] Action Scheduled: RIGHT [Exec] Action Executed: RIGHT	Intention: A   Current reference: D [Physical Subsys] Action Scheduled: RIGHT [Internal Affect] Action Scheduled: RIGHT [Exec] Action Executed: RIGHT
Intention: A   Current reference: C [Physical Subsys] Action Scheduled: UP [Internal Affect] Action Scheduled: UP [Exec] Action Executed: UP	Intention: A   Current reference: S [Physical Subsys] Action Scheduled: UP [Internal Affect] Action Scheduled: UP [Exec] Action Executed: UP	Intention: A   Current reference: I [Physical Subsys] Action Scheduled: DOWN [Internal Affect] Action Scheduled: DOWN [Exec] Action Executed: DOWN	Intention: P   Current reference: A
Intention: A   Current reference: H	Intention: A   Current reference: O	Intention: A   Current reference: D	

Figure 56: Influence in decision-making validation.

In order to validate the motivational module, the same test scenario from before was utilized. However, this time the agent was positioned at reference 'A', and the initial intention was manually set to reference 'L'. The purpose of this setup was to ensure that the agent would pass through reference 'K', which is known to have a very low emotional score. As illustrated in Figure 57, as the agent reached reference 'K', its emotional state deteriorated, triggering the selection of a new drive aimed at improving the current emotional state. Consequently, the intention was automatically set to reference 'O', which is known to have a high emotional score. The agent then began its journey towards reference 'O', with its emotional state gradually improving along the way. Eventually, its emotional state improved to a point where it was no longer considered negative, and the drive transitioned back to the "Explore". This shift allowed for the automatic generation of a random reference as the new intention, enabling the agent to continue exploring its surroundings.

Intention: L   Current reference: A [Internal Affect] Emotional score: 55.75	Intention: O   Current reference: K [Internal Affect] Emotional score: 25.77
Intention: L   Current reference: K [Internal Affect] Emotional score: 21.6004 Negative Emotional State [New drive] Lift mood	Intention: O   Current reference: A [Internal Affect] Emotional score: 53.15
Intention: O   Current reference: F [Internal Affect] Emotional score: 46	Intention: O   Current reference: D [Internal Affect] Emotional score: 86.9849 Emotional State Improved [New drive] explore
Intention: O   Current reference: K	Intention: C   Current reference: I

Figure 57: Drive selection validation.

To validate the loud noise detection and subsequent "Avoid danger" drive selection, a different experimental setup was devised. Utilizing the same eLTM file, the agent was initially positioned at reference 'P' with a randomly generated intention of reference 'Q', as its drive is initially set to "Explore".



As the agent approached reference 'M', the volume of the speaker, which was maintaining neutral decibel values, was abruptly increased. This sudden increase surpassed the threshold defined in the decibel sensor, triggering the driver to send a signal to the condition node. As depicted in Figure 58, the emotional score associated with reference 'M' significantly decreased and the drive switches to "Avoid danger". Consequently, the agent's intention was redirected to a reference known to have a high emotional score, as a precautionary response to the detected danger.

```

-----
Intention: Q | Current reference: P
eSTM:
0 -> P : 56.0216
6 -> : 0
5 -> : 0
4 -> : 0
3 -> : 0
2 -> : 0
1 -> : 0
-----
Intention: Q | Current reference: C
eSTM:
1 -> C : 55.9052
0 -> P : 56.0216
6 -> : 0
5 -> : 0
4 -> : 0
3 -> : 0
2 -> : 0
Loud noise signal caught
[New drive] Avoid danger
-----
Intention: O | Current reference: M
eSTM:
2 -> M : 26
1 -> C : 55.9052
0 -> P : 56.0216
6 -> : 0
5 -> : 0
4 -> : 0
3 -> : 0
-----
Intention: O | Current reference: c
eSTM:
2 -> C : 55.9052
1 -> M : 26
0 -> P : 56.0216
6 -> : 0
5 -> : 0
4 -> : 0
3 -> : 0

```

Figure 58: Loud noise detection and avoid danger drive validations.

## 4. Discussion

The validation tests provide compelling evidence of (1) the complete functionality and effectiveness of the memory manipulation processes, (2) the significant influence of the internal affect subsystem on the decision-making process showcasing its pivotal role in shaping the agent's choices, and (3) the successful adaptation of the motivational module to the agent's emotional states, highlighting its capability to dynamically adjust intentions. Additionally, the successful performance of the threat detection mechanism further reinforced the CA 's responsiveness to external stimuli.

These validation tests, although focused primarily on validating components of the internal affect subsystem, as it was the main objective of the dissertation, also rely on the designed components for the BT engine. Furthermore, as these tests were conducted within the context of a fully integrated cognitive CA, they allowed for a thorough evaluation of both the integration of the subsystem within the broader architecture and its individual performance. By conducting the tests in this manner, a comprehensive assessment was achieved, ensuring a holistic understanding of the subsystem's capabilities and its impact on the overall CA.

# VI - Conclusions and Future Work

Although this dissertation presented contributions to the custom BT engine, the goal was to develop an internal affect subsystem structured in BTs. This subsystem, based on the SPA model of the brain, involved the modeling of emotions and the integration of emotional memory. To enable the subsystem to make contextually appropriate decisions and generate intentions based on the current emotional state of the agent, a practical application was devised, which consisted of navigating an agent from the current to a desired destination. Within this context, the subsystem aligned its choices with the well-being of the agent, prioritizing affectively positive routes, while dictating the drive of the agent through its current emotional state, therefore establishing its intended destination. This required integration with the physical world subsystem within an embedded environment, without compromising its reactivity to events.

The validation results, discussed in the Discussion subsection, demonstrated that the internal affect subsystem provides the CA with capabilities in decision-making and the ability to prioritize actions based on emotional value and memory recall. Even though the validation tests were performed in a controlled setting and a simplified environment, both the CA and the use case can be scaled to encompass higher complexity levels, with higher resemblance of real-world scenarios. Having that said, it is considered that the intended goal was successfully achieved. Nevertheless, there is room for improvement to enhance decision-making and take additional contextual factors into account. Possible future research directions could include:

- **Saliencies** – prioritize and filter sensory inputs, helping with focus on relevant stimuli in the environment. By quickly assessing the salience and its significance, emotions guide attention and perception. This would allow the CA to allocate cognitive resources more efficiently, resulting in improved decision-making and problem-solving.
- **Influence the recalling of physical memories** – emotional experiences are more likely to be encoded, retained, and retrieved compared to neutral experiences. The emotional content of an event enhances the formation of strong memories, improving recall and facilitating learning. This aspect of emotions would allow the CA to adapt and improve its performance over time by leveraging past experiences.
- **Additional sensory inputs** – explore the integration of additional sensory inputs beyond the existing ones. By incorporating more diverse sensory information, the CA could have a richer understanding of the environment, leading to more nuanced decision-making and behavior.

- **A more comprehensive and nuanced model of emotions** – the current model simplifies emotions by relying on the concentrations of cortisol and serotonin as proxies for stress, depression, confidence, and anxiety. Future research could explore more sophisticated models that consider a wider range of emotions and their interplay, allowing for a more accurate representation of human-like emotional experiences. This could involve integrating additional hormones or considering other psychological theories of emotions.

# References

- [1] W. F. Clocksin, "Memory and Emotion in the Cognitive Architecture," in *Intelligent Information Technologies: Concepts, Methodologies, Tools, and Applications*, 2008, pp. 1906-1918.
- [2] J. Vallverdú, M. Talanov, S. Distefano, M. Mazzara, A. Tchitchigin and I. Nurgaliev, "A cognitive architecture for the implementation of emotions in computing systems," in *Biologically Inspired Cognitive Architectures*, vol. 15, 2016, pp. 34-40.
- [3] P. Ekman and R. J. Davidson, "Affective Science: A Research Agenda," in *The Nature of Emotion*, Oxford University Press, 1994, pp. 411-430.
- [4] E. Hudlicka, "Beyond Cognition: Modeling Emotion in Cognitive Architectures," in *Sixth International Conference on Cognitive Modeling ICCM*, 2004.
- [5] P. de la Cruz, J. Piater and M. Saveriano, "Reconfigurable Behavior Trees: Towards an Executive Framework Meeting High-level Decision Making and Control Layer Features," in *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Toronto, Canada, 2020.
- [6] M. Colledanchise and L. Natale, "On the Implementation of Behavior Trees in Robotics," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, July 2021.
- [7] P. O. and M. C. , *Behavior trees in robotics and AI*, Taylor & Francis - CRC Press, 2018.
- [8] G. Flórez-Puga, M. A. Gómez-Martín, P. P. Gómez-Martín, B. Díaz-Agudo and P. A. González-Calero, "Query-Enabled Behavior Trees," *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, vol. 1, no. 4, December 2009.
- [9] M. Iovino, E. Scukins, J. Styrud, P. Ögren and C. Smith, "A Survey of Behavior Trees in Robotics and AI," 2020.
- [10] M. Colledanchise and P. Ogren, "How Behavior Trees Generalize," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Daejeon, 2016.
- [11] A. Marzinotto, M. Colledanchise, C. Smith and P. Ogren, "Towards a Unified Behavior Trees Framework for Robot Control," in *IEEE International Conference on Robotics & Automation (ICRA)*, Hong Kong, 2014.

- [12] M. Colledanchise and L. Natale, "Analysis and Exploitation of Synchronized Parallel Executions in," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.
- [13] R. Wray, M. van Lent, J. Beard and P. Brobst, "The Design Space of Control Options for AIs in Computer Games," in *Reasoning, Representation, and Learning in Computer Games*, Edinburgh, 2005.
- [14] V. Sklyarov, "Hierarchical Finite-State Machines and," *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, vol. 7, no. 2, June 1999.
- [15] A. Lieto, *Cognitive Design for Artificial Minds*, London: Routledge, 2021.
- [16] J. R. Anderson, *How Can the Human Mind Occur in the Physical Universe?*, Oxford: Oxford University Press, 2007.
- [17] J. E. Laird, "Extending the Soar Cognitive Architecture".
- [18] J. E. Laird, A. Newell and P. S. Rosenbloom, *Soar: An Architecture for General Intelligence*, Stanford, 1986.
- [19] F. E. Ritter, F. Tehrani and J. D. Oury, "ACT-R: A cognitive architecture for modeling cognition," *WIREs Cognitive Science*, vol. 10, no. 3, 2018.
- [20] K. Thórisson and H. Helgasson, "Cognitive Architectures and Autonomy: A Comparative Review," *Journal of Artificial General Intelligence*, vol. 3, no. 2, pp. 1-30, 2012.
- [21] S. Chipman, *The Oxford Handbook of Cognitive Science*, Oxford University Press, 2017.
- [22] S. Hélie, N. Wilson and R. Sun, "The CLARION Cognitive Architecture: A Tutorial," *Proceedings of the Annual Meeting of the Cognitive Science Society*, 2008.
- [23] R. Sun, "The importance of cognitive architectures: an analysis based on CLARION," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 19, no. 2, pp. 159-193, 2007.
- [24] A. Sloman and R. L. Chrisley, "More things than are dreamt of in your biology: Information-processing in biologically inspired robots," in *Cognitive Systems Research*, vol. 6, R. Sun, Ed., 2005, pp. 145-174.
- [25] P. D. Levy, "Beyond kansei engineering : the emancipation of kansei design," *International Journal*

*of Design*, vol. 7, no. 2, pp. 83-94, 2013.

- [26] M. A. Arbib and J.-M. Fellous, "Emotions: from brain to robot," in *TRENDS in Cognitive Sciences*, vol. 8, 2004, pp. 554-560.
- [27] G. L. Clore and A. Ortony, "The Semantics of the Affective Lexicon," in *Cognitive Perspectives on Emotion and Motivation*, vol. 44, Springer, Dordrecht, 1988, pp. 367-397.
- [28] R. W. Picard, "Recognizing and Expressing Affect," in *Affective Computing*, The MIT Press, pp. 165-192.
- [29] H. Lövheim, "A new three-dimensional model for emotions and monoamine neurotransmitters," in *Medical Hypotheses*, vol. 78, 2012, pp. 341-348.
- [30] K. Scherer, "What are emotions? and how can they be measured?," *Social Science Information*, vol. 44, no. 4, pp. 695-729, 2005.
- [31] R. Plutchik, "The Nature of Emotions," in *American Scientist*, vol. 89, 2001, pp. 344-350.
- [32] A. Tambini, U. Rimmele, E. A. Phelps and L. Davachi, "Emotional brain states carry over and enhance future memory formation," *Nature Neuroscience*, vol. 20, p. 271–278, 2017.
- [33] J. E. Dunsmoor, V. P. Murty, L. Davachi and E. A. Phelps, "Emotional learning selectively and retroactively strengthens memories for related events," *Nature*, vol. 520, p. pages345–348, 2015.
- [34] S. E. Alger and J. D. Payne, "The differential effects of emotional salience on direct associative and relational memory during a nap," *Cognitive, Affective, & Behavioral Neuroscience*, vol. 16, p. 1150–1163, 2016.
- [35] A. H. U. S. M. N. M. M. A. S. Tyng Chai M., "The Influences of Emotion on Learning and Memory," *Frontiers in Psychology*, vol. 8, 2017.
- [36] L. J. L. a. S. L. Burgess, "Beyond General Arousal: Effects of Specific Emotions on Memory," *Social Cognition*, vol. 15, no. 3, 2011.
- [37] "BehaviorTree.CPP," [Online]. Available: <https://github.com/BehaviorTree/BehaviorTree.CPP>. [Accessed September 2021].
- [38] "GROOT," [Online]. Available: <https://www.behaviortree.dev/groot/>. [Accessed 6 September

2022].

- [39] "TinyXML2," [Online]. Available: <https://github.com/leethomason/tinyxml2>. [Accessed 2022].
- [40] "BehaviorTree.CPP Documentation," [Online]. Available: <https://www.behaviortree.dev/docs/intro>. [Accessed 2022].
- [41] J. Silva, *BT-Enabled Cognitive Architecture: Physical World Sensing-Processing-Acting Cycle Subsystem*, 2023.
- [42] "PugiXML," [Online]. Available: <https://github.com/zeux/pugixml>. [Accessed September 2022].
- [43] "XML Benchmarkings," [Online]. Available: <https://pugixml.org/benchmark.html>. [Accessed September 2022].
- [44] "PugiXML Reference Manual," [Online]. Available: <https://pugixml.org/docs/manual.html>. [Accessed 2022].
- [45] "C++ Container reference," [Online]. Available: <https://en.cppreference.com/w/cpp/container>. [Accessed 2023].
- [46] H. Hinnant, "shared\_mutex Reference Implementation," 09 09 2007. [Online]. Available: [https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2406.html#shared\\_mutex\\_imp](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2406.html#shared_mutex_imp).
- [47] "Notes on readers writers in C++17," [Online]. Available: <https://cslab.pepperdine.edu/warford/cosc450/cosc-450-Notes-on-Readers-Writers-in-C++17.pdf>. [Accessed 2022].
- [48] R. Atkinson and R. Shiffrin, "Human memory: A proposed system and its control processes," in *The psychology of learning and motivation: Advances in research and theory*, vol. 2, K. Spence and J. T. Spence, Eds., New York: Academic Press, 1968, pp. 89-195.
- [49] R. Shiffrin and R. Atkinson, "Storage and Retrieval Processes in Long-Term Memory," in *Psychological Review*, vol. 76, 1969, pp. 179-193.
- [50] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information.," in *Psychological review*, vol. 63, 1956, pp. 81-97.

- [51] N. Cowan, Z. Chen and J. Rouder, "Constant Capacity in an Immediate Serial-Recall Task: A Logical Sequel to Miller (1956)," *Psychological Science*, vol. 15, no. 9, pp. 634-640, 2004.
- [52] D. J. F. W. N. Carmen L. A. Zurbriggen, "Rosy or Blue? Change in Recall Bias of Students' Affective Experiences During Early Adolescence," *Emotion*, vol. 21, no. 8, p. 1637–1649, 2021.
- [53] J. Ceraso, "The Interference Theory of Forgetting," *Scientific American*, vol. 217, no. 4, pp. 117-127 , 1967.
- [54] G. P. Chrousos, "Stress and disorders of the stress system," *Nature Reviews Endocrinology*, vol. 5, p. 374–381, 2009.
- [55] J. Herbert, "Cortisol and depression: three questions for psychiatry," *Psychological Medicine*, vol. 43, no. 3, pp. 449-469, 2013.
- [56] M. N. Shalaby , "The Determinants of Leadership: Genetic, Hormonal, Personality Traits Among Sport Administrators," *International Journal of Pharmaceutical and Phytopharmacological Research*, vol. 7, no. 5, pp. 9-14, 2017.
- [57] H. M. v. P. ,. S. W. M. A. G. B. Rene S. Kahn, "Serotonin and Anxiety Revisited," *Biological Psychiatry*, vol. 23, pp. 189-208, 1988.
- [58] "Resource Acquisition Is Initialization (RAII)," [Online]. Available: <https://en.cppreference.com/w/cpp/language/raii>. [Accessed 2022].
- [59] "DJB2's first mention in comp.lang.c," 1991. [Online]. Available: <https://groups.google.com/g/comp.lang.c/c/ISKWXiuNOAk>. [Accessed 2023].
- [60] "Hash Functions," [Online]. Available: <http://www.cse.yorku.ca/~oz/hash.html>. [Accessed 2022].
- [61] "Buildroot," [Online]. Available: <https://buildroot.org/>. [Accessed January 2023].



# Appendix A

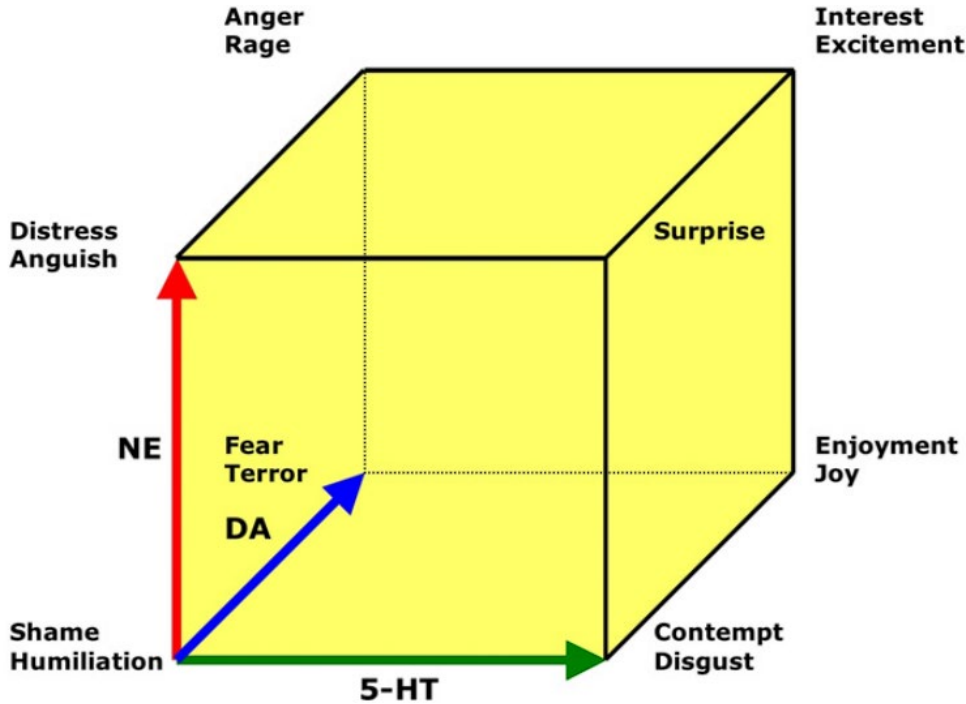


Figure A.1: Lövheim's Cube of Emotion [27] (return to Figure 12).

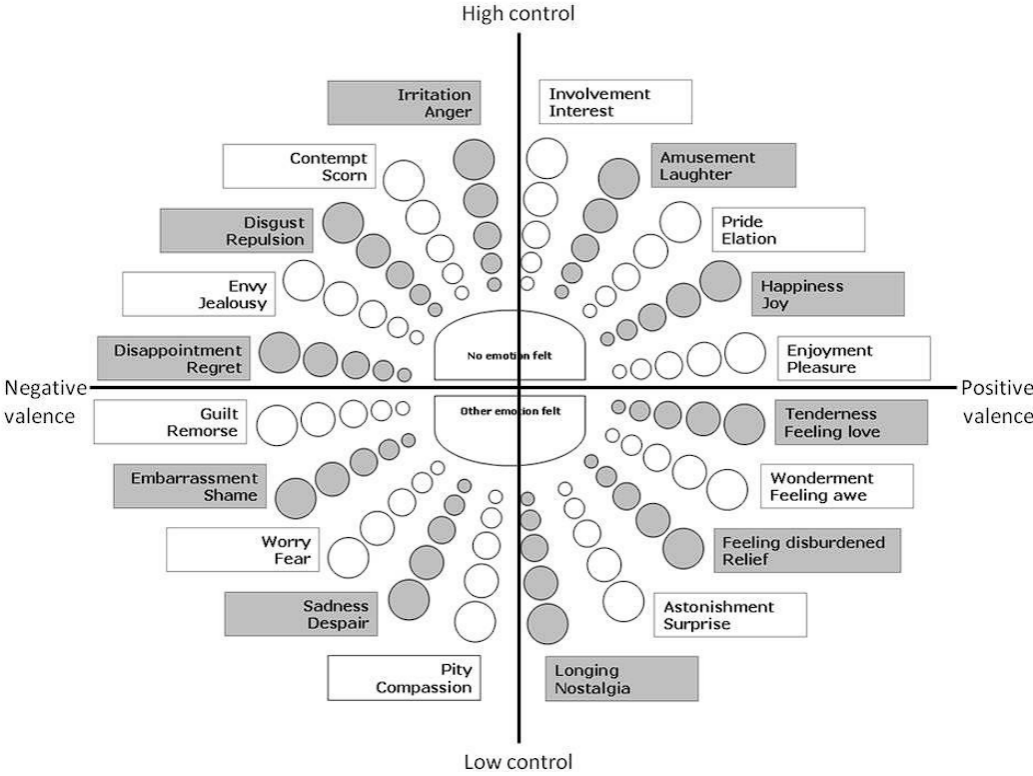


Figure A.2: Geneva emotion wheel [28] (return to Figure 12).

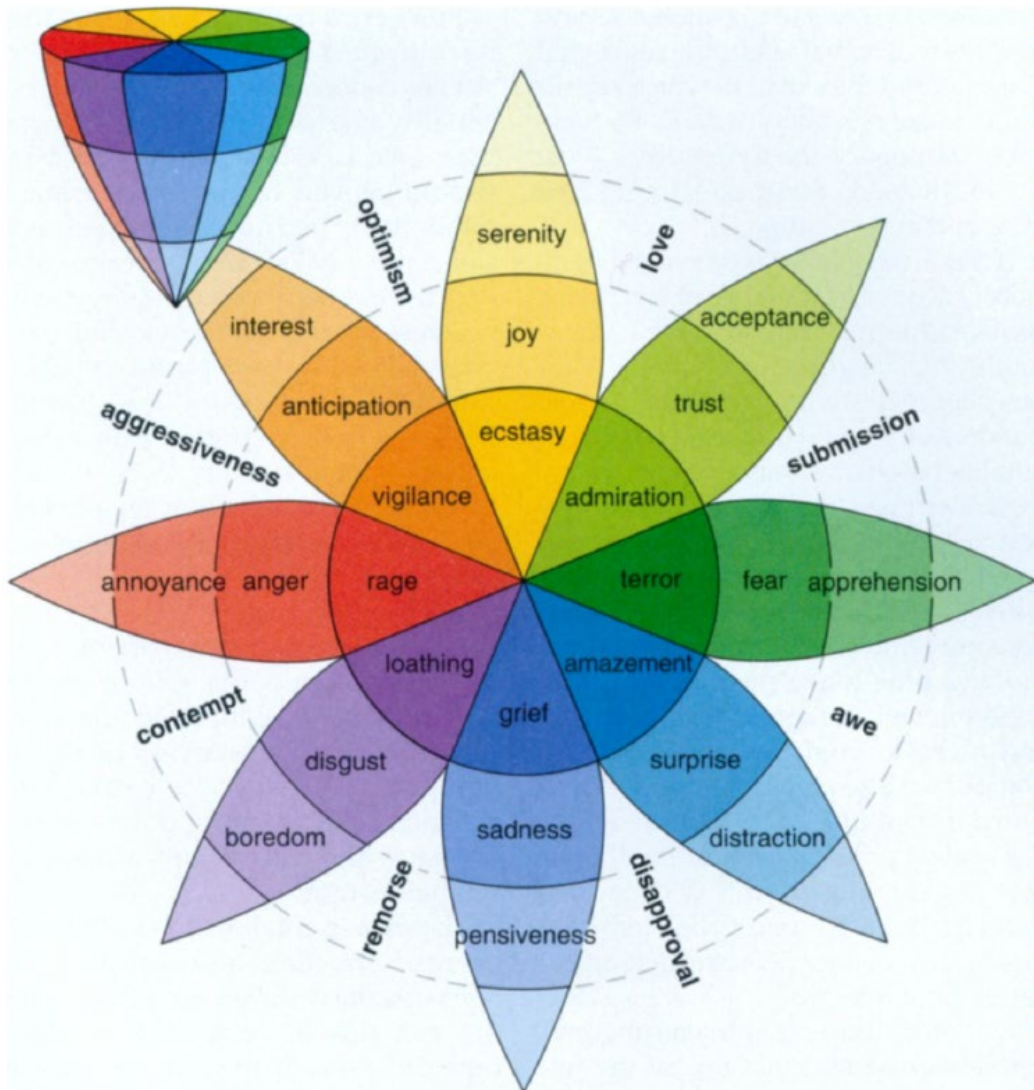


Figure A.3: Plutchik's Wheel of Emotions [29] (return to Figure 12).

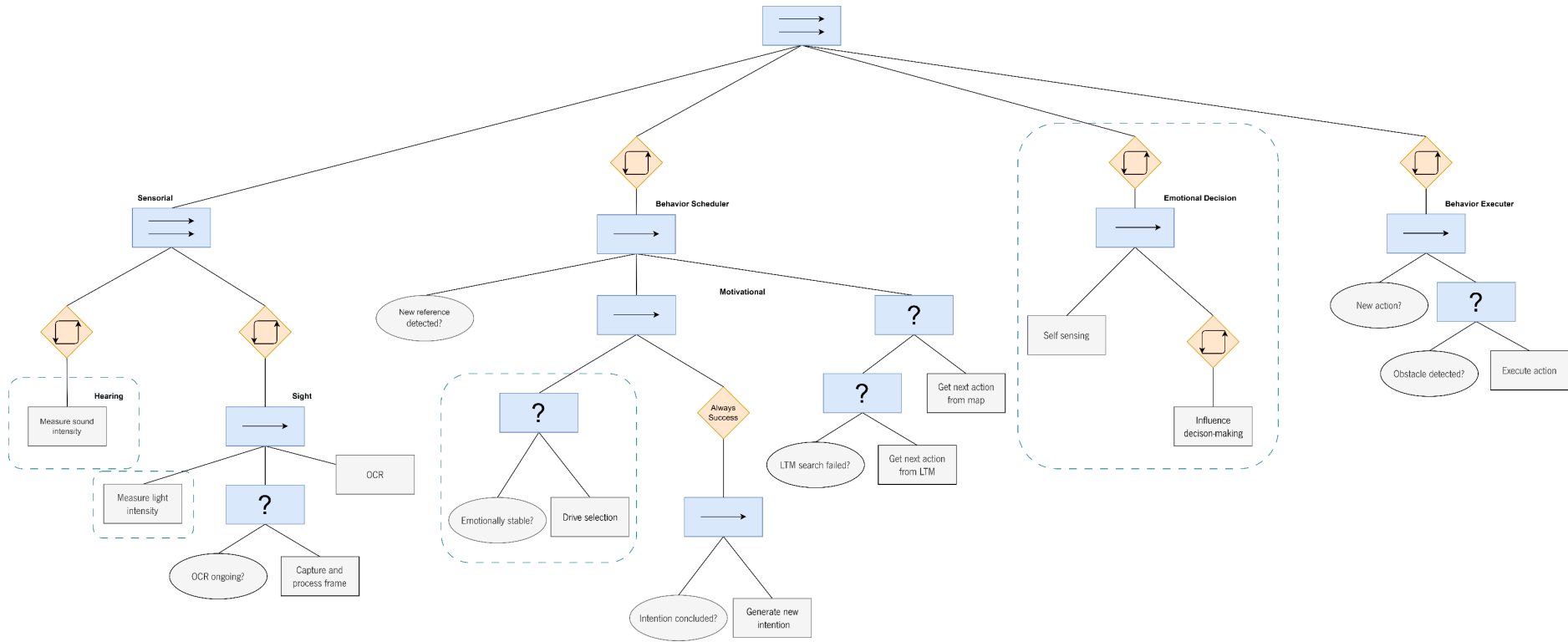


Figure A.4: Designed cognitive architecture structured in BTs (return to Figure 38).

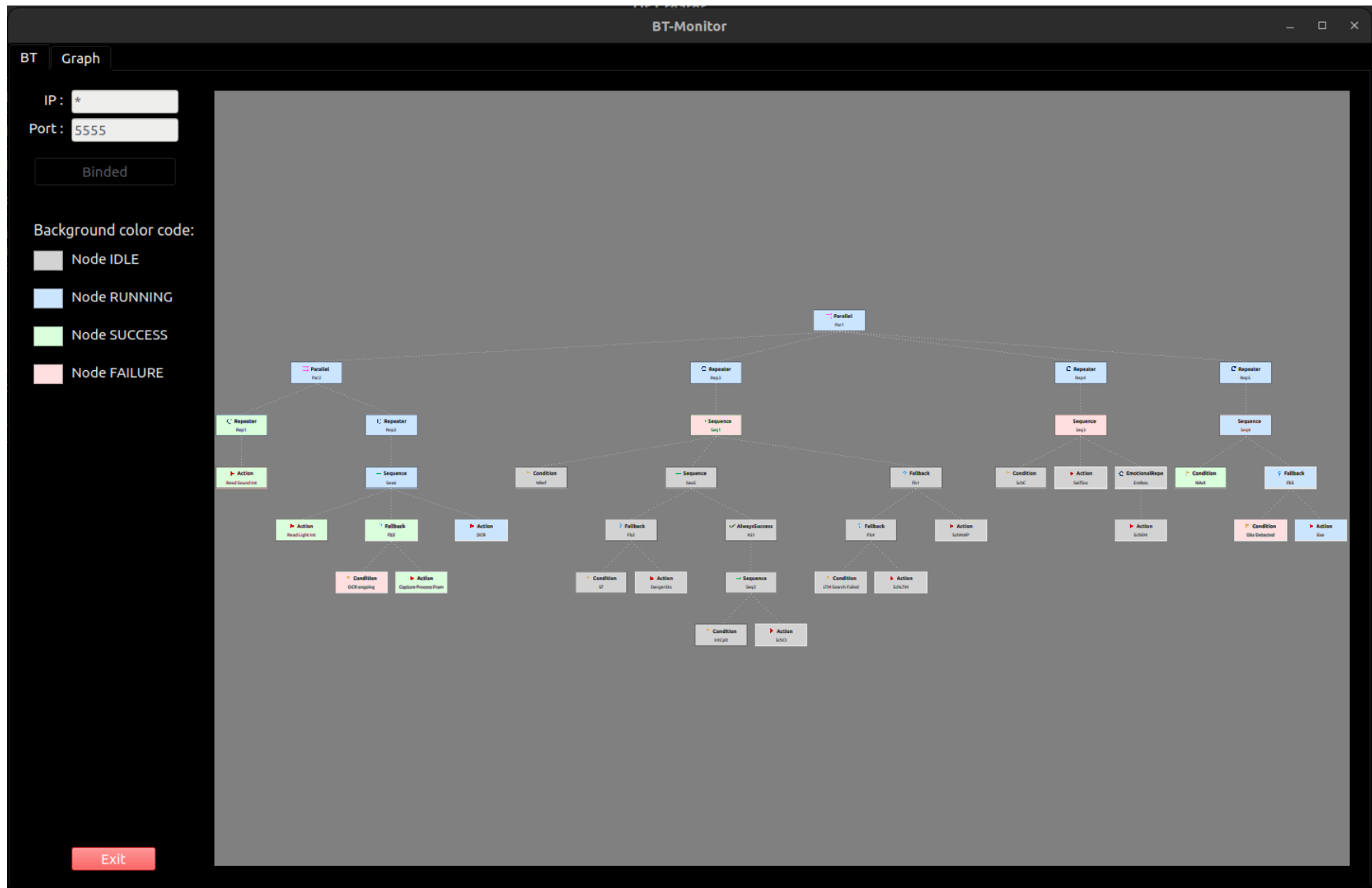


Figure A.5: Cognitive architecture execution being monitored with BT monitoring tool (return to Validation and Results).

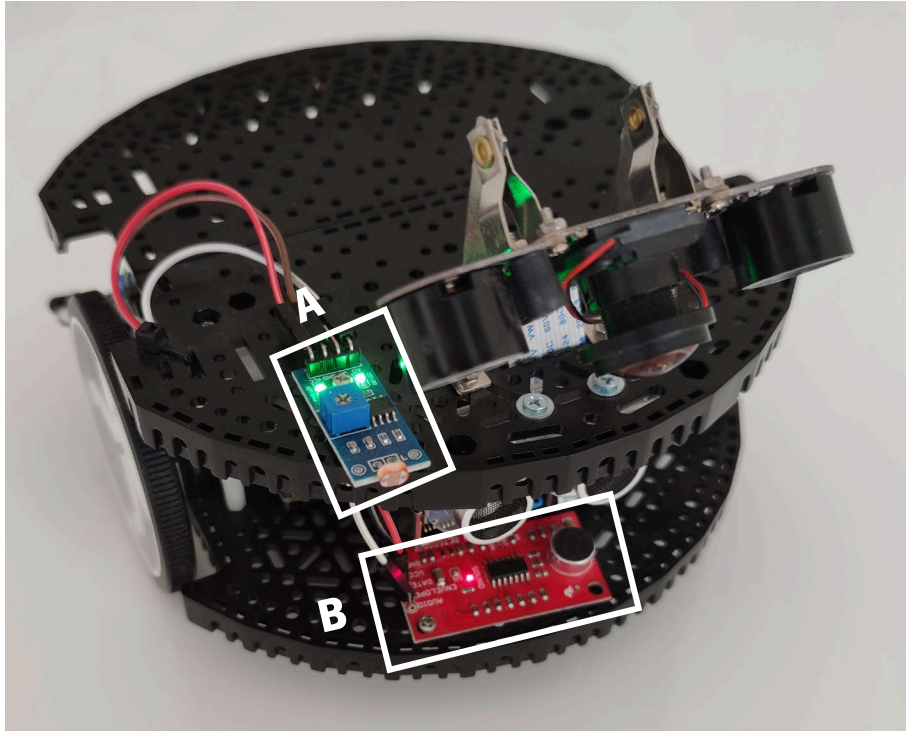


Figure A.6: Luminosity (A) and decibel (B) sensors added to the prototype (return to Physical Structure).

# Appendix B

Table B.1: List of components used in the prototype (return to Physical Structure).

<b>Category</b>	<b>Component</b>	<b>Quantity</b>
Main processing unit	Raspberry Pi 4 Model B (4GB)	1
Secondary processing unit	Romi 32U4 Control Board	1
Sensors	MakerHawk Fisheye Camera	1
	HC-SR04 Ultrasonic Sensor	1
	Infrared Funduino Sensor	2
	<u>SparkFun LMV324 Sound Sensor</u>	1
	<u>LDR Light Sensor Module</u>	1
	Romi Magnetic Quadrature Encoder	2
Converter	<u>ADS1115 JOY-IT KY-053 16-bit ADC</u>	1
Structure	Romi Chassis Kit	1
	Romi Chassis Expansion Plate	1

# Appendix C

```
<?xml version="1.0"?>
<root main_tree_to_execute="BehaviorTree">
  <!-- ////////// -->
  <BehaviorTree ID="BehaviorTree">
    <Control ID="ParallelNode" name="Par1">
      <Decorator ID="Repeater" name="Rep1">
        <Control ID="SequenceNode" name="Seq1">
          <Condition ID="NewRef" name="NRef"/>
          <Control ID="FallbackNode" name="FB2">
            <Control ID="FallbackNode" name="Fb2">
              <Condition ID="Safe" name="Sf"/>
              <Action ID="DangerStatus" name="DangerSts"/>
            </Control>
            <Decorator ID="AlwaysSucceed" name="AS1">
              <Control ID="SequenceNode" name="Seq2">
                <Condition ID="IntCompleted" name="IntCplt"/>
                <Action ID="SchedCalcInt" name="SchCI"/>
              </Control>
            </Decorator>
          </Control>
          <Control ID="FallbackNode" name="Fb1">
            <Action ID="SchedLTM" name="SchLTM"/>
            <Action ID="SchedLBBT" name="SchLBBT"/>
          </Control>
        </Control>
      </Decorator>
    <Decorator ID="Repeater" name="Rep2">
      <Control ID="SequenceNode" name="Seq3">
        <Condition ID="SchedCompleted" name="SchC"/>
        <Action ID="SelfSystem" name="SelfSys"/>
        <Decorator ID="EmotionalRepeater" name="EmRep">
          <Action ID="SchedEM" name="SchEM"/>
        </Decorator>
      </Control>
    </Decorator>
    <Decorator ID="Repeater" name="Rep3">
      <Control ID="SequenceNode" name="Seq4">
        <Condition ID="NewAction" name="NAct"/>
        <Action ID="Exec" name="Exe"/>
      </Control>
    </Decorator>
  </Control>
</BehaviorTree>
</root>
```

Listing C.1: Cognitive architecture XML representation (return to Validation and Results).