TRABALHO DE GRADUAÇÃO

SIMULATION AND EXECUTION OF DYNAMIC BEHAVIOR TREES IN THE SERVICE ROBOTS CONTEXT

GABRIEL F P ARAÚJO

Brasília, Novembro de 2021

ENGENHARIA MECATRÔNICA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

SIMULATION AND EXECUTION OF DYNAMIC BEHAVIOR TREES IN SERVICE ROBOTS CONTEXT

SIMULAÇÃO E EXECUÇÃO DE ÁRVORES DE COMPORTAMENTO DINÂMICAS NO CONTEXTO DE ROBÔS DE SERVIÇOS

GABRIEL F P ARAÚJO

ORIENTADOR(A): GENAÍNA NUNES RODRIGUES, DR.

TRABALHO DE GRADUAÇÃO EM ENGENHARIA
MECATRÔNICA

BRASÍLIA/DF: NOVEMBRO - 2021

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

SIMULATION AND EXECUTION OF DYNAMIC BEHAVIOR TREES IN THE SERVICE ROBOTS CONTEXT

GABRIEL F P ARAÚJO

TRABALHO DE GRADUAÇÃO SUBMETIDA AO DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE CONTROLE E AUTOMAÇÃO.

APROVADA POR:

BRASÍLIA, 05 DE NOVEMBRO DE 2021.

FICHA CATALOGRÁFICA

ARAÚJO, GABRIEL F. P.
Simulation and Execution of Dynamic Behavior Trees in the Service Robots Context [Distrito Federal] 2021.
xvii, 56p., 210 x 297 mm (CiC/FT/UnB, Engenheiro de Controle e Automação, Engenharia Mecatrônica, 2021).
TRABALHO DE GRADUAÇÃO – Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Ciência da Computação

1. Robôs de Serviços	2. Sistemas ciberfísicos
3. Software de Controle de Robôs	4. Árvores de Comportamento
I. CiC/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ARAÚJO, G. F. P. (2021). Simulation and Execution of Dynamic Behavior Trees in the Service Robots Context . TRABALHO DE GRADUAÇÃO em Engenharia Mecatrônica, Publicação , Departamento de Ciência da Computação, Universidade de Brasília, Brasília, DF, 56p.

CESSÃO DE DIREITOS

AUTOR: Gabriel F P Araújo
TÍTULO: Simulation and Execution of Dynamic Behavior Trees in the Service Robots Context .
GRAU: Engenheiro de Controle e Automação ANO: 2021

Gabriel F P Araújo

Departamento de Ciência da Computação (CiC) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

To myself

ACKNOWLEDGMENTS

"Tenho muito a agradecer a muita gente" esse foi o pensamento que passou pela minha cabeça todas as vezes que pensei no que deveria escrever nessa seção de agradecimentos. Chegar aqui com certeza não foi uma tarefa fácil, entrei na UnB no primeiro semestre de 2012 e foi uma caminhada deveras lenta e particular e eu não a teria feito sem ajuda. Espero aqui não esquecer de ninguém e se assim fiz, minhas mais sinceras desculpas. Depois de passar quase 10 anos na graduação, 18 semestres no papel e 1 desligamento e ainda assim concluir o curso não soa como uma prazeroza vitória, mas sim como o final de uma guerra onde não importa mais quem ganhou ou perdeu, só importa que acabou – "ao vencedor, as batatas".
Não chego aqui com um sorriso no rosto, mas com lágrimas nos olhos. Não me entenda mal, não é que eu não esteja feliz ou fazendo descaso do momento, mas o a pilha de emoções e sentimentos acumulados até aqui não me deixa esquecer os instantes de sofrimento e loucura – por mais que eu queira esquecê-los – hoje essa bagagem faz parte de mim. Contudo, assim como a vida não é feita de só de dias felizes, esse momento não precisa ser só alegria, pode ser, também, várias outras coisas. Então, meus agradecimentos vão muito além do que posso colocar em palavras.
"Como começar esses agradecimentos?" é uma pergunta difícil, então a farei do jeito mais mesquinho e egocêntrico possível: primeiramente agradeço a mim mesmo por estar vivo até hoje, por ter acreditado no meu próprio sonho quando eu mesmo achei que não deveria. Também agradeço aos meus pais, Evaldo e Maria de Lourdes, não porque me colocaram no mundo, mas por terem me criado e educado da melhor maneira que eles puderam; por todo esse amor: muito obrigado! À minha irmã, Joana Patrícia, agradeço por todos esses anos de parceria, brigas e brincadeiras.
Um obrigado especial, também, a todos os professores que me ajudaram durante essa caminhada, antes mesmo da UnB. Dentro da UnB, tive a oportunidade de conhecer pessoas incríveis. O Laboratório de Automação e Robótica foi um lugar extremamente importante na minha formação, onde pude crescer e aprender ao lado de pessoas muito inteligentes, fica aqui meu obrigado a: George Brindeiro, Waldez Gomes, Guilherme Anselmo, Tiago Pimentel, Marcos Pereira, Rafael Ramos, Bruno Calazans, Caio Eduardo, Thiago Rocha, Geovany Borges, Roberto Baptista. Além da sala de aula, pude encontrar amizades verdadeiras na Salsa UnB, Marina Ohana e Ana Moreno Diniz.

No Laboratório de Engenharia de Software, também, conheci pessoas maravilhosas, Gabriel Rodrigues, Vicente Moraes, Eric Gil, Genaína Rodrigues, muito obrigado pela hospitalidade e pelo nosso trabalho aqui documentado. A família Dias que cuidou muito de mim nesses anos aqui em Brasília, Vovó Paula, Tia Paulinha, Tia Claudinha, Tio Weliton e Tia Lucimar. À todos esses, uma grandioso obrigado.
Brandon Dias, Bruno Macedo, Saulo Cardoso, Marina Moreira, Ana Cláudia, Ricardo Diniz e Gabriel Sousa fazem parte de tantas histórias minhas que é impossível agradescer e espero que saibam que serão para sempre lembrados em meu coração. Por fim, eu preciso agradecer ao meu psicólogo, Pedro Martini, que me acompanhou nesses últimos anos e me ajudou não só por em ordem a minha vida e prioridades, mas também a minha cabeça, pois saiba que sem seu trabalho com certeza eu não estaria escrevendo essa seção de agradecimentos.
É difícil listar e agradecer essas ruma de gente e sinceramente eu gostaria de fazê-lo pessoalmente, pois cada pessoa mencionada aqui teve um impacto na minha vida que não tenho como descrever. Obrigado... Obrigado. Obrigado!!

Gabriel F P Araújo

RESUMO

Título: Simulação e Execução de Árvores de Comportamento Dinâmicas no Contexto de Robôs de Serviços
Autor: Gabriel F P Araújo
Orientador(a): Genaína Nunes Rodrigues, Dr.
Brasília, 05 de novembro de 2021

A robótica alcançou muito sucesso na fabricação industrial. Manipuladores têm sido usados em linhas de montagem por sua velocidade, exatidão e precisão. Embora esse mercado deva movimentar US $ 75,3 bilhões até 2026, há um mercado crescente para robôs cada vez mais flexíveis. Estima-se que esse mercado receba US $ 102,5 bilhões até 2025. Os objetivos sociais e econômicos moldam a nova realidade em que os robôs irão operar e, para cumprir suas tarefas, esses robôs requerem mobilidade e a capacidade de cooperar com outros robôs. Servicerobots são robôs projetados para atuarem entre humanos em ambientes como hospitais, logística e transporte de forma autônoma. A ISO 8373: 2012 define que tais robôs devem "realizar tarefas úteis para humanos ou equipamentos, excluindo aplicações de automação industrial" e tem a "capacidade de realizar tarefas pretendidas com base no estado atual e detecção, sem intervenção humana". Vários setores, como comércio eletrônico e manufatura, já mostram sinais de aumento da demanda por robôs de serviço. Os algoritmos de planejamento clássicos freqüentemente presumem que o mundo é estático e conhecido, com todas as mudanças ocorrendo como resultado da ação executada por um agente controlado. No entanto, isso não é verdade para robôs de serviço, e esses robôs precisam lidar com ambientes estruturados, embora populados e dinâmicos. Para preencher a lacuna entre os algoritmos clássicos de planejamento de tarefas e os robôs de serviço recém-chegados, devemos implementar uma estrutura para gerenciar ações e comportamentos robóticos para executar um plano de alto nível. Propomos Árvores de Comportamento Dinâmico para coordenar as habilidades do robô e executar um plano neste trabalho. As Árvores de Comportamento Dinâmico também são responsáveis por gerenciar componentes do robô como o sistema de navegação e as interfaces homem-máquina. Nós avaliamos o framework em uma simulação em um ambiente hospitalar. Temos uma Enfermeira que solicita uma tarefa de entrega, o robô é designado e recebe um plano para realizar a tarefa. O robô deve seguir o plano executando suas habilidades. Se ocorrer alguma exceção, o robô deve enviar um relatório ao planejador. Nossa estrutura aborda o problema de interface entre as funcionalidades deliberativas e reativas em um sistema híbrido. O framework proposto mostra como um sistema reativo lida e segue uma entrada deliberativa usando Árvores de Comportamento Dinâmico. A estrutura também mostra como as Árvores de Comportamento Dinâmico encapsulam tudo o que é necessário para executar uma determinada tarefa.

Palavras-chave: Robôs de Serviços, Sistemas ciberfísicos, Software de Controle de Robôs, Árvores de Comportamento.

ABSTRACT

Title: Simulation and Execution of Dynamic Behavior Trees in Service Robots Context
Author: Gabriel F P Araújo
Supervisor: Genaína Nunes Rodrigues, Dr.
Brasília, November 05th, 2021

Robotics has achieved much success in industrial manufacturing. Manipulators have been used in assembly lines for their speed, accuracy, and precision. Even though this market should be worth US$ 75.3 billion by 2026, there is a growing market towards robots even more flexible. This market is estimated to get US$ 102.5 billion by 2025. Social and economic purposes shape the new reality where robots will operate, and to fulfill their jobs, these robots require mobility and the ability to cooperate with other robots. Service robots are robots designed to act among humans in environments such as hospitals, logistics, and transportation in an autonomous manner. The ISO 8373:2012 defines that such robots should "perform useful tasks for humans or equipment excluding industrial automation applications" and has the "ability to perform intended tasks based on current state and sensing, without human intervention". Several sectors like e-commerce and manufacturing already show signs of increased demand for Service Robots. Classical planning algorithms often assume the world to be static and known, with all changes occurring as a result of the actions executed by one controlled agent. However, this is not true for service robots, and these robots have to tackle structured though populated and dynamic environments. To bridge the gap between classical task planning algorithms and the newly arrived Service Robots, we aim to implement a framework to manage robotic actions and behaviors to execute a high-level plan. We propose Dynamic Behavior Trees to coordinate the robot's skills and execute a plan in this work. The Dynamic Behavior Trees are also responsible for managing robot components like the navigation system and the human-machine interfaces. We evaluated the framework in a simulation in a hospital environment. We have a Nurse that requests a delivery task, the robot is assigned and receives a plan to conduct the task. The robot has to follow the plan executing its skills. If any exception occurs, the robot must send a report to the planner. Our framework tackles the problem of interfacing deliberative and reactive functionalities in a hybrid system. The proposed framework shows how a reactive system handles and follows a deliberative input using Dynamic Behavior Trees. The framework also shows how Dynamic Behavior Trees encapsulate everything needed to execute a given task.

Keywords: Service Robots, Cyber-phisical systems, Robot Control Software, Behavior Trees.

SUMMARY

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

LIST OF SYMBOLS

Greek symbols

π	Plan or policy	
δ(**s, e**)	State transition function	
θ	Robot orientation around the z axis	[rad]

Latin symbols

C	Battery discharge rate	[%/s]
E	Set of events	
e	Event	
$\mathbf{L}_{batt}$	Battery energy level	[%]
S	Set of states	
s	Robot state	
x	Robot position in the forward direction in relation of the rigid-body	[m]
y	Robot position in the left direction in relation of the rigid-body	[m]
z	Robot position in the up direction in relation of the rigid-body	[m]

Dimensionless symbols

$(a, b, ..., u)$	Tuple
$\langle a, b, ..., u\rangle$	Sequence

LIST OF ACRONYMS AND ABBREVIATIONS

3T	3-Tiered Architecture. 17
ASP	Answer Set Programming. 3
BLE	Blender Game Engine. 15, 16, 32
BT	Behavior Tree. x, 3–6, 8, 10–12, 15–22, 24, 25, 27, 34, 35, 44, 45
DBT	*Dynamic* Behavior Tree. 3, 4, 17, 21, 44, 45
FSA	Finite State Automata. 9
FSM	Finite State Machine. 5, 10, 17
LIDAR	Light Detection and Ranging. 13, 23, 26, 31, 38
MAR	Robotics 2020 Multi-Annual Roadmap For Robotics in Europe. 2
NPC	Non-Player Characters. 10
P,S-A	Plan,Sense-Act. 6, 7
PDDL	Planning Domain Definition Language. 3
ROS	Robot Operating System. 8, 12–17, 23, 25, 26, 30–33, 35, 36, 42
ROS2	Robot Operating System 2. 13–16, 35, 36, 42
VM	Virtual Machine. 35

NOTATION

`SENSE-PLAN-ACT` The hierarchical paradigm where S-P-A sequence is followed. 7
3-Tiered Architecture A hybrid control architecture for mobile robots developed at NASA. 3, 6, 7, 17, 18, 44

Blackboard is a component implemented in a Behavior Tree to share data among the nodes.. 10, 24–27
Bullet Library is state-of-the-art library for physics simulation for games, visual effects, robotics and reinforcement learning. 15

Deliberative Layer The Deliberative Layer reasons about the high-level goals of the system, using symbolic task planner, constraint solver, analysis tools, etc.. 3, 15
Differential wheeled robot is a mobile robot whose movement is based on two fixed wheels placed on either side of the robot, it supposed that each wheel is attached of its own motor. 30
Dynamic Behavior Tree is a Behavior Tree created in runtime. 28

Finite State Machine is a control structure that specifies what a robot should be doing at a given Time. It maps a set of robot states to a set of events that may happen through a transition function. 5

Odometry is the use of data from motion sensors to estimate the change in position over time. 31

Pose is the state, (x, y, θ), of a rigid-body in $\mathbb{R}^3$. 21, 22, 26, 31
Position is the state, (x, y), of a rigid-body in a $\mathbb{R}^2$. 31
Potential Fields Methodology control architecture that use vectors to represent the action output of behaviors and vector summation to combine vectors from different behaviors to produce an emergent behavior. 5, 7
PyTrees Behavior trees implemented in this work used the PyTree framework. 15, 19, 35, 36

Reactive Layer The Reactive Layer reacts to environment events without to much reasoning behind, it has to be simple and quick.. 3

Sequencing Layer The Sequencing Layer is responsible for situation depended task execution, coordinating and configuring all other software components in the system. 15, 18
Skill Layer The Skill Layer realizes the functionalities required to fulfill the task and goals.. 15, 23
State Machine same as Finite State Machine. 3, 5, 17

Subsumption Architecture is a control structure where a number of controllers are executed in parallel, and higher priority controllers subsume (or suppress), the lower priority ones, whenever needed. 5, 7

Tick The signal that excites the root node and it is propagated through the Behavior Tree. 19, 24
Tree Traversal is the process of visiting each node in a tree data structure. 19

Waypoints Waypoint is a point in a path or line that a robot should pass. x, 21–24, 30, 38, 42

1 INTRODUCTION

Robotics has achieved a lot of success in industrial manufacturing. Robotic arms, or *manipulators* (Figure 1.1), have been used in assembly lines for their speed, accuracy, and precision. Among other uses, it has made possible making cellphones, computers, cars and myriads of other industrial products in large scale. Even though this market should be worth US$ 75.3 billion by 2026,[1] there is a growing market towards robots even more flexible – US$ 102.5 billion by 2025.[2] These robots should be designed for particular applications usually driven by economic/social reasons and to fully utilize the potential of a working robot requires mobility and the ability to cooperate with other robotics [2].

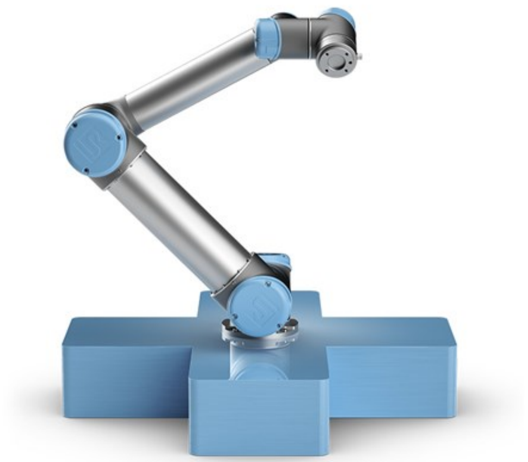


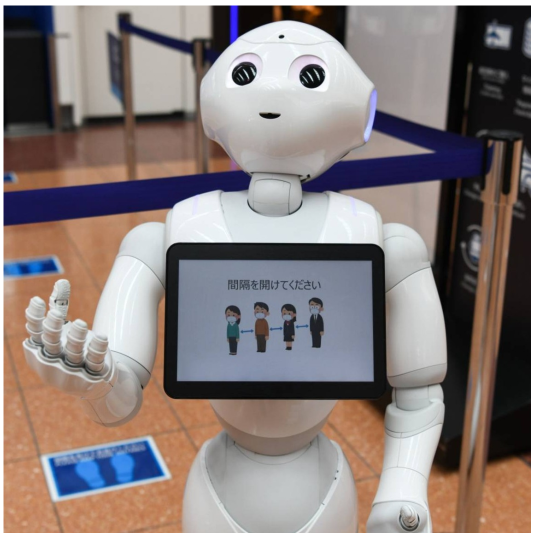
Figure 1.1 – UR3 Universal Robots (Source: `https://www.universal-robots.com/products/ur3-robot/`)

Service robots are robots designed to act among humans in environments such as hospitals, logistics, and transportation in an autonomous manner [3]. Some examples are depicted in Figure 1.2. The ISO Standard 8373 [3] from 2012 defines that such robots should "*perform useful tasks for humans or equipment excluding industrial automation applications*" and has the "*ability to perform intended tasks based on current state and sensing, without human intervention*". The report "A Roadmap for US Robotics" [4] shows the COVID-19 pandemic sped up the deployment of robots in factories and opened new demands in the health sector. Sectors as e-commerce and manufacturing sectors cast new demands towards cleaning and disinfection without a human being.

[1] `https://www.marketsandmarkets.com/PressReleases/industrial-robotics.asp`
[2] `https://www.marketsandmarkets.com/Market-Reports/service-robotics-market-681.html`

Robots as a cyber-physical system are capable of acting, sensing the environment, and executing complex tasks. Moreover, a Service robot has to do everything a usual robot does, still working alongside other agents like humans and other types of robots. There is no definition in the robotics community about the standardization of communication between autonomous agents in populated environments. Neither is there a system that facilitates communication between humans and robots. The Robotics 2020 Multi-Annual Roadmap For Robotics in Europe (MAR) [5] defines the developmental drivers for R&D in European robotics projects. The drivers go from configurability, adaptability, and dependability and define each driver's development levels to be taken. In order to meet these drivers and deploy robots in real-world applications, we need a flexible robot control architecture to coordinate the robot's behavior.

(a) Locus Robots


(b) Pepper from Softbank Robotics

(c) Model of Cleaning robot

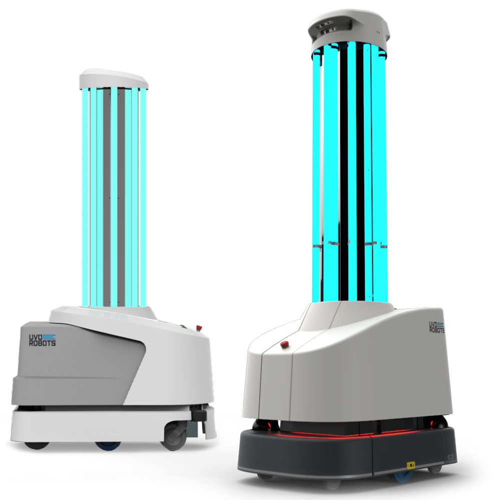

(d) UVD Robots

Figure 1.2 – Examples of Service Robots

A *robot control architecture* guides and constrains the organization of a robot's "*brain*".

The *hybrid control architectures* blend actions in a long-timescale, Deliberative Layer, and actions in a short-time scale, Reactive Layer [6]. The 3-Tiered Architecture is a hybrid control architecture where three layers, Skills, Sequencer, and Planner - are organized by the scope of the robot state. The Skill layer works on the present state, handling what the robot has to do now. The Sequencer layer links the past with present states remembering what the robot has done and what is successful or not, while the Planner layer works on the future using information from the past. The Sequencer instantiates the Skills, a collection of behaviors to coordinate robot actions [1]. For Service robots, this means that the skills manage robotics components like navigation system, perception, motion planning, communication, and end effectors.

The Planner layer searches for sequence of actions or tasks to meet a goal. On the other hand, the robot must be able to manage its skills to follow the plan in runtime. In the past years, roboticists solved this issue using State Machines. However, their usage declined due to the difficulty in programming complex and hierarchical behaviors. Roboticists are switching their solutions to use Behavior Trees (BTs) due to code modularity, reusability, and composing behaviors. Nevertheless, there is a gap in using BTs with standard and already researched planners, like in hybrid control architectures. So, we devise to use BTs to implement the Skills that encapsulate a given task. Then we can switch between different BTs at runtime following a plan using a Sequencer process to handle the switch. We call these *Dynamic* Behavior Trees (DBTs). Our goal here is to investigate the usage of our proposed DBTs, bridging this gap.

1.1 PROBLEM DESCRIPTION & PROJECT GOALS

BTs are gaining popularity in robotics because they are highly flexible, reusable, and well suited to define reactive and deliberative elements in robotics architectures [7, 8, 9, 10, 11, 12]. Even though their appeal, it is still not clear how to design a BT that follows the output from a classical task planner such as a Planning Domain Definition Language (PDDL)-based or Answer Set Programming (ASP)-based.

The Software Engineering Lab (LES) in the Computer Science Department at University of Brasilia (UnB) researches using high-level goal-oriented planners [13]. These planners implement task planning like the deliberative and hybrid control architectures seen in the robotics literature. However, there is an open question: “*How to manage robotics skills and execute such plans at runtime?*”.

Therefore, we will investigate the feasibility of using BTs to execute the plans in this work. The main goal of this work is to bridge the gap between classical planners and BT. Specific goals are twofold: (i) implement a framework to use BTs and classical planners, and

(ii) evaluate if DBTs are suitable to implement behaviors at skill level.

1.1.1 Work Overview

The present chapter brought up the behavior coordination problem in robotics and its challenges. The first section explained how service robots emerged in the 21st-century social-economics scenario to fulfill particular applications – services. Consequently, the service robots concept fills a gap between robots being developed at research labs and robots being deployed in commercial uses. As service robots have to deliver a task and still react to events that may happen while running, there is a need for an approach to be flexible enough to handle both sides. Then, the problem description and goal of this work were defined.

Including this Introduction chapter, this work is organized into five chapters. In Chapter 2, the theoretical background and tools are presented. The Chapter 3 shows the project development and how to implement Skills with BTs. Chapter 4 describes the achieved results in a simulation environment. The simulation tools utilized in this work are also shown in this chapter. Last, Chapter 5 contains the conclusion, related works, and future work ideas to enhance and better evaluate the work done.

1.1.2 Achieved Results

This work outcomes are divided into development and simulation results. The development results are what we could achieve and implement as a usable framework. On the other hand, the simulations show if we met our goals. The programming framework implemented to develop DBTs turns into more straightforward future works and applications involving implementing new behaviors and skills for robots in service robots scenarios. Although we do not target multi-robot problems, we suggest a communication system between agents at such a level, either Human-Robot or Robot-Robot Communication, is possible and easy to interface with high-level task planners. The Skill library is a shared library among robots as well. The BT engine written in Python allows a robot to execute a Sequential Plan if a BT implementation is available in the library. Afterward, the simulation enables an environment to execute and test the framework with a local plan. The simulation was built to be swappable by a real-world robot, adding more flexibility and reducing the amount of reprogramming when deploying and testing the Skills or Plans. The programming framework[3] and simulation[4] are publicly available in GitHub as well.

[3]`https://github.com/Gastd/py_trees_ros_behaviors/`
[4]`https://github.com/Gastd/morse_simulation`

2 BACKGROUND AND TOOLS

Autonomously controlling a robot is not a new problem, but it is not old either. The first time humans tried to answer "*How a robot should do this?*" was about the '60s when the first real robots appeared, but the question is still more open than ever. Nowadays, this problem has passed through time from research labs to the market where robots are coming.

Finite State Machine (FSM) is a specification tool primarily used to compose abstract behaviors and coordinate them. An FSM is usually represented by a set of states S, events E, and state transition function δ(s, e). The states represent the current robot situation and specify how the robot should behave, events model the external events that may happen, and the transition function calculates the next state based on the current state of an event. Alongside Potential Field and Subsumption Architecture, the robotics community has used FSM to model what a robot should be doing at a given time [1]. However, apart from Potential Field and Subsumption Architecture, Finite State Machines models behaviors where the interactions may be more nuanced. That is why FSM is still one of the most used tools to develop robotics behaviors nowadays.

Even though FSMs are easy to understand and implement, they have downsides [14]. Some of them are:

- Many states and many transitions are hard to compute and understand. Adding and removing states require re-modeling the State Machine.
- Transitions between states depend on internal variables and specific states, making it impractical to reuse the same sub-FSM in multiple projects.
- Transitions are hard to model for continuous transitions.[1]
- FSM have unpredictable behaviors whenever there is an unmodeled state or an event is happening.

A BT, depicted in Figure 2.1, is a tool developed to represent how an agent should behave. The game community was the one to create and use to control autonomous characters. In 2012 at CMU, Bagnell et al. used a BT to model the comportment of a robot for the first time. In the following years, the robotics community followed their steps pushed by how modular and reusable BT software is.

[1]Take for example a race self-driving car, there is two modeled states, straight-line driving and curve driving. It is difficult to design a transition function between these two states. Because, it is not clear where a straight-line ends and begins a curve, and vice-versa.

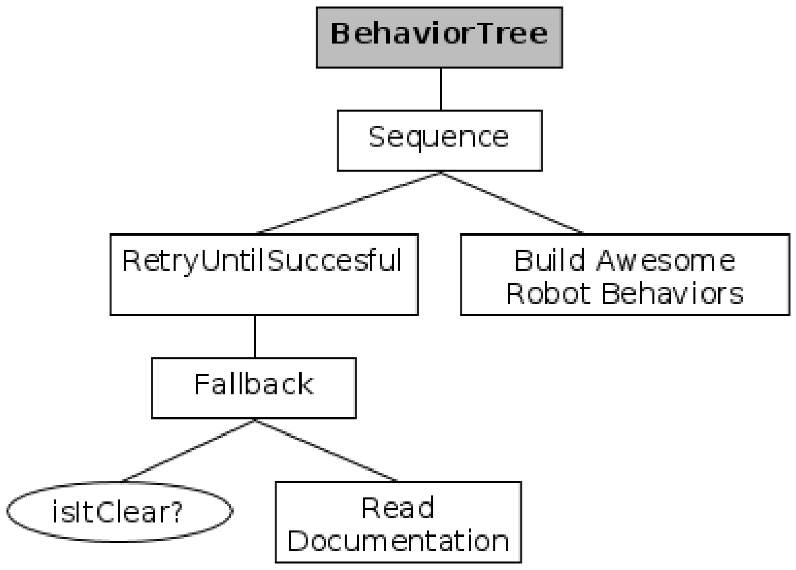


Figure 2.1 – Behavior Tree Example (Source `https://www.behaviortree.dev/`)

Michelle Colledanchise, together with Peter Ögren and several others, published various papers and a book [8] showing how effective BTs are to model robotics behaviors. He is also one of the creators behind one of the current most popular BT implementations, the BehaviorTree.CPP.[2] Besides BehaviorTree.CPP, there are many implementations in several languages for multiple purposes.[3,4] Even the Boston Dynamics' Spot[5] has BT as its main Mission API.

2.1 3-T ARCHITECTURE FOR PLAN, SENSE-ACT

The Plan,Sense-Act (P,S-A) interaction is a paradigm for robotic control architectures that decouples the planning from the sensing-acting loop as the planner functions on long time horizons. Figure 2.2 shows the relationship between the plan and sense-act, in runtime the sense-act loop is usually called as *executor* because it executes the plan. As in the Service robots context, we may have several agents working together. We had to choose an architecture that follows P,S-A. Decoupling the Planner allows the system to plan complex goals with multiple agents, and each agent executes its local plan. Thus, We chose 3-Tiered Architecture.

The 3-Tiered Architecture was developed at NASA as a merging solution between three previous architectures [16, 17, 18]. The name 3-Tiered Architecture is because the robotic

[2] `https://github.com/BehaviorTree/BehaviorTree.CPP`
[3] LUA Example: `https://github.com/tanema/behaviourtree.lua`
[4] Python Example: `https://github.com/splintered-reality/py_trees`
[5] `https://www.bostondynamics.com/spot2_0`

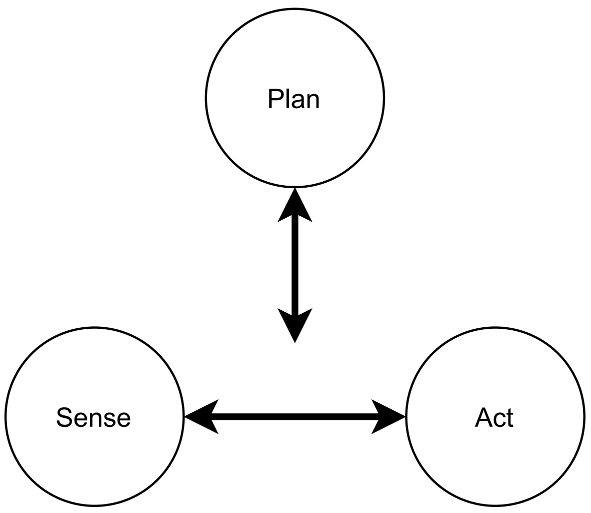


Figure 2.2 – Plan,Sense-Act style of interaction (Adapted from [1])

software system was divided into three layers, one reactive, one deliberative, and the last one is an interface between the previous two. At the Jet Propulsion Laboratory , this architecture was used for planetary rovers, underwater vehicles, and robot assistants for astronauts [1, Chapter 12]. Nevertheless, as defined by Murphy, 3-Tiered Architecture is an example, of a state-hierarchy style organizing its activities around the scope of time knowledge. Figure 2.3 shows the architecture overview for 3-Tiered Architecture, the red box highlights where the work proposed in this document fits in the given architecture.

A short history behind the P,S-A architectures goes back to the early days of artificial intelligence and robotics. In the '60s, the hierarchical paradigm was the first approach where robots must follow the sequence `SENSE-PLAN-ACT`. The hierarchical paradigm provides order and relationship regarding the steps program steps `SENSE-PLAN-ACT`. However, mainly because of the planning algorithms, there was a bottleneck performance-wise. Even though we have computers that can handle expensive planning algorithms nowadays, this paradigm has fallen from popularity due to the paradigm holding back modularity and portability – hierarchical programs tend to be monolithic rather than object-oriented.

After the rise and fall of the hierarchical paradigm, the reactive approach takes inspiration in biology and, as several animals, connects the sensing to acting as an end-to-end solution. It was a rather popular approach, as there is no need for long-term planning the programs were fast with quick responses, running in the old computers. The most famous example of reactive systems is the potential field methodology, where a function of all behaviors is a vector, and the result is the vector sum of the behaviors. Two examples of the reactive approach are: (i) the Subsumption Architecture and (ii) Potential Field. Both architectures are modular and reusable, but the reactive paradigm's main disadvantage was the lack of high-level abstraction. High-level abstractions and reasoning allow robots to work in different problems even when the devised problem is different from the real one because

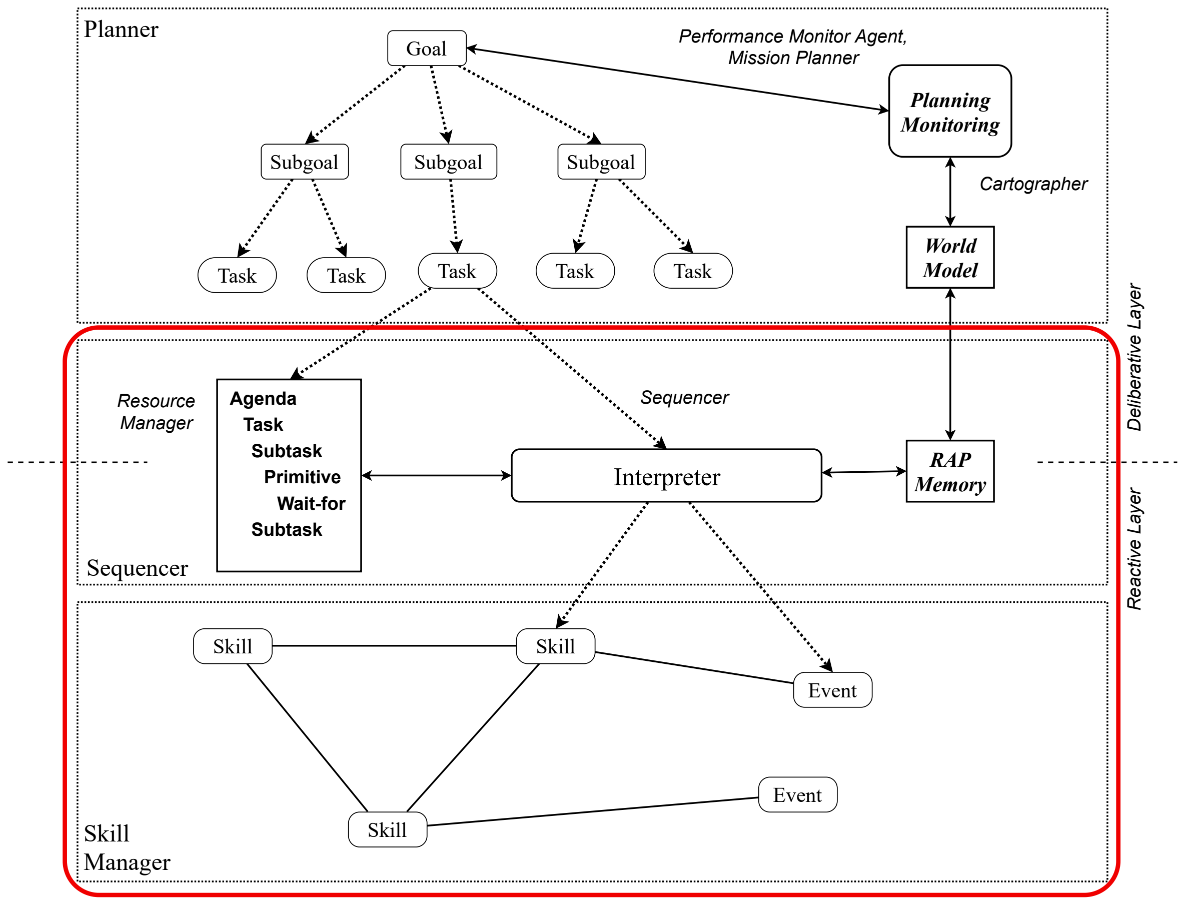


Figure 2.3 – 3-T Architecture overview (Source: [1])

the representations are still there at a high level. Reactive programs are too attached to their problems, even though the programs are reusable. If the robot is running in a different problem, e.g., a different environment, from the designed problem, there is a possibility that a new undesired behavior emerges.

The hybrid approach tries to take advantage of both hierarchical and reactive, in a shape where the disadvantages could be erased. The advantage is that having the planning algorithm running detached from the sense-act allows a quick and inexpensive sense-act loop required in dynamic environments while planning systems has time to plan the tasks and actions.

This chapter aims to define the concepts shown throughout the document and show the features of the tools used to build this work. We begin settling the concepts of what a plan and task planner and skills are. We also compile what the literature says about BTs and how to implement them. Then, we talk about Task Planning, Skills, and Behavior Trees. We finish listing the tools used like Robot Operating System (ROS) and Morse.

2.2 SKILLS

Skills are constructs to create generic templates of assemblages of behaviors [19, 17]. They work as an actor's script, where the robot acts like it is reading the script. Table 2.1 shows how an actor's script compares to a robot script. The primary sequence of events is called a *causal chain*. The causal chain is critical because it embodies the coordination control program logic just as an Finite State Automata (FSA) [1]. A sub-script could be used if an unexpected event occurs to handle the exception.

For example, consider a robot that is delivering medicine to some patient. The first action on the causal chain is to get the medicine in someplace in the house. The second is to pick up the right medicine. Then, move to the delivery room. Lastly, deliver the medicine to the patient.

Table 2.1 – Comparison of actor's script structures to skills(Adapted from [1])

Actor's Script	*Skills Elements*	*Example*
Goal	Task	deliver a medicine $\texttt{M}_1$ to the patient $\texttt{P}_1$
Places	Applicability	a house (domestic robot)
Actors	Behaviors	`GET_OBJECT`, `MOVE_TO_GOAL`, `DELIVER_OBJECT`, `IDENTIFY`
Props, Cues	Percepts	red, blue
Causal Chain	Sequence of Behaviors	`MOVE_TO_GOAL(KITCHEN)`, `GET_OBJECT`($\texttt{M}_1$), `MOVE_TO_GOAL`($\texttt{P}_1$`_ROOM`), `IDENTIFY`($\texttt{P}_1$), `DELIVER_OBJECT`($\texttt{M}_1$)
Sub-scripts	Exception Handling	if have $\texttt{M}_1$ and drop, try `GET_OBJECT` three times

Ghallab et al. define the difference between a *skill* and a *plan* as Skills are an organization of steps in a more complex structure than a plan. A skill such as a navigation skill has to involve sensing and actuating functions, loops, and conditionals steps. Skills are also a recipe, a collection of actions. However, they must be more structured as the robot instantiates them to coordinate its actions and components.

A Skill coordinates the functions distributed in the robotics modules (components) according to the task requirements in this work [21]. An agent chooses and retrieves appropriate skills from a library developed offline; it instantiates and adapts them to the current context [20].

2.3 BEHAVIOR TREES

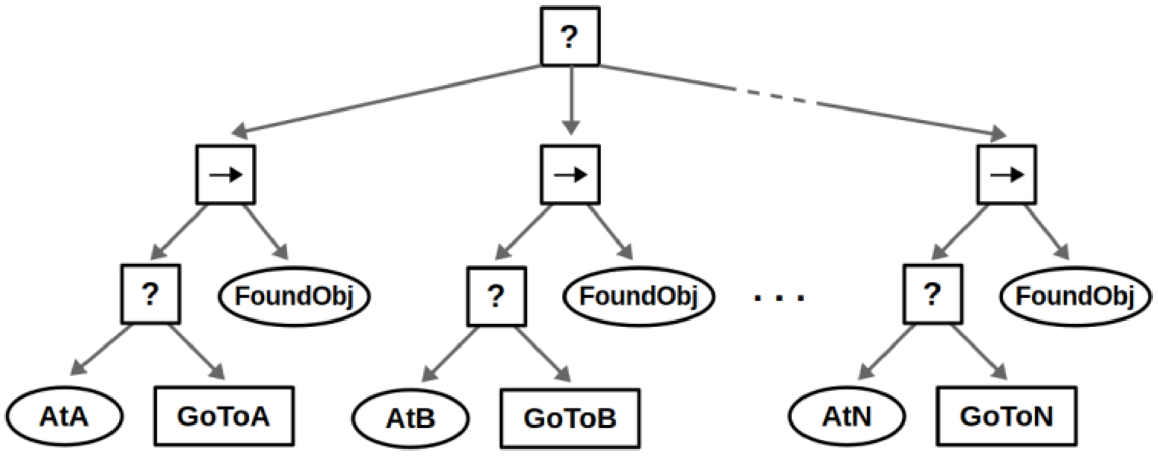


Figure 2.4 – *Behavior Tree* example (Source: `https://roboticseabass.com/2021/05/08/introduction-to-behavior-trees/`)

As shown in Figure 2.4, the Behavior Tree is a mathematical model for representing how switching actions are related. A BT has a tree-based structure, a set of nodes are connected, one node can have only one parent but any quantity of children. A more formal definition for BTs is directed acyclic graphs where one node is assigned as the root.

The games community created BTs to model the behavior and control the actions of Non-Player Charactersss (NPCs). Before BTs, NPCs were developed as FSM. However, they were replaced with a far more modular and reusable approach. One of the first uses of BT in a robotics system was in [15] being used to control a manipulator [22]. The citation below by Bagnell et al. display the leading reason behind the adoption of BTs in robotics domain.

> "The main advantage is that individual behaviors can easily be reused in the context of another higher-level behavior without needing to specify how they relate to subsequent behaviors." [15]

A *Behavior Tree* has simple components structured in tree shape, where the tree leaves are execution nodes, and the intermediate nodes control the tree flow. BTs have a component called blackboard used to share data among the nodes. Figure 2.5 shows the four categories of control nodes are: (i) *Sequence*, (ii) *Fallback*, (iii) *Parallel*, and (iv) *Decorator*. The leaf nodes or execution nodes have two types, Action or Condition, and each node can have three states, they are: (i) *Success*, (ii) *Failure*, (iii) *Running*.

Algorithms 1, 2, and 3 show the pseudocode for the Sequence, Fallback, and Parallel nodes. The Sequence ticks its children from the left until finds a child that returns either

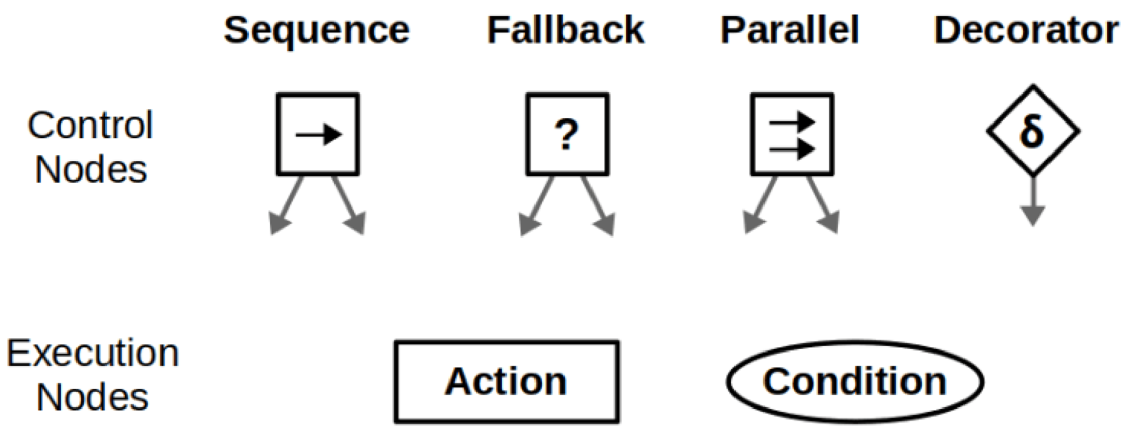


Figure 2.5 – Node type seen in Behavior Tree (Source: `https://roboticseabass.com/2021/05/08/introduction-to-behavior-trees/`)

Failure or *Running*. It returns *Success* if all children returns *Success*. The Fallback node route the ticks to its children from the left until it finds a child that returns either *Success* or *Running*. It returns *Failure* if all its children returns also *Failure*. The Parallel returns *Success* if M children returns *Success*, it returns *Failure* if $N - M + 1$ children return *Failure*. Lastly, an Action node executes a command. It returns *Success* if the command is correctly completed or *Failure* otherwise. When, it receives a tick, a Condition node checks a condition. It returns *Success* or *Failure* depending on if the condition is *True* or not.

Algorithm 1 Pseudocode of a Sequence node with N children

```
1: for $i \leftarrow 1 \in N$ do
2:     $childStatus \leftarrow$ TICK(child(i))
3:     if $childStatus = Running$ then
4:         return $Running$
5:     else if $childStatus = Failure$ then
6:         return $Failure$
7:     end if
8: end for
9: return $Success$
```

Algorithm 2 Pseudocode of a Fallback node with N children

```
1: for $i \leftarrow 1 \in N$ do
2:     $childStatus \leftarrow$ TICK(child(i))
3:     if $childStatus = Running$ then
4:         return $Running$
5:     else if $childStatus = Success$ then
6:         return $Success$
7:     end if
8:     return $Failure$
9: end for
```

Algorithm 3 Pseudocode of a Parallel node with N children and success threshold M

```
1: for $i \leftarrow 1 \in N$ do
2:     $childStatus \leftarrow$ TICK(child(i))
3: end for
4: if $\sum_{i:childStatus(i)=Success} 1 \geq M$ then
5:     return $Success$
6: else if $\sum_{i:childStatus(i)=Failure} 1 \geq N - M$ then
7:     return $Failure$
8: end if
9: return $Running$
```

Behavior Tree (BT) have been showing much applicability in different fields. BTs are being used for robotic manipulation [15, 23, 24] and mobile manipulation [25, 26, 27]. While BTs have also been used to allow train non-specialists to program pick and place tasks [28]. In academia, studies show that BTs can generalize earlier ideas like subsumption architecture [10] and follow deliberative policies [12]. [8, 9] show some design principles to achieve readability, reactivity, safety, and deliberation.

However, BTs are not perfect. Some disadvantages hold back the usage in the service robots domain. In order to be executable by currently available engines, the BT has to be designed beforehand. Thus, problems arise regarding visualization and maintainability when the robot works as a subsystem of a much larger one. Service robots work on valuable tasks for humans and can perform the tasks without human intervention [3]. In this field, the robot requests a human or another autonomous system to perform several tasks in an environment populated by humans. A BT that encodes every possible task and handles faults is significantly large and is difficult to maintain. A possible solution is to have a smaller BT that encodes just one task and switch at runtime between them.

2.4 ROS

2.4.1 ROS1

ROS is a *framework* to develop robotics software. It is a collection of libraries, tools, and conventions that simplify and standardize software production. The robotics community considers ROS as a historical event that shapes the robotics software production afterward. With ROS, the robotics software can be hardware agnostics, e.g., the developer can write their code without previous knowledge of the hardware, and the same software could be used in different platforms without more extensive changes. Furthermore, ROS has a large, active, and collaborative community developing software and integrating frameworks from different research groups across the world. In 2021, ROS is 14 years old, and its ecosystem's main characteristics are code integration using the conventions standardization and its message transport library.

ROS have these features:

- message transport
- message recording and playback
- remote call functions
- distributed parameters system

2.4.1.1 Navigation

The robot navigation used in this work is the *2D navigation stack* from ROS. The ROS navigation stack is a set of libraries, algorithms, and executables for mobile robots, and they implement the architecture shown in Figure 2.6. ROS has its own navigation system, which uses odometry and Light Detection and Ranging (LIDAR) to build a world (environment) representation and plan the sequence of movements in a two-step way to plan and execute the desired path. The main program that manages the stack is called *move_base*, Fig. 2.7.

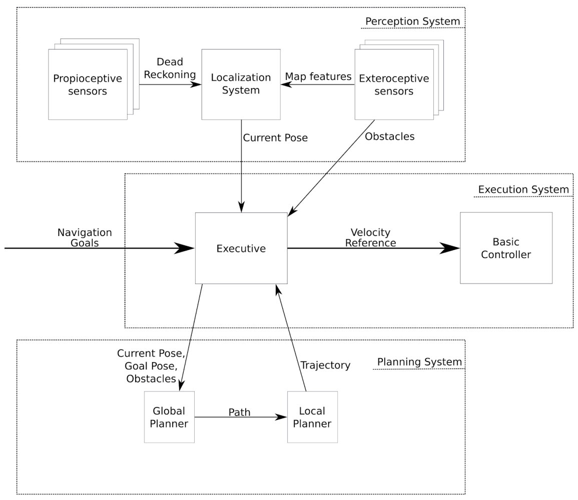


Figure 2.6 – ROS software architecture used in mobile robots

2.4.2 ROS2

Robot Operating System 2 (ROS2) is the successor of the famous ROS presented in section 2.4. As ROS, ROS2 is also a set of libraries and tools used to develop and reuse

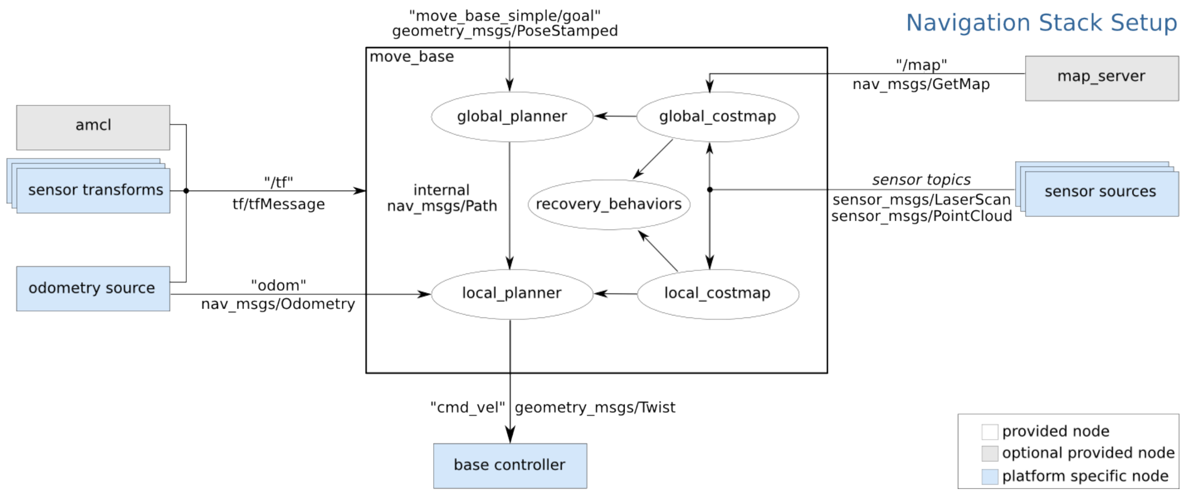


Figure 2.7 – move_base overview (Source: `http://wiki.ros.org/move_base`)

robotics software quickly. After fifteen years of worldwide use, ROS is used in several domains that it was not even designed to fulfill. The primary purpose of the original ROS was to be used as a development environment by *Willow Garage*'s PR2 robot. Even though ROS was designed to push the reuse of robotics research software, it was too much coupled to PR2 design drivers. PR2 had two computers connected by a wired network. There were no real-time requirements neither commercial uses. Funny enough, nowadays, ROS is used in a wide variety of robots far different from the old than the old PR2, e.g., legged humanoids, industrial arms, and self-driving cars.

So, ROS2 is designed to tackle problems that the old ROS was not meant to solve, and ROS2 is also made to push forward the progress and robotics future. This new environment is designed to:

- Teams of multiple robots
- Small embedded platforms
- Real-time systems
- Non-ideal networks
- Production environments
- Prescribed patterns for building and structuring systems

ROS2 improvements over ROS it is not meant to kill ROS but to work alongside. Many robots that run the old ROS are too risky to get rid of such a widespread environment. Further reading in post *Why ROS2?*[6] by Brian Gerkey.

[6] `http://design.ros2.org/articles/why_ros2.html`

In this work, we used both ROS and ROS2. There are two main reasons for ROS2 was chosen to be in this project. First, we chose to use ROS2 because the features needed for this project were much more developed in pytrees, 2.5, in its ROS2 version. The second reason, the Sequencing Layer has to interface with the Skill Layer with the Deliberative Layer. In our project, the Deliberative Layer is not inside the robot.

2.5 PYTREES

PyTrees [29] is a python BT library used to develop decision-making systems for the robotics community quickly. Their main sell point is using python language features like generators and decorators to help easy-to-use and quickly develop engines and new behaviors. The library design goals are:

- Quick development
- Medium-scale software
- Not real-time reactive

As python is an interpreted language, it does not add any problems in instantiating and destroying objects. So, it is a match if we want to change the BT in runtime with different parameters. The library has a ROS implementation called *py_trees_ros* where extends the library to use in the ROS environment. The *py_trees_ros* implements some Action Nodes as ROS nodes to transfer back and forth data from ROS objects to *pytrees* objects. However, the pytrees stable-release is in ROS2 with many features missing in its ROS1 version. In this work, we use pytrees and py_trees_ros, both in version 2.1 for ROS2.

2.6 MORSE

Simulators are programs built for simulating some environment in order to test systems before the actual deployment. Simulators are used in the robotics community because of how quickly novel algorithms can be tested without the overhead of configuring and testing in real life. The *Morse Simulator*[7] [30] is a robotics simulator. Its main focus is simulating a dozen robots from small to large environments, either indoor or outdoor. It uses the Blender Game Engine (BLE) to render and Bullet Library for physics simulation. *Morse* is written in python for easy and fast modification and extension. Along with *Morse*, there is many simulators options each one with its own characteristics like Gazebo, CoppeliaSim, and Webots.

[7]`https://www.openrobots.org/morse/doc/stable/morse.html`

Morse supports several middlewares, including ROS. However, its interface is only available for ROS1, which had a great deal with BTs working on ROS2. In 2020 April, Morse was stated as a dormant project by its maintainers[8] as BLE was removed from Blender in the latest release.

[8] https://sympa.laas.fr/sympa/arc/morse-users/2020-04/msg00001.html

3 IMPLEMENTING SKILLS WITH BEHAVIOR TREES

The 3-Tiered Architecture glues a planner running on the deliberative layer to a skill manager that runs on top of the robotic components, the reactive layer. The sequencer selects a skill to execute the tasks while monitoring its execution. 3-Tiered Architecture (3T) was created in the ’90s but still inspires robot control architectures nowadays. For example, the ROS Navigation stack [31] uses a State Machine to sequence the navigation while making queries to a path planner and keeping the sense-act loop alive. As the Navigation stack, many robots use FSM to sequence their actions. However, in a service robot domain, we have to rule the robot’s components, subsystems, and communications. This increased complexity reaches the limit that we can go with State Machines [14]. Behavior Tree (BT) shows great modularity, reusability, and flexibility to express reactive behaviors [15, 32, 10, 8, 33, 34, 35]. Hence, developers worldwide are using Behavior Tree (BT) to coordinate complex behavior in robotics [22, 36].

A BT coordinates a robot’s actions and components to achieve a task. Even though a BT can be designed to show aspects such as deliberation and reactiveness [10], the BT still needs to be static to be executed in the available BT engines. If an unknown plan arrives, a static BT has to have structures and designs to workaround and follow the plan. However, a DBT is a more straightforward approach, where smaller BTs can be configured and constructed at runtime to handle the plan. We can also use previous states or the plan knowledge to build these structures. So, following this idea, we describe how to implement such Skills using what we call *Dynamic* Behavior Tree (DBT).

A DBT is an extension to BTs where the entire BT is not allocated in memory, and just the current behavior (subtree) is running. The current behavior is a subtree that embodies a goal or specific skill that the robot should achieve. A DBT is mounted and expanded in runtime following a Skill Implementation previously implemented and available in a library where the robot retrieves. We proposed the use of DBTs as a middle point/alternative to static specified BTs.

3.1 PLANNING LAYER

Planning is the searching for the outcomes of possible actions and how a sequence of these actions changes the world [6] if they are going to reach a desired goal or state or not. Task Planning in the robotics domain outputs a sequence of actions for a robot to accomplish a goal impossible to reach with only one action [37, 38].

To execute a given task, the Planner gives the robot a recipe. These are then encoded as a *Sequential Plan* and sent to the robot. The robot receives the *Sequential Plan* as input and this should remain static while the robot is executing it. Then, the Sequencing process consumes the plan controlling the robot that must follow the plan. The plan represents a recipe or script, what task, parameters, and sequence the robot should perform to achieve its goal. Each robot receives a local plan that they have to follow. Our plan is a sequence, $\pi = \langle a, b, ..., u\rangle$, of n-tuples, $a' = (a, b, ..., u)$, where the first tuple element is a skill label, and the following tuple elements are the parameters of the skill, a whole tuple encodes a robot skill that will be mapped to a BT at runtime. Each skill has its own set of parameters that will be presented in the next section.

EXAMPLE 3.1 (Plan Example)

$$\pi = \langle (NavTo, Room4), (ApproachRobot, Manipulator2), (NavTo, Room15)\rangle$$

Example 3.1 shows a simple plan where the robot must move to a room called *Room4*, approach the robot *Manipulator2*, and finally move to another room, *Room15*. Parameters like *Room4* and *Manipulator2* are in the world knowledge, so the planner only has to deliver the plan to each robot with the proper labels. In this work, we do not have a planner up and running alongside our approach as we aim to build an executor, the sense-act loop shown in section 2.1. The plan step – in this work – is offline.

3.2 SEQUENCING PROCESS

As defined by the 3-Tiered Architecture in section 2.1, the *Sequencer*, *Sequencing process*, or *Sequencing Layer* is responsible for coordinating the robot behaviors, what behaviors will execute, and interface the deliberative layer with the reactive layer – monitoring the execution and warning the planning layer if something goes wrong. The Sequencer selects and instantiates skills to execute the tasks in the plan. In this work, we implemented the Sequencer as a *dynamic* BT Engine that switches the executed behavior following a received plan, as Figure 3.1 shows.

The Sequencer is developed in three parts:

1. An interface that receives a local plan from the mission coordinator and reports the plan status
2. A BT engine with the role of choosing the proper behavior and executing it

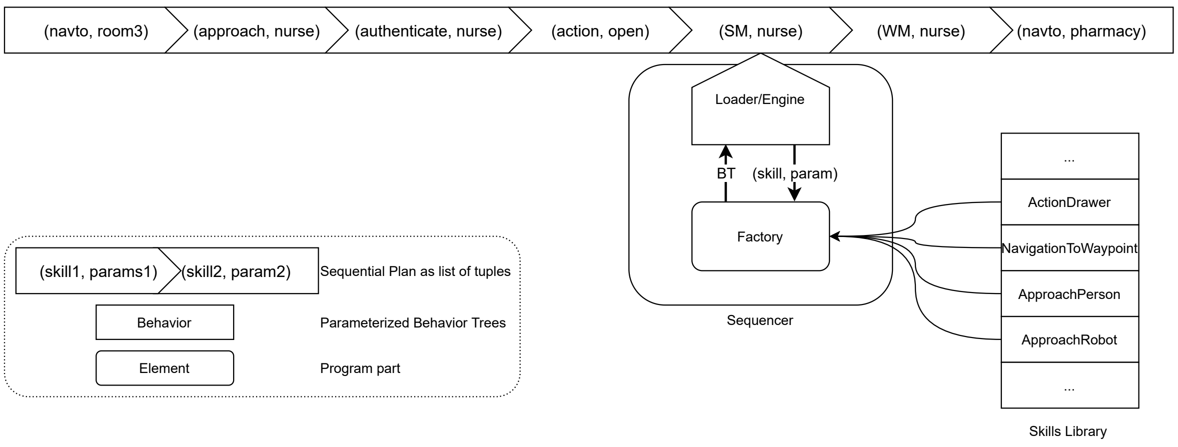


Figure 3.1 – Sequencer architecture

3. A *Factory* that maps each skill label to a Skill Implementation and builds the Active Skill to get ready for execution

A BT engine aims to load a BT and tick the BT root node. Then the signal goes through the tree, exciting the nodes. A BT engine is a tricky program to implement because the tick generation and the tree traversal have to be executed simultaneously as the action execution [8]. We used the pytrees library [29], which implements the BT loading, tick generation, and traversing. The developr's job is reduced to

1. map each skill label to a behavior,
2. build the BT,
3. dynamically switch at runtime between different BTs,
4. report the BT status to the mission coordinator (upper layer).

The *Sequencer* receives the local plan as an ordered list of skills. The Engine gets the right skill in the library and uses the skill parameters to build the BT. Then, the BT is loaded and ticked. After every tick, the Sequencer sends the root node status as a report to the planning layer.

Algorithm 4 shows how the Sequencer works. The Sequencer begins checking for the plan. When there is an available plan, the execution starts; otherwise, the Sequencer waits. For each skill in the plan, the Sequencer queries a Skill Implementation and activates it. Then, the Active Skill is ticked, the local mission is updated, a report is sent, and the loop restarts.

Algorithm 4 Sequencing Process

```
Require:
 1: $local\_mission$: manages local plan, provides the tasks into the correct order
 2: $active\_skill\_ctrl$: manages the life-cycle of the skills
 3: $task\_status$: interface with the knowledge base
 4:
 5: function SEQUENCING($local\_mission, active\_skill\_ctrl, task\_status$)
 6:     if $local\_mission$.HAS_NO_PLAN( )then
 7:         return
 8:     end if
 9:     if $active\_skill\_ctrl$.IS_IDLE( )then
10:         $next\_task \leftarrow local\_mission$.NEXT_TASK()
11:         $skill\_impl \leftarrow skill\_library$.QUERY($next\_task$)
12:         $active\_skill\_ctrl$.LOAD($skill\_impl$)
13:         $task\_status$.SET_VALUE($status$)
14:     end if
15:     $tick\_status \leftarrow active\_skill\_ctrl$.TICK()
16:     $local\_mission$.UPDATE($tick\_status$)
17:     $ts \leftarrow local\_mission$.GET_TASK_STATUS()
18:     $task\_status$.SET_VALUE($ts$)
19: end function
```

3.3 SKILL LAYER

The *Skill Library* developed in this work implements a set of skill implementations that specify how a BT should be instantiated and how the skill parameters that appear in the plan should be substituted.[1] This happens because a BT used in this work is not fully known in design time, and the remnant is parameters chosen by the Planner.

To execute a BT, we need to know the skill parameters before the execution, to specify the *Skill Implementation* in a library. Then, when the robot needs the skill, it should retrieve the Skill Implementation substitute with the appropriate parameters to get an *Active Skill*.

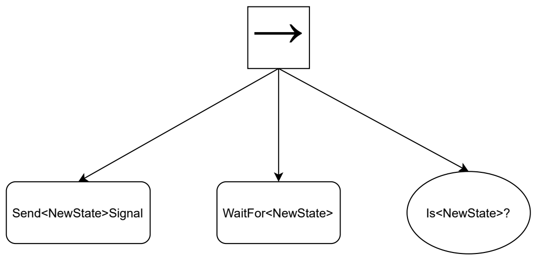
3.3.1 Active Skill vs. Skill Implementation

A *Skill Implementation* is a function – $f : params \mapsto bt$ – that specifies how a BT should be created based on a set of parameters. The parameters are known only at runtime because the planner configures them. Appendix A shows the list of the Skill Implementations for all skills implemented in this work.

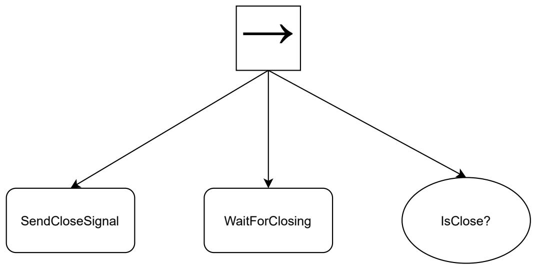
Active Skill is a BT Instance created at runtime using the implementation specification and the parameters. Active Skill is the return of the Skill Implementation function. After

[1]A fully specified BTs do not have any parameters as input, although they can consume a set of parameters at runtime and route the behavior accordingly.

its creation, the BT Engine/Sequencer executes the BT. Further explanation about how this happens is shown further in section 3.2.



(a) Skill Implementation for controlling the drawer state.



(b) Active Skill for the Implementation in 3.2a, where the $NewState = Close$.

Figure 3.2 – *Skill Implementation* and its corresponding *Active Skill.* The Skill Implementation is shown in Algorithm 6.

Figure 3.2 shows the difference between the Skill Implementation and Active Skills when the parameter $NewState = Close$ for the *ActionDrawer* Skill. In order to execute a BT, its parameters must be configured before the execution. That is one of the reasons behind using DBTs to execute a plan.

3.3.2 Expansion and Substitution in *Dynamic* Behavior Tree

As the parameters are only known at runtime, we build the behaviors using the templates and the parameters set by the planner. However, the planner also sends the waypoints to navigation for the Navigation Behavior. So, for this behavior, we implemented a BT expansion.

Algorithm 5 Waypoint Expansion

```
 1: function WAYPOINT_EXPANSION($waypoint\_list$)
 2:     $root \leftarrow$ SEQUENCENODE
 3:     for all $waypoint \in waypoint\_list$ do
 4:         $pose.x \leftarrow waypoint.x$
 5:         $pose.y \leftarrow waypoint.y$
 6:         $pose.\theta \leftarrow$ ATAN2($waypoint.y - old\_way.y$, $waypoint.x - old\_way.x$)
 7:         $GoToPose \leftarrow$ SENDNAVGOALROS($pose$)
 8:         $old\_way \leftarrow waypoint$
 9:         $root$.ADD_CHILDREN($GoToPose$)
10:     end for
11:     return $root$
12: end function
```

Algorithm 5 shows the BT expansion for a list of waypoints received from the planner. For each waypoint, a navigation goal in the format of a pose is calculated, and an Action Node *GoToPose* is constructed using the pose. Then, a sequence behavior is built using the list of *GoToPose* nodes.

Figure 3.3 shows the subtree expanded by Algorithm 5 using the waypoint list from Example 3.2, while Figure 3.4 shows the *NavTo* Skill with the subtree attached. Example 3.2 also show the pose calculated to use as navigation goals in the *GoToPose* nodes.

EXAMPLE 3.2 (Waypoint List)

$$\langle(-21.0, 18.0, -1.57), (-21.0, 16.0), (-18.0, 16.0), (-18.0, 13.0, 1.57)\rangle \rightarrow$$

$$\begin{array}{rcrrcr}
Pose1.x &=& -21.0 & Pose2.x &=& -21.0 \\
Pose1.y &=& 18.0 & Pose2.y &=& 16.0 \\
Pose1.\theta &=& -1.57 & Pose2.\theta &=& -1.57 \\
\\
Pose3.x &=& -18.0 & Pose4.x &=& -18.0 \\
Pose3.y &=& 16.0 & Pose4.y &=& 13.0 \\
Pose3.\theta &=& -1.57 & Pose4.\theta &=& 1.57
\end{array}$$

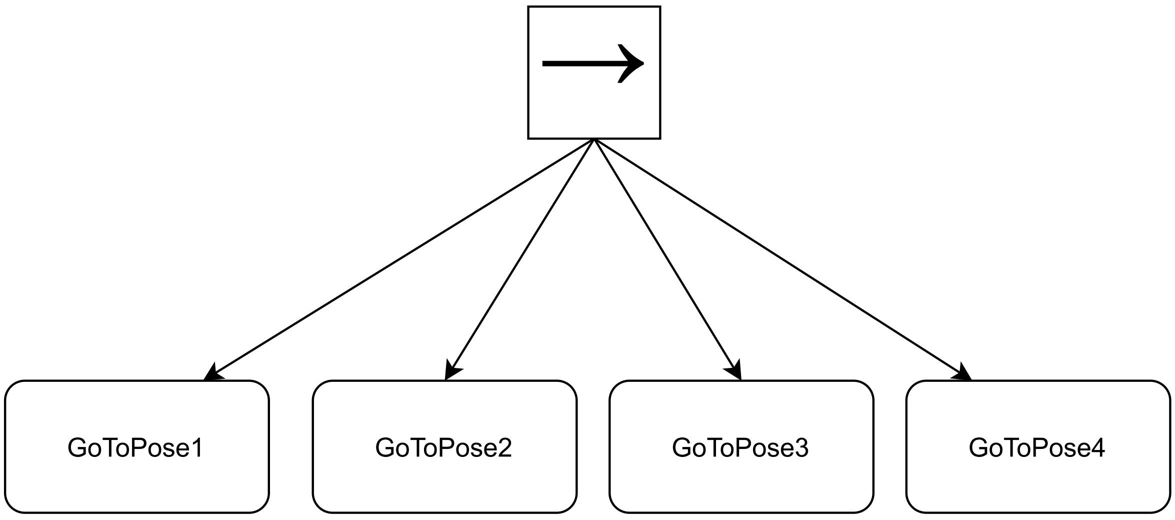


Figure 3.3 – Waypoint list Behavior Tree (BT) expanded for 4 waypoints

3.3.3 Human-Robot Communication and Robot-Robot Communication

To deal with *Human-Robot Communication* and *Robot-Robot Communication*, we implemented Communications Skills to handle. The communications are between two agents working in the same environment. Both communication processes rely on a synchronization process based on client-server model. The first agent sends a message to the second, and then the second agent responds to the first one.

If the second agent is a human, the behavior coordinates the human-robot communication component to handle the communication. This component could be, for example, a touch-

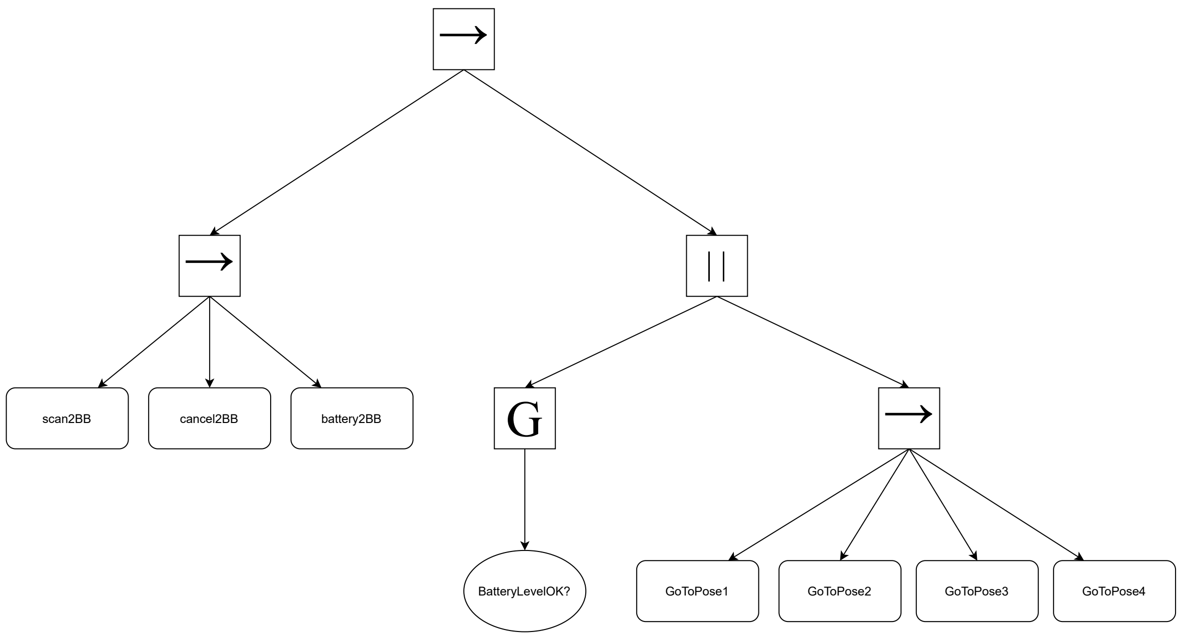


Figure 3.4 – NavTo Skill for 4 waypoints

screen interface, a mobile app, or a voice interface. If the second agent is another robot, the behavior coordinates a communication using ROS topics. Both the communication behaviors work with the same interface for the planner, and they are labeled as *SendMessage* and *WaitMessage* skills.

3.3.4 Skill Implementation for Service Robots

In this subsection, we talk about the Skill Implementation for Service robots domain. In order to address each task specified in the plan, we designed seven Skill Implementations that could be parameterized. Here we also show how the communication tasks [39] are implemented as Synchronization Skill. We split the Skill Layers into groups: (i) Action Skills, (ii) Sync Skills:

3.3.4.1 Action Skills

Action Skills are skills related to the coordination of robotics components like a navigation system, and perception.

Navigation To: Skill used to send navigation goals and report navigation status using waypoints. First, the skill stores the last LIDAR scan reading, the last cancel command, and the last battery state. After that, the parallel watches if the Battery level is above $5\ \%$ and sends sequentially the route waypoints to the navigation component. If any leaf node fails, the whole Behavior is a *failure*. The Behavior is a success when the last waypoint is reached. Figure 3.5 shows the Active Skill with two waypoints as input, and the decorator **G** is the

EternalGuard[2] decorator that forces the BT to run the child every tick.

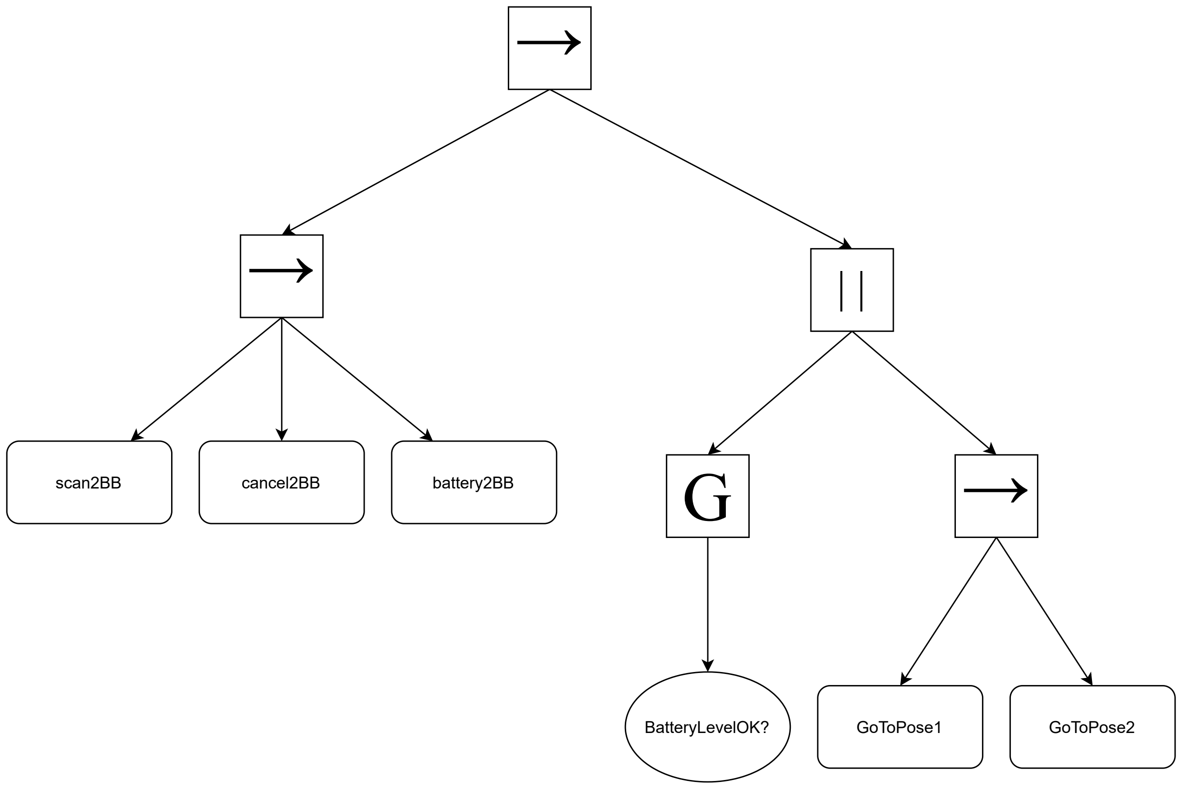


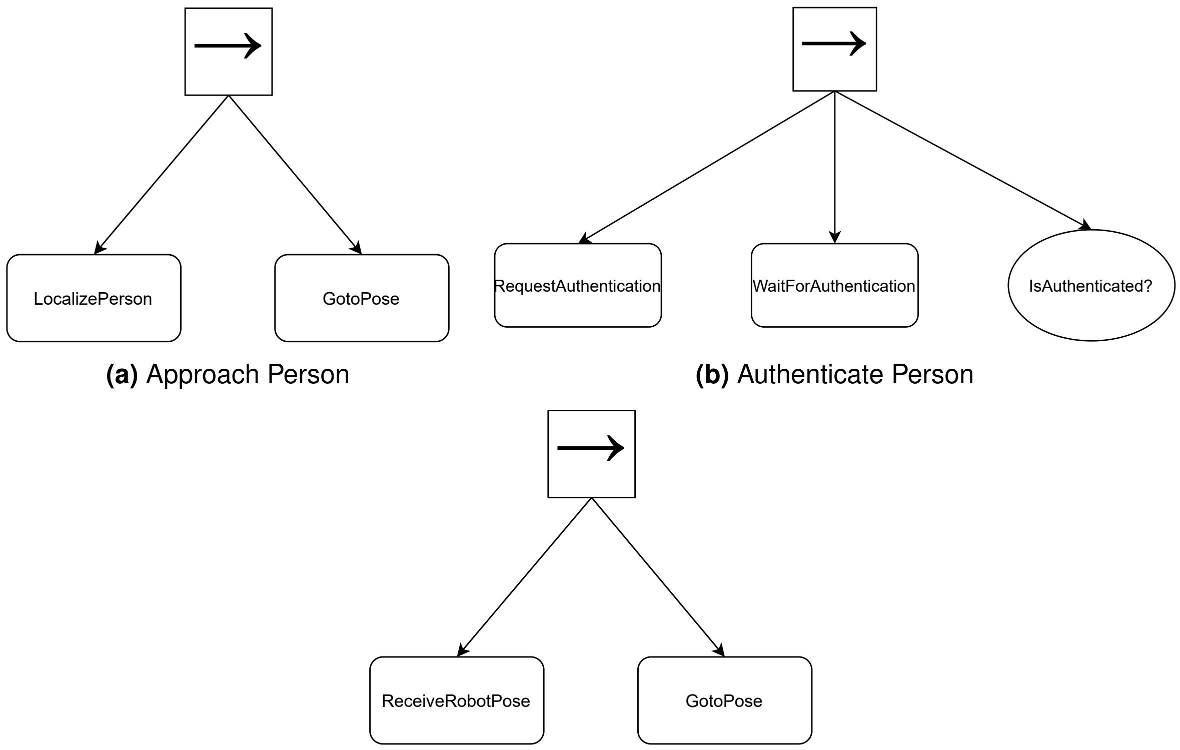
Figure 3.5 – NavTo Skill with 2 waypoints

Approach Person: Skill used to localize a person and get close to them. The Skill requests a Human Detection, when there is a match the person's position is stored in the blackboard and then the Skill sends to the navigation component the a position close to the person, for example $p_{person} = (36.5; 41.2) \mapsto p_{goal} = (35.5; 40.2)$. Figure 3.6a shows this Skill Implementation

Authenticate Person: Skill used to authenticate a Human. The Skill sends a authentication request to the Human-Robot interface and waits for the authentication. The skill reported status is the root node status, *success* if the last condition is *success* or *failure* if any leaf node fails. Figure 3.6b shows this Skill Implementation.

Approach Robot: Skill used to localize a known robot and get close to it. It is similar to the *Approach Person* Skill. The Skill requests to the friend robot a available and close position, when the robot answer, the answer is stored in the blackboard and then the Skill sends to the navigation component the answer, for example $p_{robot} = (20.5; 15.0)$. Figure 3.6c shows this Skill Implementation.

[2]https://py-trees.readthedocs.io/en/devel/idioms.html#eternal-guard



(a) Approach Person

(b) Authenticate Person

(c) Approach Robot

Figure 3.6 – Action Behaviors

3.3.4.2 Synchronization Skills

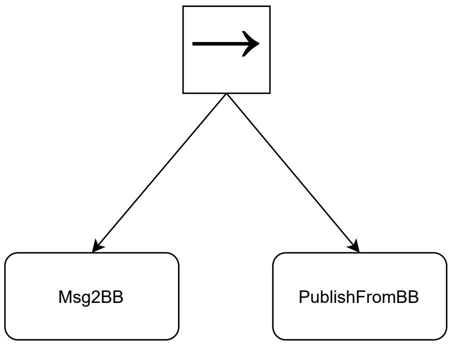
Synchronization Skills are skills shared between the different robots in the environment, even heterogeneous like a mobile robot and a manipulator. They are based on the client-server model, as shown in subsection 3.3.3.

Send Message: Skill used to send a message to another agent. The Skill stores the message in the blackboard, then it publishes the message in a ROS topic given by the planner. The skill reported status is the root node status, *success* if the last condition is *success* or *failure* if any leaf node fails. Figure 3.7a shows this Skill Implementation.

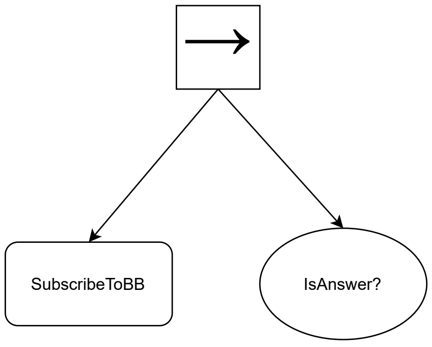
Wait Message: Skill used to wait for a message from another agent. The Skill subscribe a ROS topic, when a message arrives the skill stores the answer in the blackboard and checks if the answer is the same as the planner expects. The skill reported status is the root node status, *success* if the last condition is *success* or *failure* if any leaf node fails. Figure 3.7b shows this Skill Implementation.

3.3.5 Leaf Nodes

As defined in section 2.3, the intermediate nodes from a BT route the control to the Leaf Nodes as they are the ones that check a condition or do an action. In this subsection, we



(a) Send Message Behavior



(b) Wait Message Behavior

Figure 3.7 – Synchronization Behaviors

define and describe the implemented Leaf Nodes for specifying the Skills.

The Leaf Nodes are divided into Action Node and Conditional Node. Action nodes are nodes that perform some task or coordinate some system component, e.g., navigation. Conditional Nodes are nodes that check if some information is *True* or *False*. Both node types have to return a status, *success*, *failure* and *running*. The conditions and comportment of each Leaf Node is described below:

3.3.5.1 Action Nodes

scan2BB: Whenever this condition receives a tick, it stores in the blackboard last LIDAR reading. The condition returns *success* if the last reading is available. It returns *failure* otherwise.

cancel2BB: Whenever this condition receives a tick, it stores in the blackboard last cancel topic reading. The condition returns *success* if there is a last reading is *False*. It returns *failure* otherwise.

battery2BB: Whenever this condition receives a tick, it stores in the blackboard last battery reading. The condition returns *success* if the last reading is available. It returns *failure* otherwise.

GoToPose: Whenever this action receives a tick, it sends a request to a navigation component (in our implementation we use the ROS Navigation Stack, Subsection 2.4.1.1 [31]), and then it waits if the destination is reached. The action returns *failure* if the navigation component cannot move the robot to the Pose. It returns *success* if the robot reaches a destination. It returns running if the robot is navigating towards the destination.

LocalizePerson: Whenever this action receives a tick, it sends a request to the human detection component, which is implemented as mock. [3] The action return *success* if there is a successful detection. It return *failure* otherwise. The position of the detected person is written in the blackboard.

ReceiveRobotPose: Whenever this action receives a tick, it sends a request to the robot-robot component asking for the target robot its own localization. The action return *success* if there is a successful response. It returns *failure* otherwise.

RequestAuthentication: Whenever this action receives a tick, it sends a string message to the Human-interface component to handle a authentication from a user. The action return *success* if the authentication is requested successfully. It returns *failure* otherwise.

WaitForAuthentication: Whenever this action receives a tick, it waits for a response from the human-interface component. The action return *success* if a response is received. The received response is stored in a blackboard variable. It returns *failure* if the timer expires and a response is not received.

SendSignalDrawer: Whenever this action receives a tick, it sends a signal for the *Drawer* component to *Open* or *Close*. The action return *success* if the signal is sent successfully. It returns *failure* otherwise.

WaitForAction: Whenever this action receives a tick, it waits for a message from the *Drawer* component updating its state. The action return *success* if a response is received. The updated state is stored in a blackboard variable. It returns *failure* if the timer expires and a response is not received.

3.3.5.2 Conditional Nodes

BatteryLevelOK?: Whenever this condition receives a tick, it checks if the battery level is greater than $5\ \%$. The condition return *success* if it is *True*. It returns *failure* otherwise.

IsAction?: Whenever this condition receives a tick, it checks if the updated state stored in the is the same as the requested. The action return *success* if the *Drawer* conditional is *True*. It returns *failure* otherwise.

IsAuthenticated?: Whenever this condition receives a tick, it checks if the response stored in the blackboard is a successful authentication. The action return *success* if the authentication is a success. It returns *failure* otherwise.

IsAnswer?: Whenever this condition receives a tick, it checks if the response stored in the blackboard is the same response expected in the plan. The action return *success* if the condition is *True*. It returns *failure* otherwise.

[3]Identify and Localize a person is not an easy problem, actually has its own research domain, and there is no gain in really localizing a simulated person. So, we implemented this component as a direct interface between the simulator and BT, mimicking the behavior of real Detector.

4 EVALUATION AND RESULTS

In this work, we implement and use Dynamic Behavior Trees to command and supervise robots' behaviors and skills to follow a plan. In this section, we show the plan execution using a simulation environment. The simulation environment will act as a plant giving feedback on the skills to work as specified. To evaluate, we show two runs: the first where the robot executes its plan as expected, and the second one where an injected hardware failure simulation halts the plan.

4.1 LAB SAMPLES LOGISTICS MISSION

In the *Lab Samples Logistics mission* used to test the framework, robots are deployed in a hospital environment. This scenario is adapted from the RoboMax repository of exemplars [40]. In that scenario, the robots should transport samples from patient rooms to the laboratory. A nurse is responsible for collecting the sample and can request delivery to the laboratory, identifying the room where the collection should take place. The system must include a robot with a securely locked drawer, which must then navigate to the collection room, identify the nurse, approach her, open the drawer, await the deposit, close the drawer and then navigate to the laboratory carrying the sample. In the laboratory, the sample can be picked up by a robotic arm or laboratory personnel. The robotic arm picks up samples, scans the barcode in each sample, sorts, and loads the samples into the entry module of the analysis machines.

The high-level planner allocates the plan to a target robot, taking into account the knowledge about the state of the system at the moment that the request is handled (i.e., the allocation follows an Instant Allocation, Multi-Robot, Single Task model [41]). When the system receives a task, it can have a varying number of available robots, but – in this work – only one of the robots receives the plan.

Definition 4.1 shows the local plan for the robot in the 'Lab Samples Logistics' mission as a sequence of Skills. The plan is encoded as JSON structure and sent to the robot. The Sequencer layer begins to coordinate the robot as soon as the plan arrives.

Definition 4.1 (Tested Plan)

$$\pi_1 = \begin{array}{l} \langle(NavTo,\ Room3),\ (ApproachPerson,\ Nurse),\ (Authenticate,\ Nurse), \\ (ActionDrawer,\ Open),\ (SendMessage,\ open\_drawer,\ person), \\ (WaitMessage,\ person),\ (ActionDrawer,\ Close), \\ (NavTo,\ Lab),\ (ApproachRobot,\ Manipulator1) \\ (ActionDrawer,\ Open),\ (SendMessage,\ open\_drawer,\ robot), \\ (WaitMessage,\ robot),\ (ActionDrawer,\ Close)\rangle \end{array}$$

4.2 EVALUATION GOALS

We assess our implementation via simulation, aiming at showing if the developed framework was implemented as devised and has everything that needs to execute a deliberative plan. This test is divided into two trials:

- Run the Sequential Plan defined in Definition 4.1.
- Run a similar plan, however the robot spawns at different locations requiring that it should move more and run out the battery.

This setup shows how the framework works in the Simulated Mission, we expect that the framework executes the whole plan without significant errors, and if something goes wrong, the Sequencer has to notify the upper layers to replan or cancel the request, otherwise.

4.3 SIMULATION ENVIRONMENT

Before running the framework along with the simulation, we had to build the environment, configure a simulated robot and implement the interfaces to work with our framework.

4.3.1 Simulated environment

We built the simulated environment to represent a real hospital, so we designed the floor plan based on a real hospital, and then we modeled the building using a 3D CAD software. The CAD was exported as an STL mesh to be used as an environment inside the simulator. Figure 4.1 shows the floor plan used as the source. The red box shows the hospital wing used to run the tests with simulated robots. Figure 4.2 shows the metric map used to help the Navigation Skill and the topological map used by the high-level planner to plan the waypoints to define the Navigation Skill.

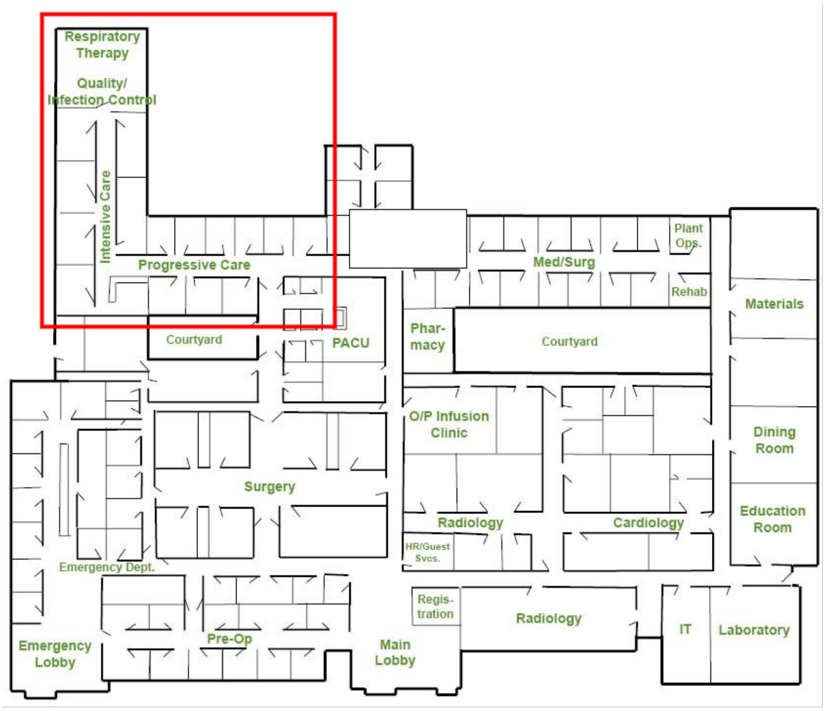


Figure 4.1 – Hospital floor plan used as reference

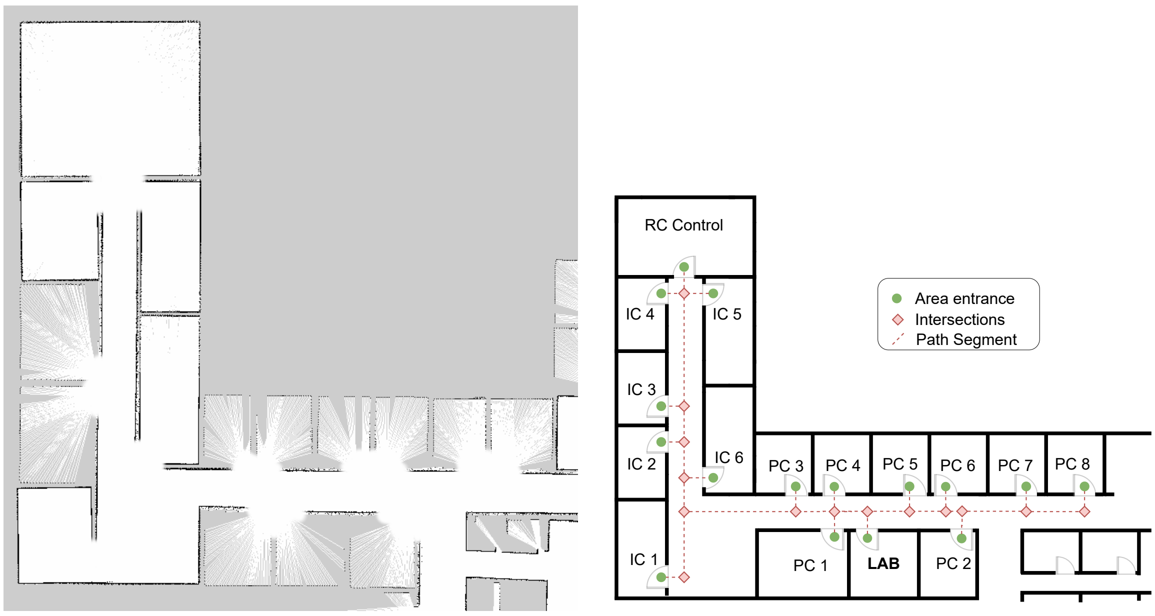
4.3.2 Simulated Robots

The Morse Simulator [30] has a handful of robots already ready to use and a built-in ROS interface, making it suitable for our purposes. Choosing the right platform for our work took into account two guidelines: (i) it should be suitable for the hospital environment, (ii) the simulation has to be realistic enough.[1]

From the mobile bases available in Morse, only two meet our requirements. The *Pioneer 3-DX*[2] and the *Segway RMP 400*.[3] Both platforms use *Differential Driver Actuator* Morse actuator. This actuator calculates the velocity for each wheel following differential-drives Kinematic Model and the friction between the tire wheels and the floor. The thought process behind choosing one is that the ROS Navigation Stack requires a differential-drive or an omnidirectional robot.

We chose the *Pioneer 3-DX* as the main platform for the experiments.

[1]Realistic simulation here means that the robot should follow a kinematic model and it has to take account friction.
[2]`https://www.openrobots.org/morse/doc/latest/user/robots/pioneer3dx.html`
[3]`https://www.openrobots.org/morse/doc/latest/user/robots/segwayrmp400.html`



(a) Partial Slam result of the Hospital wing **(b)** Hospital wing with topological map

Figure 4.2 – Simulated Environment

4.3.2.1 Available Components

The simulated robot has several available components:

LIDAR: The LIDAR component simulates a Hokuyo 30-LX sensor. It is mounted above the robot. The LIDAR publishes the reading in 10 Hz with a range of a maximum of 10 m and a scan window of $360°$. It is published in the ROS topic `<robot_name>/lidar` in the LIDAR frame.

Odometry: The Odometry is used as available in Morse. Simulates the robot's pose using the readings from the movement sensors. The odometry is published at the same rate as the simulation, 20 Hz. The odometry is published in `<robot_name>/odom` topic.

Position: The Position component simulates the localization system available in every real-world mobile robot. The localization system used readings from proprioceptive and exteroceptive sensors, such as odometry and LIDAR, respectively, to estimate the robot's pose relative to the map's origin. As we are using a simulator, the localization system itself does not exist. The robot's pose in the map is published directly from the simulator. The localization is published at the same rate as the simulation, 20 Hz. The localization is published in `<robot_name>/pose` topic.

ROS Navigation Stack: The ROS Navigation stack takes information from odometry, exteroceptive sensors, and a navigation pose yields velocity references for the mobile base to follow. It is a navigation component that handles all the navigation problems like path planning, obstacle avoidance, and velocity control. The main loop runs at 15 Hz. The main program is the *move_base* that receives sensor feed – from Odometry, LIDAR, and Position components to know where the robot is. It receives the navigation goal sent by the Sequencer during the Navigation Skill execution. As output, it drives the robot using the `<robot_name>/cmd_vel` topic to send the velocity commands and use the sensors as feedback.

4.3.2.2 Implemented Modules

Battery A battery usually is the main power source for a mobile robot. As our robot is supposed to run for hours answering requests from the hospital crew, a power outage event has to be considered. In order to simulate such component, we added a new module as a new component inside each robot. The module is instantiated with two parameters: (i) an initial battery percentage L_{batt} and (ii) a discharge rate C. The battery module works as a sensor that updates its state every second decreasing the discharge rate from the capacity, then uses a ROS topic to publish the battery state. When the battery reaches $5\ \%$, Battery Module forcefully shuts down the robot's movement, simulating a power outage. As Battery Module works as a sensor, its interface is a ROS topic – `<robot_name>/battery` – where the simulator publishes the Battery State message[4] every second of the simulation.

Item Exchanger In order to get the sample from the nurse to the mobile robot, we implemented an *Item Exchanger*[5] Module. This module uses the BLE API[6] to change the ownership of the samples from the human to the robot. The component receives the new owner in `/<robot_name>/exchange`. The message should be the object label as a string, then the component transfers the ownership to the robot and teleports the object to be on top of the robot.

Drawer The *Drawer* is supposed to be a physical component where the robot can securely carry objects. As this project's scope does not aim for a real platform, there is no need to simulate the physics of such component. So, we implemented the component as an interface using ROS topics where a higher component can control the component, as it would be in real life. The *Drawer* receives a command in the topics `<robot_name>/drawer_open` and `<robot_name>/drawer_close`. Each topic receives a ROS String message,[7] and the action is successful when the string matches the topic name.

[4]`http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/BatteryState.html`

[5]Vicente Moraes discovered how to use the BLE API to transfer objects ownership between agents inside MORSE. He implemented the first version of Item Exchange. *Thank you!*

[6]MORSE is built on top of Blender.

[7]`http://docs.ros.org/en/noetic/api/std_msgs/html/msg/String.html`

4.3.3 Lab Samples Skills

For the Lab Samples Mission, we implemented a skill to handle an end effector. A drawer is a hardware component controlled by a command sent in a ROS topic. The *ActionDrawer* skill controls this component, as shown in Figure 4.3.

Action Drawer: Skill used to either open and closed the Drawer. The *Action Drawer* behavior is relatively straightforward. It requests the Drawer component, a controller, or a hardware interface, to *Open* or *Close*, waits for the action to conclude, and checks if the component executed the action. The skill reported status is the root node status, *success* if the last condition is a *success*, or *failure* if any leaf node fails. Figure 4.3 shows this Skill Implementation.

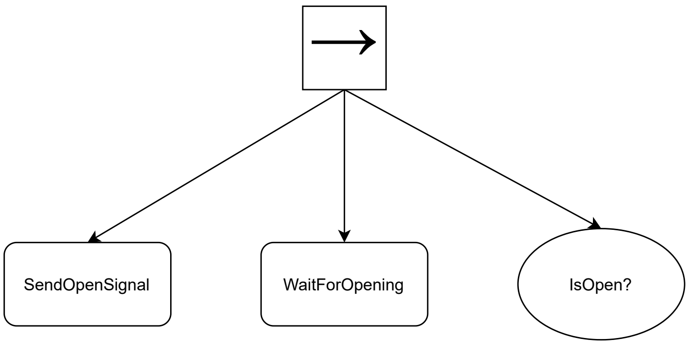


Figure 4.3 – Action Drawer

4.3.4 Leaf Nodes for Action Drawer

In this subsection, we define the Leaf nodes for the *ActionDrawer* skill.

4.3.4.1 Action Nodes

SendSignalDrawer: Whenever this action receives a tick, it sends a signal for the *Drawer* component to *Open* or *Close*. The action return *success* if the signal is sent successfully. It returns *failure* otherwise.

WaitForAction: Whenever this action receives a tick, it waits for a message from the *Drawer* component updating its state. The action return *success* if a response is received. The updated state is stored in a blackboard variable. It returns *failure* if the timer expires and a response is not received.

4.3.4.2 Conditional Nodes

IsAction?: Whenever this condition receives a tick, it checks if the updated state stored in the is the same as the requested. The action return *success* if the *Drawer* conditional is *True*. It returns *failure* otherwise.

4.4 ROS PACKAGES CREATED

Some packages used in this work are not entirely related to the main topic of the presented work. These topics are briefly explained below and should be seen as auxiliary packages as they hold configuration files, world knowledge, and auxiliary code.

4.4.1 motion_ctrl

The *motion_ctrl* package has the main purpose of configuring the navigation stack for each simulated robot and implementing the interface used to integrate the ROS1 navigation stack with the BT in ROS2.

The ROS1 navigation stack uses the *Actionlib*[8] interface to receive navigation goals. As in our architecture, the goals are managed by the *NavTo* BT. We have to use the ros1_bridge package to integrate ROS1 and ROS2 software environments. However, the *Actionlib* interface is not implemented yet. So, to work around this problem, we implemented a new node to receive navigation goals as standard messaging from the ROS2 BT, send the received goals to the navigation stack via *Actionlib*, and update the BT when the goal is reached or not.

Besides this new node, the *motion_ctrl* package holds the configuration for the motion controllers and planners. Available in our repository on Github.[9]

4.4.2 turtulebot3_hospital_sim

The *turtulebot3_hospital_sim* package has the software used to simulate the robots. The software implements the modules described in subsection 4.3.2.2. Available in our repository on Github.[10]

[8] `http://wiki.ros.org/actionlib`
[9] `https://github.com/Gastd/motion_ctrl`
[10] `https://github.com/Gastd/turtlebot3_hospital_sim`

4.4.3 hmrs_hospital_simulation

The *hmrs_hospital_simulation* package holds the world knowledge and model of the simulated environment used in this work. The model represents the environment described in subsection 4.3.1. Available in our repository on Github.[11]

4.5 TOOLS

Some tools were used to glue together the simulation. The ros1_bridge package helped us make the Morse Simulator and the Navigation Stack work together with our framework that runs in ROS2. The containerized simulation saved much effort in developing and setting different environments. Lastly, a simple Python script sets runs, and shuts down the containers autonomously.

4.5.1 ros1_bridge

The *ros1_bridge*[12] is a package distributed by Open Robotics as a ROS2 package that helps integrate ROS1 and ROS2 projects. *ros1_bridge* provides a network bridge to exchange messages between ROS1 and ROS2 environments. As the BT engine and the skills implementations were developed using the pytrees version for ROS2.

4.5.2 Docker

To test the Simulated Mission alongside our framework in a reproducible manner, we break apart the Simulation Environment in Docker Machines. The Docker containers work as Virtual Machines (VMs), separating the working environment of each machine as it would be if the framework ran in different physical computers in a real-world setting. Figure 4.4 depicts our Simulation Environment. In a top-down view, the first row shows the main executable for each working environment, the second row shows the working environment, and the third row shows which simulation element the working environment represents. The dockerized simulation enables us to have a flexible simulation and deploy a multi-robot simulation if needed. The ‘Robot 1’ box can be duplicated to spawn a second robot and so on without the need for further adjustments.

[11]https://github.com/gabrielsr/hmrs_hospital_simulation
[12]https://github.com/ros2/ros1_bridge

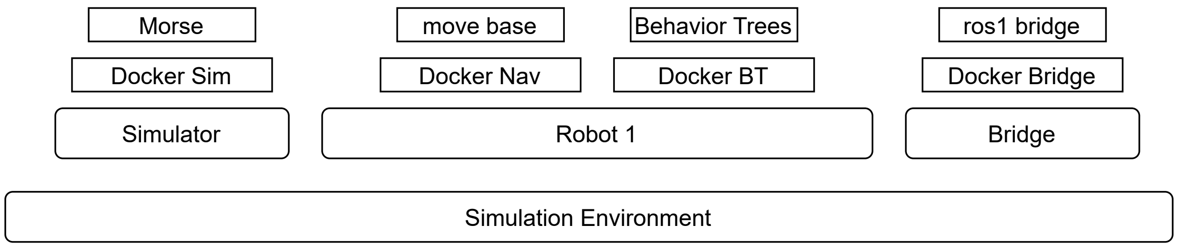


Figure 4.4 – How Docker supports the simulation

4.5.3 Python script

We develop a python script to set the Linux environment variables and call the Docker service to up and run the Docker machine as we please. This script also checks the simulation log searching for a successful mission accomplished or an error event if any of these events happen the script shuts down the Docker machines.

4.6 SIMULATION RESULTS

To run the Simulated Mission, we call our python script that sets the Docker machines to up and run them. After the set-up stage, we send the robot plan to the Sequencer and begin monitoring the execution. We stop the execution when the robot either reports an error or the sample is delivered.

4.6.1 Simulation Setup

We have one Docker machine for the simulator to reproduce a framework with simulated and real-world robots. Bundling the simulator in a virtual machine also allows more than one robot simulation as two or more robots cannot share the same computer. We have another Docker machine for the navigation system. Different from the Sequencing process, the navigation works on ROS. So, a machine was allocated for the navigation stack. For the Sequencer, we have a Docker machine running a ROS2 environment with pytrees.

Lastly, we still need something to interface ROS1 and ROS2 environments. We should note that the shared environment between ROS1 and ROS2 is highly unstable and unpredictable due to differences in how they use environment variables, executable paths, and network usage. For this reason, we chose to instantiate a fourth Docker machine to set and run the *ros1_bridge* package.

4.6.2 Execution

After all Dockers are up and running, a Linux environment variable is used to set the robot plan in JSON format. The Sequencer monitors this environment variable. If the variable is different from `void`, the Sequencer calls the sequencing function, Algorithm 4, in a $1\ Hz$ loop. The Sequencer sends a signal as a report if the task changes its status, either a *success* or a *failure*.

In order to check whatever happens inside the simulation and if the Sequencer is executing the plan, we implemented a simple logger service[13] to persist in a log file: (i) when an Active Skill starts and ends, (ii) if the sample was delivered, (iii) if the robot is with a low battery. In the following sections, we show and describe two executions, one success and one failure, with the support of this logger. We used an automatic script to start and run the executions. The script shuts down the simulation when the logger shows either a successful delivery or a low battery error.

[13]A shout out to Gabriel Rodrigues and Vicente Moraes that implemented the Jupyter notebooks to analyze the logs. Thank you very much! Without the notebook I could have not shown any result from the simulation.

4.7 EXECUTION #1

Table 4.1 – Configuration for Execution #1

L_{batt0}	$23.99\ \%$
C	$0.034\ \%/s$
Robot Initial Room	PC Room 5
Nurse Room	PC Room 2

Before running the first experiment, we configured the environment so the robot could execute its plan with any fault as battery depleted. Table 4.1 shows the parameters used to configure the initial simulation state.

In the first trial, we set the robot to execute a similar plan to the Chapter Implementing Skills with Behavior Trees plan. This plan simulates the mission described in section 4.1. Table 4.2 shows the log data recorded from an execution where the sample was delivered successfully. It seems that "send_message" had not its end state logged even though the Sequencer logged its ending. Also, the last skill, "wait_message", did not show in the log file. Both events happen because the script used to set the test shuts down the simulation right away when the successful delivery event appears.

Figure 4.5 shows the planned path from PC Room 5 to PC Room 2 and its waypoints. Figure 4.6 shows the robot navigating towards the nurse. The red path is the robot's motion, the green circle is the robot's footprint, the axis is the nurse position, and the white dots are the LIDAR's measurements. Like Figure 4.5, Figure 4.7 shows the waypoints planned to robot go to the Lab from PC Room 2. Figure 4.8 shows the robot going to the Lab. The displayed elements are the same as the previous figure. After the receiving the sample the robot moves to the Lab. In both paths, the robot reaches

Table 4.2 – Recorded log from a successful running

skill	label	start_time	end_time	expent_time	end_state
navigation	navto_room	9.62	122.92	113.30	success
approach_person	approach_nurse	122.92	123.68	0.76	success
authenticate_person	authenticate_nurse	123.70	124.13	0.43	success
operate_drawer	open_drawer_for_nurse	124.17	125.28	1.11	success
send_message	notify_nurse_of_open_drawer_completed	125.30	129.48	4.18	success
wait_message	wait_nurse_to_complete_deposit	129.48	130.48	1.00	success
operate_drawer	close_drawer_nurse	130.48	131.65	1.17	success
navigation	navto_lab	131.67	245.82	114.15	success
approach_robot	approach_arm	245.83	247.02	1.19	success
operate_drawer	open_drawer_lab	247.02	248.25	1.23	success
send_message	notify_lab_arm_of_open_drawer_lab_completed	248.27	249.32	1.05	success

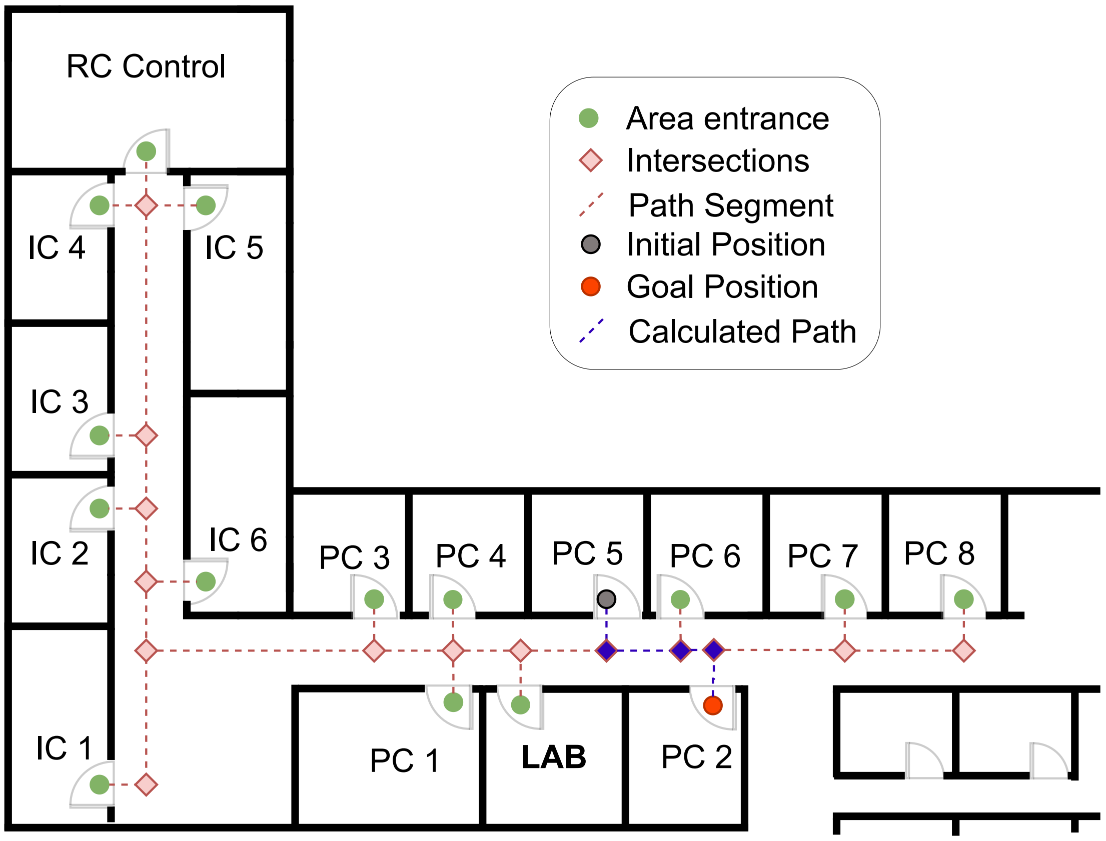


Figure 4.5 – How Docker supports the simulation

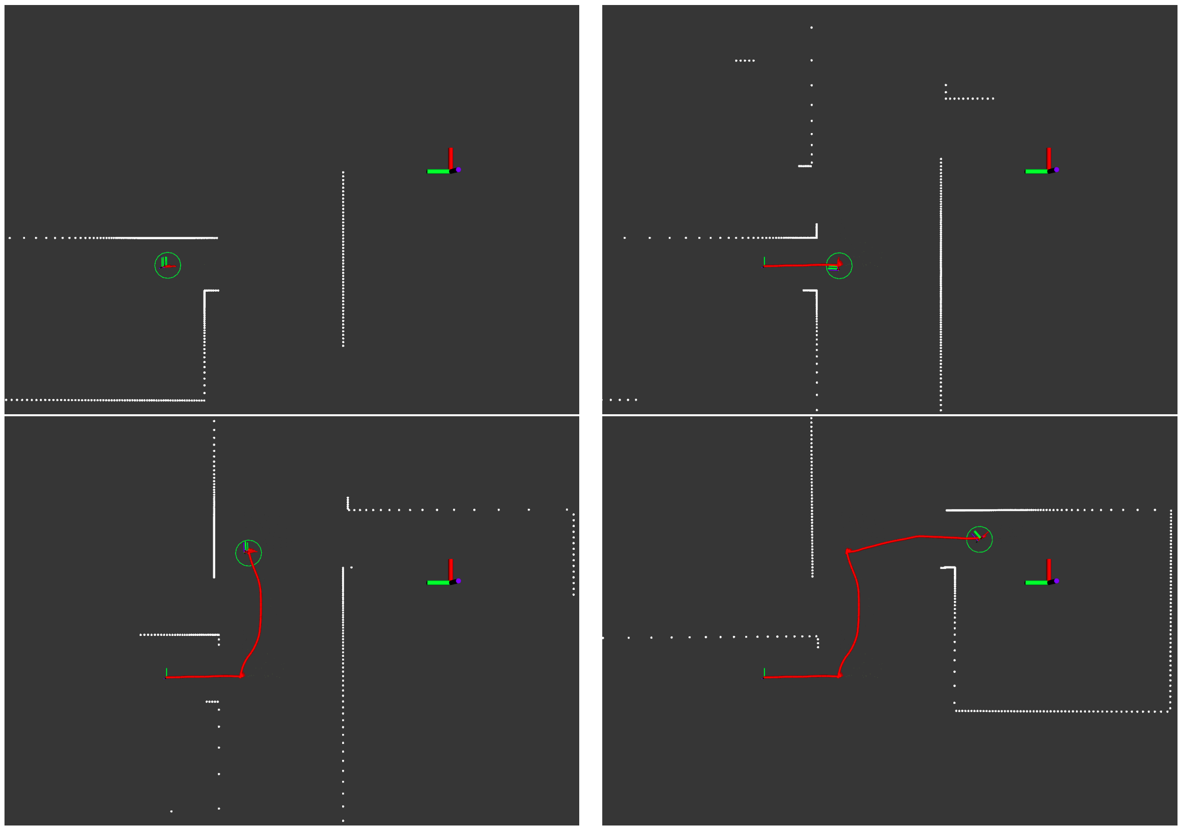


Figure 4.6 – Simulated Environment

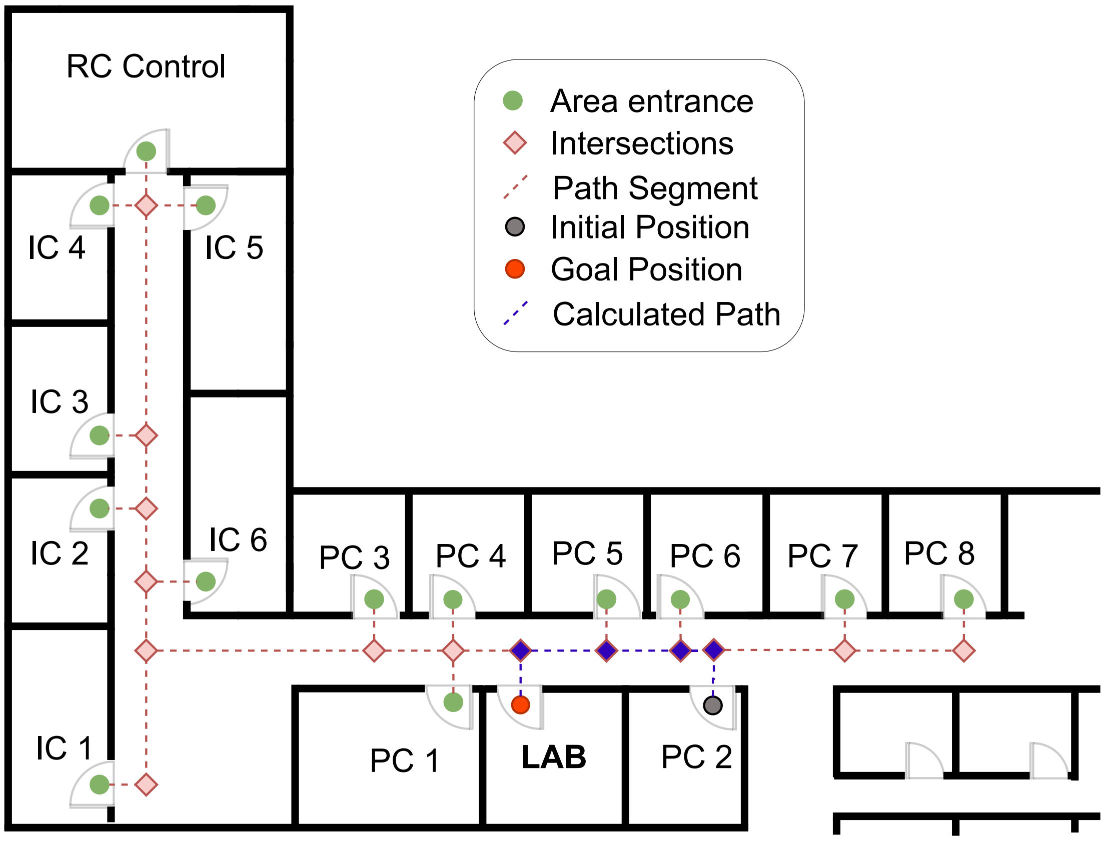


Figure 4.7 – How Docker supports the simulation

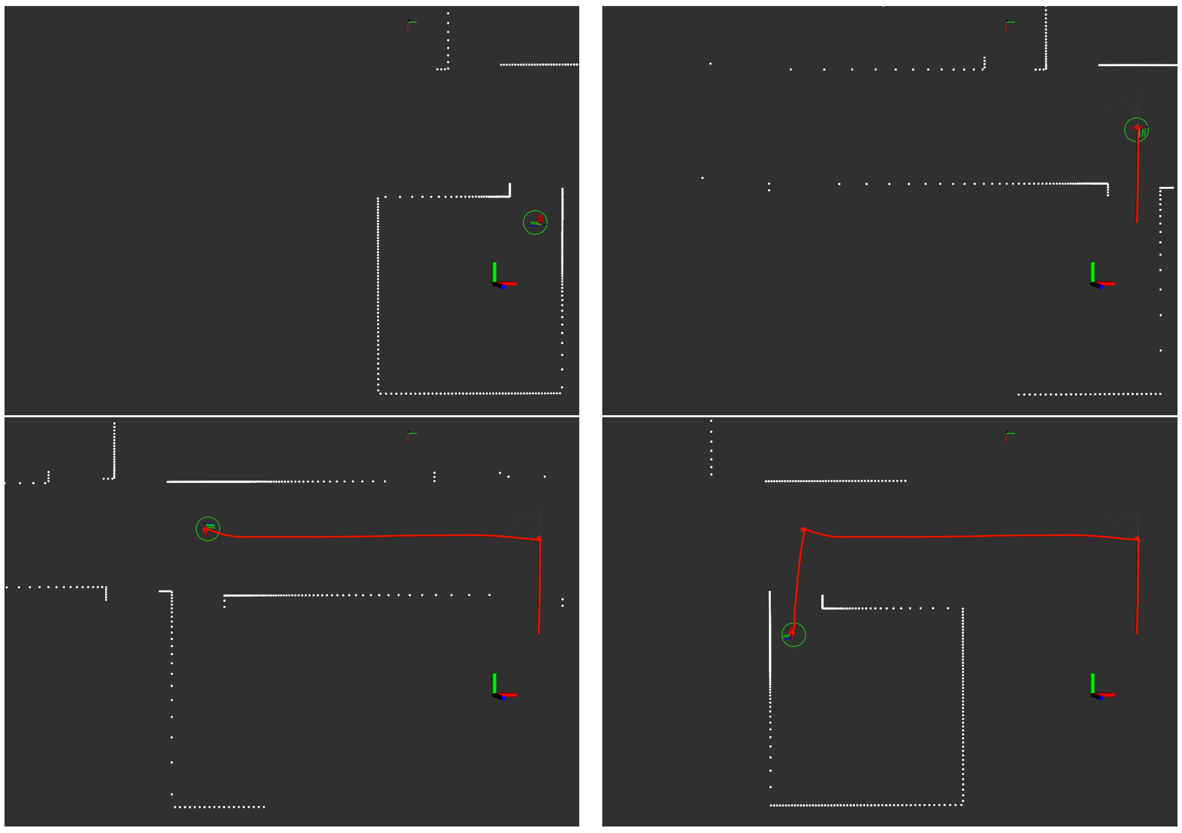


Figure 4.8 – Simulated Environment

4.8 EXECUTION #2

Table 4.3 – Configuration for Execution #2

L_{batt0}	$17.49\ \%$
C	$0.06\ \%/s$
Robot Initial Room	IC Room 5
Nurse Room	IC Room 6

Execution #2 was configured to force a fault from the robot's side. In order to simulate the fault, we configured the robot with slightly less battery, and we increased the discharge rate. As in the first experiment, the Initial Robot Room and Nurse Room are close to the Lab Room's delivery point. We also moved them away to distant rooms. It is expected that the robot to run out of battery at some moment during the plan execution. Table 4.3 shows the used configuration for the second execution.

In the second execution, we executed the sequence of skills as the first one. Table 4.4 shows the execution result. All the skills work well until it comes to the last navigation skill, where the robot goes from the Nurse Room to the Lab Room. As happened in the first execution, the last skill did not register its end state because the script shut down the simulation before the log registered.

It is important to note that the simulated error occurs when the battery module goes to less than $5\ \%$, and it is sensed by the reactive layer, then when reactive layer recognizes that nothing can be done, it warns the plan layer.

Table 4.4 – Recorded log when the robot reaches a low battery level during execution

skill	label	start_time	end_time	expent_time	end_state
navigation	navto_room	18.17	205.15	186.98	success
approach_person	approach_nurse	205.15	205.80	0.65	success
authenticate_person	authenticate_nurse	205.82	206.28	0.46	success
operate_drawer	open_drawer_for_nurse	206.28	207.18	0.90	success
send_message	notify_nurse_of_open_drawer_completed	207.20	210.33	3.13	success
wait_message	wait_nurse_to_complete_deposit	210.33	212.15	1.82	success
operate_drawer	close_drawer_nurse	212.15	213.07	0.92	success
navigation	navto_lab	213.07	218.63	5.56	low-battery

4.9 ISSUES & LIMITATIONS

This section compiles some of our problems, thoughts, and framework limitations based on the current literature. We go further and beyond the scope of this work, describing possible problems with extensions because we think some of our ideas could thrive in domains like multi-robot.

4.9.1 Issues

During the development of this project, we faced several issues. Although we already knew what skills to implement, the first issue we did have was that we did not knkow how much action a single skill should coordinate. For example, the Navigation Skill sends the waypoints to the Navigation System without knowing what happens. In this example, it is okay when a waypoint is reached, but when something blocks the path, the BT does not know, and it can not report anything more than just a "navigation error". Also, it is pretty challenging to design the requirements and goals for each Skill if the execution is detached from the planning. Some kind of structured interface needs like the preconditions and postconditions seen in planning languages like in the PDDL[14] family.

Another problem encountered was related to the *ros1_bridge*. This package is still beta, and quite a few features are missing. The ROS Navigation Stack uses an interface not implemented in *ros1_bridge*. Thus, we made an intermediate node to work around this problem. However, we already did not have enough data about the navigation execution. We lost even more feedback because of the lack of this interface.

The ROS1 does not have any reliability in delivering its messages. In this work. We had a lot of synchronization problems due to the messages passing from ROS2 to ROS1 without any trustworthiness.

4.9.2 Limitations

Regarding planning problems, the scope of this work is to match an already planned setting like a recipe without further refinements. Nevertheless, when we are in the execution domain in real-world applications, a feature much needed that rules us all: time. Our approach works well with the recipe given. However, a lot of time management it had to be done inside each skill, like timeouts and so on. We do not have implemented any time limit for the given tasks yet. This problem hinders the usage of our approach on a commercial scale.

Even though it is not the scope of this project to target Multi-Robot systems, the framework has limitations to further explore essential issues in the Service robots context. Our framework does enable communication between different kinds of agents. However, there is still no strong cooperation between them. Our approach goes as Murphy defines as *Weakly Coordinated* where the robots know about other agents, where each agent has its own time to act, but the robots do not work together in the same Skill. A couple of advantages are: (i) we reduce the problem of *task interference*, and (ii) we enable coordination between hetero-

[14]`https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language`

geneous agents.[15] On the downside, our framework does not support skills that need more than one agent, e.g., carrying a box together with a human or another robot.

Finally, the framework does not support shared memory or knowledge representation. In other words, if the world map changes, each robot has to update its knowledge representation. As such, the framework does not support skills that need support or information from other agents. For example, a hospital could have a set of cleaning robots deployed every night to clean the floor of the corridors of the whole hospital. We could turn this problem into an exploration problem where the robots have to explore the hospital, cleaning its floor 'x' times. It would be a waste of time if the same area is cleaned more than 'x' times, so the robots must share and keep track of their partners' paths. Even though we could have a component to enable the shared work, our approach cannot coordinate such component.

[15]Human and mobile robot, and mobile robot and manipulator

5 CONCLUSION

This work implemented DBTs and used them to bridge the gap between standard BTs and Classical Planning. In order to diminish this gap, we implemented a framework where it is possible to receive the output from a symbolic planner and execute its steps using BTs. Our work shows how a Behavior Tree can coordinate several robotics components available in a Service robot showing how DBTs are suitable for implementing the Skill Layer.

We evaluate the framework in a simulation environment, so we built the world, the robots, and the components needed based on a specified mission. Then, we plugged everything to see if our framework indeed coordinates a robot to execute the intended plan. We present two executions to display how our framework works. In the first one, we configured the robot to not stop during its navigation skill and allocated it near the Nurse and to the delivery room. As expected, the robot goes to Nurse, receives its delivery item, and delivers it to a Manipulator in the Lab. The recorded log points that the Simulation dies before the two last skills finish their execution. This flaw happens because the program that manages the Simulation execution shuts down the simulation right when the sample arrives at the Lab.

Our approach exhibit how well-known 3-Tiered Architecture can work with nowadays popular BTs. We also display how to implement skills that depend on runtime parameters and how a classical planner can help instantiate robotics behaviors to execute a plan. At last, we execute in a simulated mission a deliberative plan showing two cases. The first the plan is executed as it should. In the second, the plan experience a failure and the Sequence recognizes it and notify the planning layer.

We do not tackle any specific multi-robot problem, but we can not escape from them in a service robots context. Hence, our framework tries to be flexible enough to deal with multi-robot issues. Here we compile some multi-robot problems mentioned by Murphy and how our approach solves them:

P: "*How is programming multiple robots different from programming a single robot?*" A: As our framework detaches the deliberative planner from the Sequencing and Skill Layers, the same structure works for multiple robots. Each robot can access a shared Skill library to retrieve skills in runtime to execute a plan. The same architecture works for both single and multi-robot missions.

P: "*It is not clear when communication is needed between agents or what to say*". A: The plan already solves when needed communication between who, but our framework shows how this happens and how different skills can deliver different types of communication.

P: "*The 'right' level of individuality and autonomy is usually not obvious in a problem* *domain*". A: The architecture used draws a line in the Sequencer Layer, separating the concerns between the robot and the planner.

Our framework tackles the problem of interfacing deliberative and reactive functionalities in a hybrid system with a Sequencer running Dynamic Behavior Trees as the Skill Implementation. The proposed framework shows how a reactive system handles and follows a deliberative input using DBTs. Our Dynamic Behavior Trees are not so different from the usual BTs. We also show how DBTs encapsulate everything needed to execute a given task.

5.1 RELATED WORK

There are not so many works that propose a Dynamic Behavior Tree as we do in this project. So, we compiled some work that correlates to some ideas implemented here. In 2008, Flórez-Puga et al. used the recurring patterns in Behavior Tree design to implement a BT that is expanded in runtime. They hint about the drawback that is having to redesign a large and already statically designed BT. Their solution is to expand some BT nodes in runtime, substituting the query nodes by a subtree [42]. In the planning and acting realm, Colledanchise et al. try to solve some issues brought by Ghallab et al. in [38, 20] using a planner inspired by the Hybrid Backward-Forward algorithm to create BTs in runtime to achieve a given goal [11]. Cai et al. extend Colledanchise et al.'s work [11] presenting a sound and complete algorithm to expand BTs to find a planning solution, bridging the gap between BT representation and formal analysis [43]. In 2021, Mayr et al. bridge the gap between BTs and Machine Learning using BTs to solve execution issues related to optimized policies using Reinforcement Learning [44].

5.2 FUTURE WORK

As future work, we devise several improvements in the presented work. However, before jumping to extensions, we need to know the earnings and losses in terms of implementation. So, we propose several tests to analyze the overhead growth of using DBTs against a Static BT baseline. The test can also show how different and difficult it is to develop a Static BT that can execute the same plan in the same way. As features, we propose the possibility to prune and insert new branches in the BT to deal either with faults or to replanning. Currently, our Sequencer works with only one DBT each time. An extension of the Sequencer is the ability to work with concurrent Skills. This feature adds a lot more flexibility to a more complex plan where some tasks have to be done simultaneously by the same agent. Lastly, each skill tries to achieve its goal, and if an exception occurs, the problem is sent to the planner. However, if the exception needs some robot action, e.g., the robot has to move from a dangerous place, this does not happen until the replan. Thus, we propose a “fault-tolerant” skill that assumes the robot behavior when something goes wrong and handles any problem until a new plan arrives.

BIBLIOGRAPHY

[1] R. R. Murphy, *Introduction to AI Robotics*, 2nd ed. Cambridge, MA, USA: MIT Press, 2019.

[2] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*, 2nd ed. Springer Publishing Company, Incorporated, 2016.

[3] ISO Central Secretary, "Robots and robotic devices – vocabulary," International Organization for Standardization, Geneva, CH, Standard ISO 8373:2012, 2012. [Online]. Available: https://www.iso.org/standard/55890.html

[4] H. Christensen, N. Amato, H. Yanco, M. Mataric, H. Choset, A. Drobnis, K. Goldberg, J. Grizzle, G. Hager, J. Hollerbach, S. Hutchinson, V. Krovi, D. Lee, B. Smart, J. Trinkle, and G. Sukhatme, *A Roadmap for US Robotics – From Internet to Robotics 2020 Edition*, ser. Foundations and Trends in Robotics Series. Now Publishers, 2021. [Online]. Available: https://books.google.com.br/books?id=zYGPzgEACAAJ

[5] Horizon 2020, "Robotics 2020 Multi-Annual Roadmap," 2015. [Online]. Available: https://www.eu-robotics.net/sparc/upload/about/files/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf

[6] M. J. Mataric, *The Robotics Primer (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2007.

[7] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 5420–5427.

[8] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI*. CRC Press, Jul 2018. [Online]. Available: http://dx.doi.org/10.1201/9780429489105

[9] M. Colledanchise and P. Ögren, "How Behavior Trees modularize robustness and safety in hybrid systems," *IEEE International Conference on Intelligent Robots and Systems*, pp. 1482–1488, 2014.

[10] M. Colledanchise and P. Ogren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, 2017.

[11] M. Colledanchise, D. Almeida, and P. Ögren, "Towards blended reactive planning and acting using behavior trees," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2019-May, pp. 8839–8845, 2019.

[12] M. Colledanchise, G. Cicala, D. E. Domenichelli, L. Natale, and A. Tacchella, "Formalizing the execution context of behavior trees for runtime verification of deliberative policies," 2021.

[13] E. B. Gil, "Multi-Robot Missions Decomposer: MutRoSe Mission Decomposer First Official Release," oct 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5584561

[14] F. W. P. Heckel, G. M. Youngblood, and N. S. Ketkar, "Representational complexity of reactive agents," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 2010, pp. 257–264.

[15] J. A. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, M. Pivtoraiko, J.-S. Valois, and R. Zhu, "An integrated system for autonomous robotics manipulation," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, Oct 2012, pp. 2955–2962.

[16] M. Slack, "Navigation templates: mediating qualitative guidance and quantitative control in mobile robots," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 2, pp. 452–466, 1993.

[17] R. J. Firby, R. E. Kahn, P. N. Prokopowicz, and M. J. Swain, "An architecture for vision and action," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, vol. 1, 1995, pp. 72–79.

[18] E. Gat, "Reliable goal-directed reactive control of autonomous mobile robots," in *Doctoral Dissertations from Virginia Tech*, 1991.

[19] R. J. Firby, P. N. Prokopowicz, M. J. Swain, R. E. Kahn, and D. Franklin, "Programming CHIP for the IJCAI-95 Robot Competition," *AI Magazine*, vol. 17, no. 1, p. 71, 1996. [Online]. Available: https://ojs.aaai.org/index.php/aimagazine/article/view/1213

[20] M. Ghallab, D. Nau, and P. Traverso, "The actor's view of automated planning and acting: A position paper," *Artificial Intelligence*, vol. 208, no. 1, pp. 1–17, mar 2014. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2013.11.002https://linkinghub.elsevier.com/retrieve/pii/S0004370213001173

[21] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An Architecture for Autonomy," *The International Journal of Robotics Research*, vol. 17, no. 4, pp. 315–337, 1998. [Online]. Available: https://doi.org/10.1177/027836499801700402

[22] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wąsowski, "Behavior trees in action: A study of robotics applications," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 196–209. [Online]. Available: https://doi.org/10.1145/3426425.3426942

[23] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "Costar: Instructing collaborative robots with behavior trees and vision," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 564–571.

[24] F. Rovida, D. Wuthier, B. Grossmann, M. Fumagalli, and V. Krüger, "Motion generators combined with behavior trees: A novel approach to skill modelling," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 5964–5971.

[25] E. Giunchiglia, M. Colledanchise, L. Natale, and A. Tacchella, "Conditional behavior trees: Definition, executability, and applications," in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, 2019, pp. 1899–1906.

[26] H. Zhou, H. Min, and Y. Lin, "An autonomous task algorithm based on behavior trees for robot," in *2019 2nd China Symposium on Cognitive Computing and Hybrid Intelligence (CCHI)*, 2019, pp. 64–70.

[27] J. A. Segura-Muros and J. Fernández-Olivares, "Integration of an automated hierarchical task planner in ros using behaviour trees," in *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2017, pp. 20–25.

[28] K. R. Guerin, C. Lea, C. Paxton, and G. D. Hager, "A framework for end-user instruction of a robot assistant for manufacturing," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6167–6174.

[29] Splintered-Reality, "Py trees." [Online]. Available: https://github.com/splintered-reality/py_trees

[30] S. Lemaignan, G. Echeverria, M. Karg, J. Mainprice, A. Kirsch, and R. Alami, "Human-robot interaction in the morse simulator," in *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 181–182. [Online]. Available: https://doi.org/10.1145/2157689.2157745

[31] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *International Conference on Robotics and Automation*, 2010.

[32] M. Colledanchise, A. Marzinotto, D. V. Dimarogonas, and P. Ögren, "The advantages of using behavior trees in mult-robot systems," *47th International Symposium on Robotics, ISR 2016*, vol. 2016, pp. 23–30, 2016.

[33] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *IEEE Transactions on Games*, vol. 11, no. 2, pp. 183–189, 2019.

[34] A. Klöckner, *Interfacing Behavior Trees with the World Using Description Logic*. American Institute of Aeronautics and Astronautics, 2013. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2013-4636

[35] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 6793–6800.

[36] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and ai," 2020.

[37] Y.-q. Jiang, S.-q. Zhang, P. Khandelwal, and P. Stone, "Task planning in robotics: an empirical comparison of PDDL- and ASP-based systems," *Frontiers of Information Technology & Electronic Engineering*, vol. 20, no. 3, pp. 363–373, 2019. [Online]. Available: https://doi.org/10.1631/FITEE.1800514

[38] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[39] T. Gateau, C. Lesire, and M. Barbier, "HiDDeN: Cooperative plan execution and repair for heterogeneous robots in dynamic environments," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 4790–4795.

[40] M. Askarpour, C. Tsigkanos, C. Menghi, R. Calinescu, P. Pelliccione, S. Garcia, T. J. von Oertzen, M. Wimmer, L. Berardinelli, M. Rossi, M. M. Bersani, and G. S. Rodrigues, "RoboMAX: Robotic Mission Adaptation eXemplars," in *Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2021.

[41] G. A. Korsah, A. Stentz, and M. B. Dias, "A comprehensive taxonomy for multi-robot task allocation," *The International Journal of Robotics Research*, vol. 32, no. 12, pp. 1495–1512, 2013. [Online]. Available: https://doi.org/10.1177/0278364913496484

[42] G. Flórez-Puga, M. Gómez-Maryin, B. Díaz-Agudo, and P. A. González-Calero, "Dynamic expansion of behaviour trees," *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, pp. 36–41, 2008.

[43] Z. Cai, M. Li, W. Huang, and W. Yang, "BT Expansion: a Sound and Complete Algorithm for Behavior Planning of Intelligent Robots with Behavior Trees," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 7, pp. 6058–6065, 2021. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16755

[44] M. Mayr, K. Chatzilygeroudis, F. Ahmad, L. Nardi, and V. Krueger, "Learning of parameters in behavior trees for movement skills," 2021.

APPENDIX

A SKILL IMPLEMENTATIONS

This appendix list the Algorithms for each Skill Implementation used in this work. Algorithms 7, 8, 9, 10, 11, and 12 are described in Chapter 3. Algorithm 6 is explained in the Chapter 4.

A.1 ACTION DRAWER

Algorithm 6 Action Drawer Implementation

```
Require: length of $param\_list > 0$
 1: function CREATE_ACTION_DRAWER_BT($param\_list$)
 2:     $root$ ← SequenceNode()
 3:     $drawer\_state$ ← GetStringFromJson($param\_list[0]$)
 4:     $newstate2BB$ ← SetBlackboardVariable($drawer\_state$, “new_state”)
 5:     $waitForNewstateBB$ ← WaitForBlackboardVariable(“new_state”)
 6:     $ros\_publisher$ ← PublishFromBlackboard(“new_state”, “set_drawer”)
 7:     $wait\_for\_update$ ← SubscribeToBlackboard(“update_state”, “get_drawer”)
 8:     $update2BB$ ← WaitForBlackboardVariable(“update_state”)
 9:     $isAction$ ← CheckBBVariableValue(“update_state”, $drawer\_state$)
10:     $root$.ADD_CHILDREN([$newstate2BB, waitForNewstateBB, ros\_publisher$])
11:     $root$.ADD_CHILDREN([$wait\_for\_update, update2BB, isAction$])
12:     return $root$
13: end function
```

A.2 WAIT MESSAGE

Algorithm 7 Wait Message Implementation

```
Require: length of $param\_list > 0$
 1: function CREATE_WAIT_MSG_BT($param\_list$)
 2:     $root$ ← SequenceNode()
 3:     $resp\_2BB$ ← SubscribeToBlackboard($param\_list[0]$+“comms”,“resp”)
 4:     $wait\_res$ ← WaitForBlackboardVariable($resp\_msg$, “resp”)
 5:     $isOK$ ← CheckBBVariableValue(“resp”, $param\_list[1]$)
 6:     $root$.ADD_CHILDREN([$resp\_2BB, wait\_res, isOK$])
 7:     return $root$
 8: end function
```

A.3 SEND MESSAGE

Algorithm 8 Send Message Implementation

```
Require: length of $param\_list > 0$
1: function CREATE_SEND_MSG_BT($param\_list$)
2:     $root$ ←SEQUENCENODE()
3:     $msg\_2BB$ ← SETBLACKBOARDVARIABLE($param\_list$[1],“msg”)
4:     $waitForMsgBB$ ←WAITFORBLACKBOARDVARIABLE(“msg”)
5:     $ros\_publisher$ ←PUBLISHFROMBLACKBOARD(“msg”, $param\_list[0]$)
6:     $root$.ADD_CHILDREN([$msg\_2BB, waitForMsgBB, ros\_publisher$])
7:     return $root$
8: end function
```

A.4 NAVIGATION TO

Algorithm 9 NavTo Implementation

```
Require: length of $param\_list > 0$
1: function CREATE_NAV_TO_BT($param\_list$)
2:     $root$ ←SEQUENCENODE()
3:     $way\_seq$ ← WAYPOINT_EXPANSION($param\_list$[1])
4:     $topics2bb$ ←SEQUENCENODE()
5:     $scan2bb$ ←ROSEVENTTOBLACKBOARD(“/RNAME/scan”)
6:     $cancel2bb$ ←ROSEVENTTOBLACKBOARD(“/RNAME/cancel”)
7:     $battery2bb$ ←ROSEVENTTOBLACKBOARD(“/RNAME/battery”)
8:     $battery_emergency$ ←ETERNALGUARD()
9:     $check_batt$ ←CHECKBLACKBOARDVARIABLEVALUE(“battery”)
10:    $battery_emergency$.ADD_CHILD($check_batt$)
11:    $parallel$ ←PARALLELNODE()
12:    $parallel$ADD_CHILDREN([$battery_emergency, way\_seq$)
13:    $root$.ADD_CHILDREN([$topics2BB, parallel$])
14:    return $root$
15: end function
```

A.5 AUTHENTICATE PERSON

Algorithm 10 Authenticate Person Implementation

```
Require: length of $param\_list > 0$
1: function CREATE_AUTH_PERSON_BT($param\_list$)
2:     $root$ ←SEQUENCENODE()
3:     $req\_person$ ← GETSTRINGFROMJSON($param\_list[0]$)
4:     $req2BB$ ←SETBLACKBOARDVARIABLE($req\_person$, "req_person")
5:     $waitForReq$ ←WAITFORBLACKBOARDVARIABLE("req_person")
6:     $publisher$ ←PUBLISHFROMBLACKBOARD("req_person", "/auth/")
7:     $person\_resp$ ←SUBSCRIBETOBLACKBOARD("resp_person", "/fauth/")
8:     $wait\_resp$ ←WAITFORBLACKBOARDVARIABLE("resp_person")
9:     $isAuthenticated$ ←CHECKBBVARIABLEVALUE("resp_person", True)
10:    $root$.ADD_CHILDREN([$req\_person, req2BB, waitForReq, publisher$])
11:    $root$.ADD_CHILDREN([$person\_resp, wait\_resp, isAuthenticated$])
12:    return $root$
13: end function
```

A.6 APPROACH PERSON

Algorithm 11 Approach Person Implementation

```
Require: length of $param\_list > 0$
1: function CREATE_APPROACH_PERSON_BT($param\_list$)
2:     $root$ ←SEQUENCENODE()
3:     $req\_person$ ← GETSTRINGFROMJSON($param\_list[0]$)
4:     $req2BB$ ←SETBLACKBOARDVARIABLE($req\_pose$, "req_pose")
5:     $waitForReq$ ←WAITFORBLACKBOARDVARIABLE("req_pose")
6:     $publisher$ ←PUBLISHFROMBLACKBOARD("req_pose", "/localize_person/")
7:     $person\_pose$ ←SUBSCRIBETOBLACKBOARD("pose_person", "/get_pose/")
8:     $wait\_pose$ ←WAITFORBLACKBOARDVARIABLE("pose_person")
9:     $GoToPose$ ←SENDNAVGOALROSBB("pose_person")
10:    $root$.ADD_CHILDREN([$req\_person, req2BB, waitForReq, publisher$])
11:    $root$.ADD_CHILDREN([$person\_pose, wait\_pose, GoToPose$])
12:    return $root$
13: end function
```

A.7 APPROACH ROBOT

Algorithm 12 Approach Robot Implementation

```
Require: length of $param\_list > 0$
1: function CREATE_APPROACH_ROBOT_BT($param\_list$)
2:     $root$ ←SEQUENCENODE()
3:     $r\_pose$ ←SUBSCRIBETOBLACKBOARD("pose_robot", $param\_list[0]$"/pose/")
4:     $wait\_pose$ ←WAITFORBLACKBOARDVARIABLE("pose_robot")
5:     $GoToPose$ ←SENDNAVGOALROSBB("pose_robot")
6:     $root$.ADD_CHILDREN([$r\_pose, req2BB, wait\_pose, GoToPose$])
7:     return $root$
8: end function
```