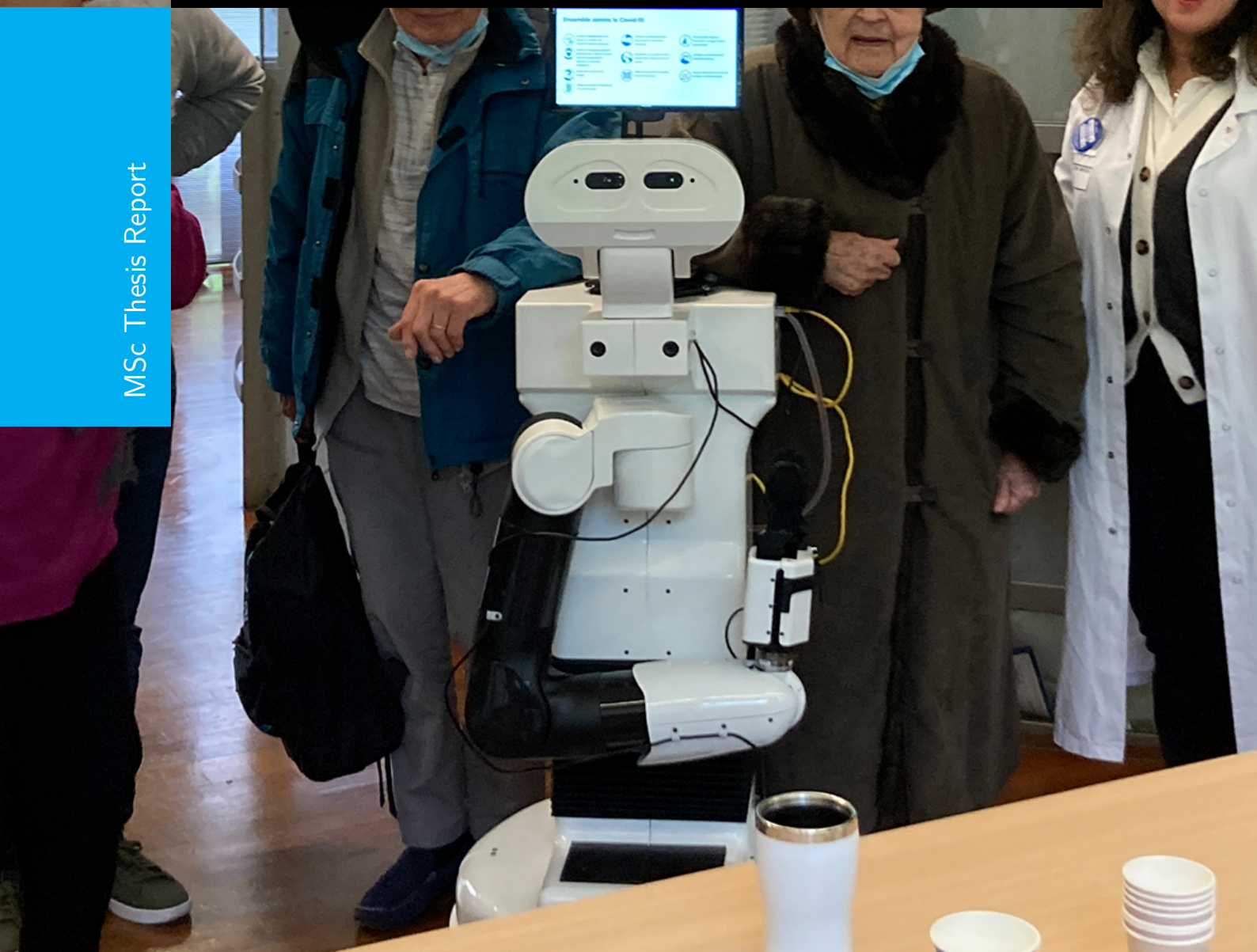


Incremental Adaptation of Behaviour Trees for Applications in Learning from Demonstration Frameworks

Rudy De-Xin de Lange

MSc Thesis Report



Incremental Adaptation of Behaviour Trees for Applications in Learning from Demonstration Frameworks

THESIS REPORT FOR THE DEGREE OF MASTER OF SCIENCE IN
BIOMECHANICAL ENGINEERING WITH A SPECIALIZATION IN BIO-ROBOTICS
AT THE TECHNICAL UNIVERSITY OF DELFT

ME51015

Author:

Rudy De-Xin DE LANGE

Student number:

4304632

Supervisors:

Dr. Cock Heemskerk
Ir. Dimitris Karageorgos
Dr. Jens Kober

Affiliation:

HIT
HIT
TU Delft 3mE

Additional Comity Members:

Dr. Luka Peternel
Ir. Corrado Pezzato

TU Delft 3mE
TU Delft 3mE

May 29, 2022



This work contains the Master Thesis of Rudy De-Xin de Lange with the topic of *Incremental Adaptation of Behaviour Trees for Applications in Learning from Demonstration Frameworks* for the title of Ir. at the faculty of 3mE of the Technical University of Delft. This thesis has been written in cooperation and under contract of Heemskerk Innovative Technology (HIT).

Reading, grading and acknowledgment/approval of this Master Thesis has been done by the following comity:

- Dr. Jens Kober
- Dr. Luka Peternel
- Ir. Dimitris Karageorgos
- Ir. Corrado Pezzato

Abstract

This thesis proposes the novel *Behaviour Tree Update Framework* (BTUF) for the initial construction and continuous incremental adaptation of Behaviour Trees (BTs) for applications in Learning from Demonstration (LfD) frameworks to create complex robot behaviours associated with Activities of Daily Living (ADL) without requiring the user to have a programming or engineering background. BTUF implements several methods towards that end. Automatic generation of the fundamental structure of BTs within BTUF allows for easy human operation of the framework's Text-based User Interface (TUI). Saving and loading of constructed trees facilitates easy expansion and reusability of constructed trees. By expanding upon an initial base behaviour, seemingly simple behaviours can be adapted to facilitate novel instances thereupon, increasing the complexity and functionality of the constructed tree over time. Experimental validation in the form of a user study has provided proof-of-concept within simulation and has given insight in the initial overall performance and general system acceptance of BTUF. Future work is recommended for further validation and improvement of the proposed framework to one day realise a real-life application within healthcare robotics.

Acknowledgments

I would like to thank *Cock Heemskerk*, *Dimitris Karageorgos*, *Luuk Doornebosch* and all of those working at *Heemskerk Innovative Technology (HIT)* for having faith in me, providing me with an internship and graduation assignment.

Special thanks to my academic tutor *Jens Kober* for taking me underneath his wing, the invaluable tutelage, the helpful yet fun meetings and referring me to *HIT* in the first place. I am very grateful to have had you as my thesis professor.

Also, I would like to thank *Corrado Pezzato* for all the discussions and insightful feedback by taking the time to discuss my thesis so many times although I was not officially one of your students.

Thanks to *Carlos Hernandez Corbato* who initially agreed to be part of my comity and to *Luka Peternel* for being able to take his place due to personal circumstances. Lastly, I would like to thank everyone that has supported me throughout the final stages of my academic life as a masters student.

Contents

1	Introduction	5
1.1	An Increasingly Ageing Society	5
1.2	Use-Case & Scenario	6
1.3	Robot Learning and Task Execution	7
1.3.1	Human Demonstration Methods	8
1.3.2	Behaviour Learning and Task Execution	8
1.4	Research Gap, Prior Recommended Future Work & Proposed Project Goal	9
1.5	Main Contribution	9
2	Background	10
2.1	Learning from Demonstration	10
2.1.1	Solving Learning from Demonstration Tasks	10
2.2	High Level Task Planning using Behaviour Trees	11
2.3	Decision Trees	12
2.4	How Behaviour Trees Generalize Other Behavioural Programming Ideas	14
2.5	Behaviour Trees	15
2.5.1	Overall Concept of Behaviour Trees	15
2.6	Methods for Learning and Adapting Behaviour Trees	21
2.6.1	Backward Chained Behaviour Trees	21
2.6.2	Automated Planning & Behaviour Trees	21
2.6.3	Genetic Programming & Behaviour Trees	22
2.6.4	Reinforcement Learning & Behaviour Trees	22
2.6.5	Conversion of Learnt Decision Trees to Behaviour Trees	23
2.7	Chapter Discussion	23
3	Proposed Method	25
3.1	Basic Principles	26
3.2	Proposed Pipeline	28
3.2.1	Forward Chaining	29
3.2.2	Backward Chaining	29
3.2.3	Forward versus Backward Chaining	30
3.2.4	Backward Chained Behaviour Trees	31
3.3	Practical Implementation	35
3.3.1	Proposed Framework Features	35
3.3.2	Node Types	35
3.3.3	Behaviour Tree Structures	36
3.3.4	Constructing and Printing a Behaviour Tree	39
3.3.5	Adding Robot Skills	40
3.3.6	Adding Conditions	41
3.3.7	Backward Chaining	41
3.3.8	Removing Actions, Conditions, Backward Chained Branches and Parallel Behaviours	43

3.3.9	Saving & Loading Behaviour Trees	44
3.3.10	Creating Parallel Behaviours	45
3.3.11	Executing a BT	47
3.3.12	Full Behaviour Tree Update Framework	47
3.3.13	Ensuring Robustness	47
3.3.14	Conventional Framework	48
4	Validation	49
4.1	Experiment Setup	49
4.1.1	Introduction	49
4.1.2	Method	49
4.2	Hypotheses & Initial Expectations	52
4.3	Experimental Results	53
4.3.1	Preliminary Study Results	53
4.3.2	Final Experiment Results	56
4.4	Overall Conclusion Towards the Validation of the Own Proposed Method	59
4.5	Discussion	61
5	Concluding Chapter	66
5.1	Summary	66
5.2	Future Work	68
5.3	Concluding Remarks	70
6	Appendices	71
	Appendix I: Participant Feedback & Comments	72
	Appendix II: Participant Handout & Form of Written Consent	73
	Appendix III: Experimental Validation Proposal	85
	Bibliography	94

Nomenclature

This list contains abbreviations used throughout this literature study report.

Abbreviations

AC	Actions taken to Completion
ADL	Activities of Daily Living
BB	Blackboard
BC	Backward Chained
BT	Behaviour Tree
BTCF	Behaviour Tree Creation Framework
BTUF	Behaviour Tree Update Framework
DMP	Dynamic Movement Primitive
DOF	Degrees of Freedom
DT	Decision Tree
FSM	Finite State Machine
GUI	Graphical User Interface
HFSM	Hierarchical Finite State Machine
HIT	Heemskerk Innovative Technologies
ID	Identity
KT	Kinesthetic Teaching
LfD	Learning from Demonstration
ML	Machine Learning
MP	Movement Primitive
NLP	Natural Language Processing
PMP	Probabilistic Movement Primitive
RC	Runs to Completion
TBTN	Total BT Nodes in final tree structure
TC	Time to Completion
TU Delft	Technical University of Delft
TUI	Text-based User Interface

Chapter 1

Introduction

1.1 An Increasingly Ageing Society

World Health Organization (2018) forecasts that by 2050 the entirety of the world population will consist of 22 percent of people aged 60 and above. This is an increase of more than double of today's percentage of 10 percent. This trend of an ageing society is seen clearly in Figure 1.1 and is found back in multiple publications such as Ellison et al. (2015), Hyun et al. (2015), Kim et al. (2015) and StatisticKorea (2017). From these publications we find that the United States of America as well as South-Korea are both expected to be super-aged countries in less than half a century with the first having 1 out of 5 citizens being the age of 65 and above, whilst the latter country is expected to have a median age of 62.1 by the year 2063. Chia and Loh (2018) identified that an increasingly ageing society has a higher demand in both hospital and general healthcare services as a consequence. This increasing demand in healthcare services will unfortunately be paired alongside high changes of increasing shortages in available medical personnel (World Health Organization, 2018).

In short it can be identified that an ever continuous aging-population brings along an increasing need for necessary healthcare improvements in the hope to tackle the correlated surge in demand for healthcare services, appliances and personnel.

To fill out shortages in medical personnel, one can rely on the increasing advances in modern robotics engineering. Hosseini and Goher (2017) identified an increasing need for personal care robots with the correlated decrease in available qualified human medical employees. From Smarr et al. (2014) we find that higher-aged adults are open to and prefer robots to assist them in performing Activities of Daily Living (ADL). By the definition of Edemekong et al. (2020) we find that ADL are tasks performed on a daily basis, considered to be fundamental to care for oneself. ADL are tasks that vary broadly amongst one another ranging from washing oneself, preparing food, grooming oneself and etcetera. The inability to perform one or multiple ADL can be considered as a decrease in the quality of life for said individual. Assistive healthcare robotics capable of autonomously performing ADL are proposed to tackle the increasing shortages in available human medical personnel.

Median age, 1950 to 2100

The median age divides the population into two parts of equal size; that is, there are as many people with ages above the median age as there are with ages below.

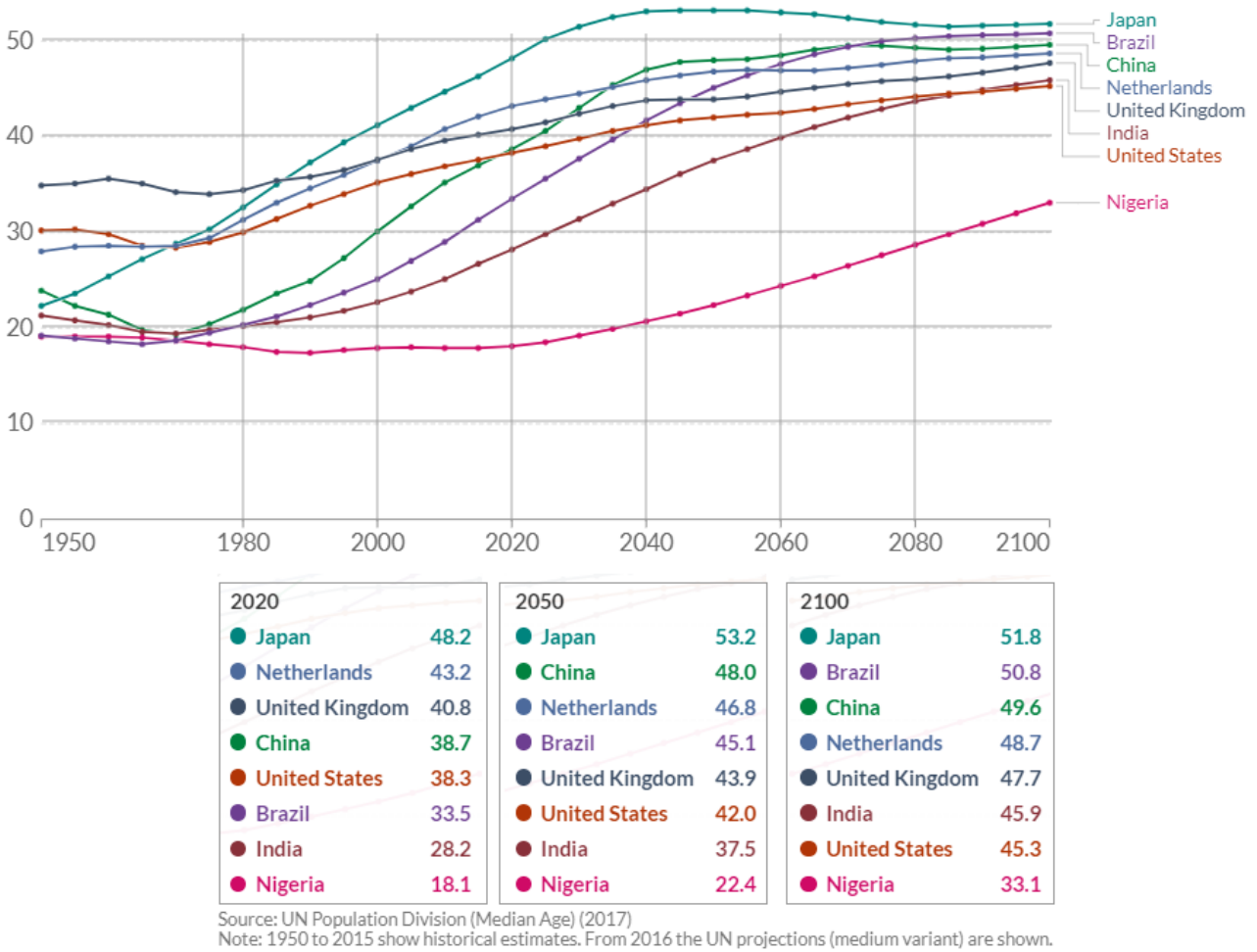


Figure 1.1: Median age for 8 different countries from 1950 to 2050 as adapted from Ritchie and Roser (2019).

1.2 Use-Case & Scenario

The use-case for this thesis project is given as:

"A robotic nurse/assistant for use in healthcare environments such as care-homes and hospitals capable of performing everyday assistive tasks within close proximity of human individuals."

A scenario is drawn wherein a healthcare robot learns and updates behaviours necessary for performing ADL within a healthcare environment given a base task and subsequent variations thereon that occur during repeated re-execution of said given task

Within the new scenario, as depicted in Figure 1.2, a robot is asked to pick up an object from the table. After learning the base task, the robot encounters an unforeseen and prior unknown obstacle upon reproduction of the learnt behaviour, hindering a safe and successful execution of the picking task. In this scenario, as retrying the learnt behaviour will most likely result in repeated failure, the robot asks a human operator for aid. Upon looking at the new instance of the picking task, the human operator demonstrates a correct behaviour for successful execution of the novel task instance after which the robot will have learnt to do so as well. Conclusively and most desirably the robot has expanded upon the learnt base task, by extending its knowledge of additionally performing another variation of the initially learnt task.

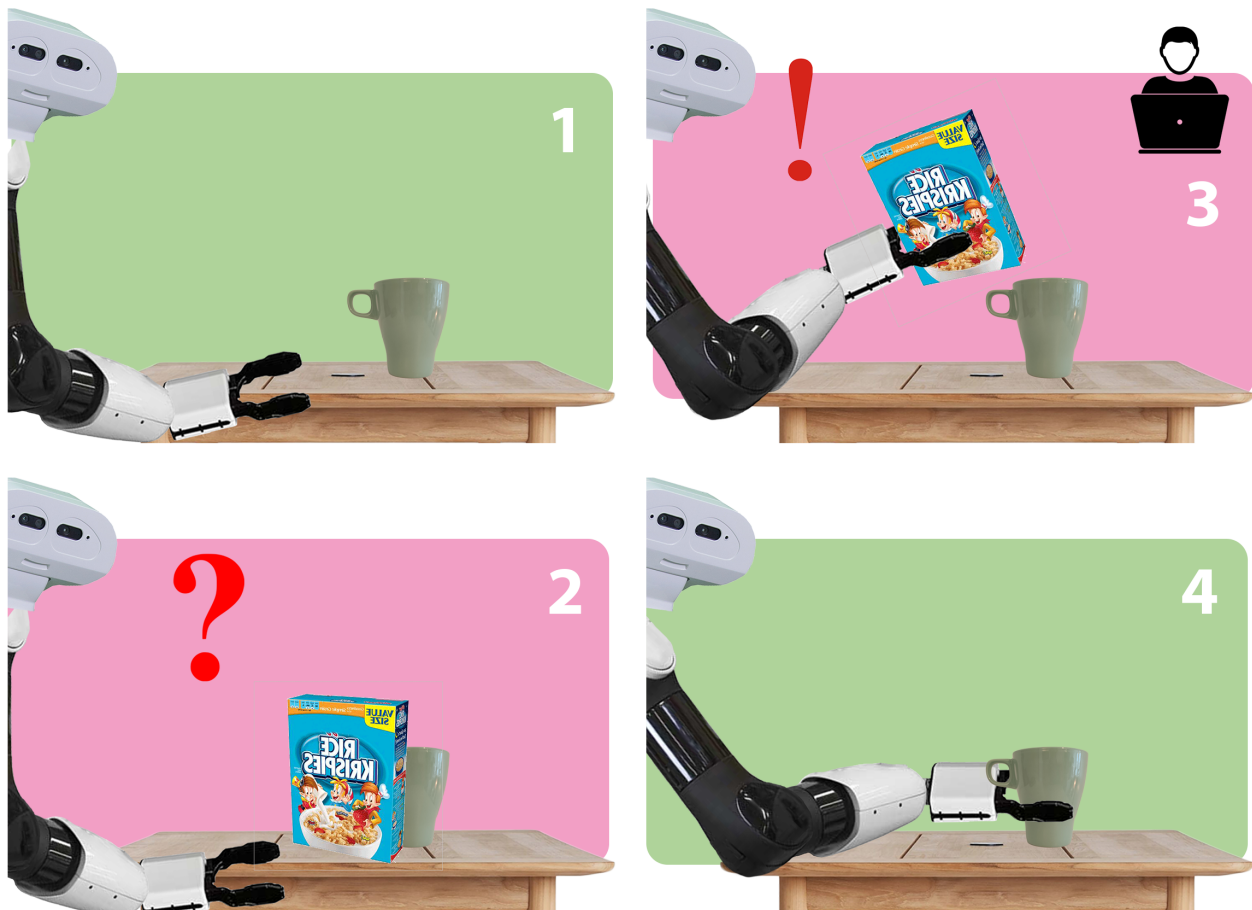


Figure 1.2: In green the base pick-and-place behaviour that the robot is capable of autonomously executing is shown, depicted by 1 & 4. In pink it is seen that the robot encounters an unforeseen obstacle, 2, as a novel instance on the base task. The robot is shown, in 3, by a human demonstrator how to account for the novel instance, after which it ideally learns how to do so autonomously. After the adaptation process the robot should be able to perform the base task and account for the novel instance if it occurs, executing 1 → 3 → 4 in sequence.

1.3 Robot Learning and Task Execution

In order for robots to autonomously assist in performing ADL behaviours, the executed robot motions have to guarantee the safety of human bystanders and the addition of collision avoidance either with self, human individuals and the surrounding world should be facilitated. Similarly, safety perception of human end-users towards the robot has to be upheld. This human-robot interaction and awareness should contribute to the overall acceptance and effectiveness of the use of autonomous robots in daily domestic and healthcare life.

Conventional motion planners, although capable of solving and executing complex motion trajectories for high DOF robots, require the necessity of being manually programmed by expertly skilled individuals. Machine Learning (ML) enables the capability of teaching certain behaviours to an agent autonomously without supervision of a human. From Argall et al. (2009) and Ravichandar et al. (2020) we find that Learning from Demonstration (LfD) forms a paradigm within ML where, if the necessary underlying framework exists, a robot can be taught to perform tasks and behaviours ‘simply’ learnt from a physical human demonstration.

1.3.1 Human Demonstration Methods

Several forms of human demonstration methods exist, some of which include Kinesthetic Teaching (KT), Shadowing, Natural Language Processing (NLP) and Teleoperation. KT requires the human to physically manipulate a robot and its end-effectors as a means of demonstration (Suay et al., 2012)(Calinon, 2018)(Rana et al., 2017). Shadowing, or passive observation, is a method that utilizes visual or sensory recordings of human motions that are in turn transformed back to robot performable actions (Vogt et al., 2017)(Welschehold et al., 2017)(Lafleche et al., 2019)(Zhang et al., 2018). Teleoperation makes use of remote controllers as a method of input, referred to as a *master*, of which the motions are sent to a *slave* that performs the demonstrated motion as accurately as possible (Hokayem and Spong, 2006). These remote controllers can range from videogame controllers, data gloves and joysticks to haptic controllers such as the phantom Omni (Kukliński et al., 2014)(Xiao et al., 2020). Teleoperation is used in other remote control approaches including supervisory control, haptic-shared control and man-machine interaction (Ewerton et al., 2019). Another, but slightly more unconventional, form of human-robot demonstrational methods is NLP. NLP is a broad field that concerns itself with the understanding and manipulation of natural human language by computers (Grosz, 2005). What sets NLP apart from the other previously mentioned methods is that it can only be considered as a feasible method of demonstration if a pre-existing motion library and/or another demonstration method is used in conjunction with NLP in order to learn the necessary low-level motions or robot motor commands (Howard et al., 2014)(Paxton et al., 2019)(Wake et al., 2020)(Song and Kautz, 2012)(Hristov et al., 2018).

Demonstration Method	Ease of demonstration	High DOF	Ease of mapping	Stand-alone	Long Distance Demonstration
Recording Human Motions	✓	✓	✗	✓	✗
Kinesthetic Teaching	✓	✗	✓	✓	✗
Teleoperation	✗	✓	✓	✓	✓
Natural Language Processing	✓/✗	✓/✗	✓/✗	✗	✓/✗

Table 1.1: Characteristics of LfD demonstration methods. Adapted from Ravichandar et al. (2020) and modified.

As can be seen from Table 1.1 it can be concluded that out of the four discussed demonstration methods, teleoperation proves to be the best when working within close proximity to human individuals with a remote operator. NLP, although not standalone, can be regarded as an additional method of demonstration that can be used to supplement for instance a LfD framework using teleoperation. The implementation of the proposed framework, discussed in Section 3.3, utilizes so-called tele-selection for choosing robot skills to construct robot behaviours.

1.3.2 Behaviour Learning and Task Execution

Within LfD a couple of overarching fields or methods for task or behaviour learning and execution exist, which are either low-level learning of Movement Primitives (MPs) with the other being High-Level Task Planning or Symbolic Learning and Reasoning. Low-level learning of MPs focuses on learning and execution of low-level control of robot actuators, whilst high-level task planning concerns itself with looking at a higher level representation of a given robot task and how to correctly sequence and execute behaviours in order to successfully fulfill the initial goal requirements.

1.4 Research Gap, Prior Recommended Future Work & Proposed Project Goal

The following research gap has been identified:

"There has not been any research conducted into a method for simultaneous learning of high-level task plans represented by Behaviour Trees and low-level actions in order to teach a (healthcare) robot to perform autonomous manipulation tasks learnt via demonstration using teleoperation."

This thesis project focuses on closing this research gap by means of the following recommendation:

"Research is conducted into creating an unifying framework joining high-level task plans and low-level robot motions using Behaviour Trees demonstrated via incremental adaptation through human input in order to create safe, modular and reusable behaviours for solving autonomous manipulation tasks found within healthcare environments."

This thesis project mainly focuses on the research towards, as well as the actual implementation of such a framework of which the main research question reads:

"How do we create a framework for the initial construction and continuous incremental adaptation of simple to complex robot behaviours for performing tasks associates with Activities of Daily Living autonomously using a (healthcare) robotic agent?"

1.5 Main Contribution

The main contributions of the proposed thesis project is twofold, of which the first contribution is given as:

To realize a combinatory framework wherein both low-level actions and high-level Behaviour Trees are created and incrementally adapted through human input in order to solve manipulation tasks for mobile robots (deployed in healthcare environments).

The second contribution expands upon the first contribution as follows:

To realize an adaptable framework capable of solving novel instances of prior learnt behaviours by locally expanding on prior constructed or existing Behaviour Trees through human input.

To summarize, this thesis main contribution comes in the form of a framework that facilitates the initial creation and continuous incremental adaptation of simple to complex robot behaviours in the form of Behaviour Trees constructed via human input, which are autonomously executable.

Chapter 2 gives an overview of the necessary background for understanding the concepts and methods as discussed in this thesis report. The inner workings of and methods behind the proposed framework are discussed in Chapter 3. Methods for the practical implementation of such a framework are mentioned. A practical implementation of said methods is discussed as well. Proof-of-concept is given in the form of an experimental validation done in simulation with real human participants which were required to create a BT to tackle a pick-and-place task and to adapt and expand thereupon to facilitate a novel instance on the base behaviour. The experimental validation and the proof-of-work is discussed in Chapter 4. An overall conclusion for this thesis project report is given in Chapter 5 in the form of a summary, recommendations for future work and some concluding remarks. Lastly, the appendices are found in Chapter 6.

Chapter 2

Background

This chapter discusses the most important background facets on which this thesis project has been built. The discussed background forms the backbone of this thesis report.

2.1 Learning from Demonstration

Conventional motion planners are used widely throughout the robotics industry, as these are able to solve and generate complex motion trajectories for high DOF robots (Latombe, 2012). These robots, however, are mostly used in industries where they are placed far out of the physical reach of humans. The main difference between industry and care robots, is their proximity to humans when executing tasks (Ravichandar et al., 2020)(Hosseini and Goher, 2017). It is expected of care robots to safely perform tasks, within environments not optimized for said robots, in the close vicinity of humans. Safety can be referred to as not colliding with the world or with oneself, or circumvention of imposing the feeling of uneasiness, danger or unpredictability during operation of the robot. A downside of conventional motion planners is that they generate efficient, but to the human non-predictable, movements. These type of planners also require a certain degree of programming knowledge and skills in order to be constructed, deployed and maintained. Learning from Demonstration (LfD) is a form of supervisory Machine Learning (ML) wherein an agent can be taught to perform tasks by a teacher without programming knowledge. LfD sets itself apart from other ML methods, as the latter still do require a certain level of knowledge when working with said methods and the former does not (Argall et al., 2009)(Ravichandar et al., 2020).

2.1.1 Solving Learning from Demonstration Tasks

For LfD we can identify two main overarching approaches for solving robotic tasks, which are low- and high-level LfD methods. Low-level learning concerns itself mostly with low-level actuator control in the form of Movement Primitives (MPs), while high-level learning concerns itself with the more overarching task structure and the sequences of motions/behaviours performed for reaching such a task goal.

Movement Primitives (MPs) are a mathematical approach for representing, learning and executing low-level actuator commands and robot movements. Several forms of MPs exist, such as Dynamic Movement Primitives (DMPs), which are the original implementation of MPs by Ijspeert et al. (2002), and Probabilistic Movement Primitives (PMPs) which are a probabilistic variation on DMPs as proposed by Ijspeert et al. (2013). From French et al. (2019) we find that Motion Primitives (MPs) are not sufficient to be used as the sole method for solving manipulation tasks. It is required that MPs are segmented and that the transitions are learned between said MPs in order to sequence a motion solution in order to solve the task. This segmentation in turn allows for reuse to sequence solutions to manipulation problems requiring similar actions as those segmented prior. Thus, reusability as well as modularity are desired when solving mobile robot manipulation tasks. All of these abilities are required to successfully solve high-level LfD approaches. Therefore, it can be easily argued that in

order to solve complex manipulation tasks via LfD, a combination of both low- and high-level learning should be applied.

Several methods within both fields of low- and high-level learning can be identified. It can be argued that for a combinatory framework for learning and updating reusable high-level task plans, the best approach would be to utilize Behaviour Trees (BTs) for high-level task planning whilst using either PMPs or Riemannian Motion Policies (RMPs) for the purpose of low-level actuator control (Paraschos et al., 2013)(Ratliff et al., 2018).

2.2 High Level Task Planning using Behaviour Trees

In this section, we discuss high level task plan representations for robotic behaviours. We discuss high level task planning using Behaviour Trees (BTs) in particular. Section 2.3 discusses Decision Trees (DTs) in short. DTs are discussed due to their shared similarities with behaviour trees. Section 2.4 discusses how Behaviour Trees generalize other behavioural programming ideas such as Decision Trees. Section 2.5 discusses the formulation of BTs in broad detail. Section 2.6 discusses some methods for learning and adapting Behaviour Trees. Finally a concluding section to the overall chapter is given in Section 2.7.

Note that this chapter mainly focuses on using BTs for creating high-level task plans for representing complex robot behaviours. Other high-level task planning methods are not discussed.

2.3 Decision Trees

Decision Trees (DTs) are a popular algorithm used for Machine Learning (ML) purposes which belong to the class of supervised learning algorithms suitable for both classification and regression purposes.

DTs are directed trees that represent a set of nested *if-then* statements which are used to derive conclusions or actions given the (dis)satisfaction of an associated set of preconditions (Sammut et al., 2003). A directed tree is a graph structure in which between the nodes, vertices or edges are drawn that indicated the flow or transfer of control or presence from one state to the other in the form of a tree. Different denotations of DT nodes exist. The denotation as used in the thesis report is that DTs consist out of several node types including a *root* node, *branches* and *leaf* nodes. The root is the first node found within the tree. From the root node we propagate through the tree from top-to-bottom and from left-to-right. Each branch denotes the outcome of a certain condition as stated in the decision nodes, whereas the leaf nodes execute the appropriate action in response to the outcome of said decision node. Thus, the leaf nodes equal atomic actions or the smallest actions taken in a tree. An example of a DT skeleton is seen in Figure 2.1.

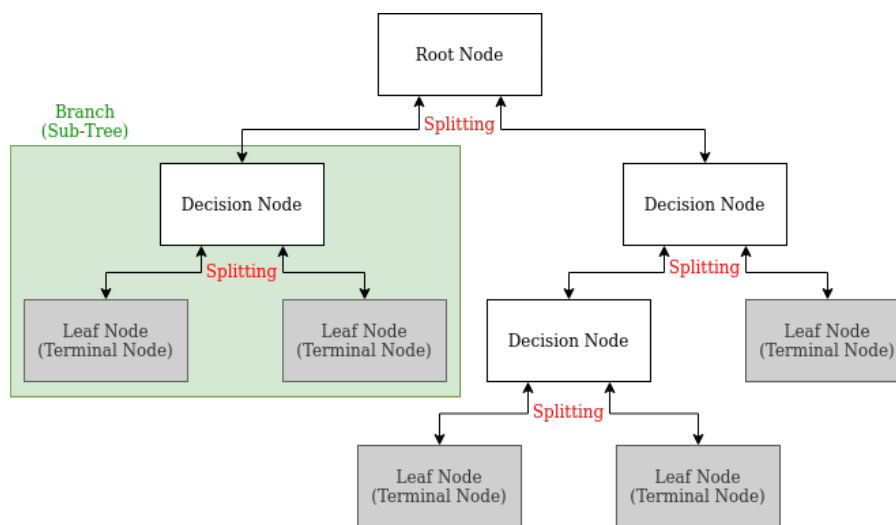


Figure 2.1: Example DT denoting the used terminology.

A more practical example of DTs representing a robotic pick task is given in Figure 2.2. The example pictures a robotic task where picking up a piece of clothing from a laundry bin is the goal to be achieved. Starting from the root node of the tree, if the robot is close to the laundry bin we move to the first child T_1 which is a decision node that determines whether there is actually any laundry in the laundry bin. If there is laundry, the robot performs the action to pick up the laundry and thus fulfills the overall goal of the DT. In the case that the robot is not close to the laundry bin, we go into the second child of the root node T_2 , which correlates to an action to move closer to the laundry bin.

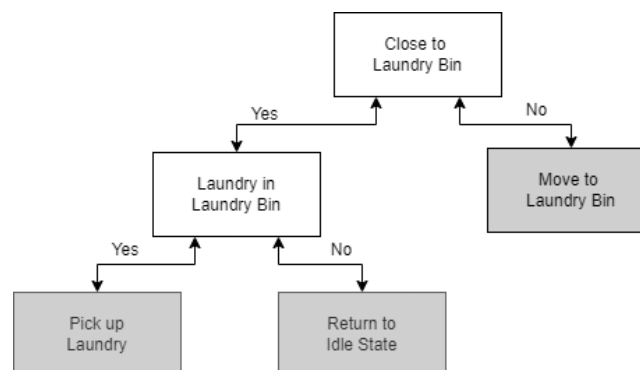


Figure 2.2: Example DT portraying a task for picking up laundry from a laundry bin.

It is simple to see, that if the end-goal of the robot is to pick up laundry from the bin, we need to go through the DT again after moving to the laundry bin, which should satisfy the root condition of the robot being close to the bin. In the scenario that the robot is close to the bin, but no laundry is found within the laundry bin, the robot returns to an idle state. In this scenario with this given tree, the robot can not possibly achieve its goal unless some laundry is to suddenly appear in the laundry bin. Thus, it can be **noted** that unless the DT is to be expanded with some conditions and actions that can fill the bin up with laundry (for instance by emptying a washing machine or collecting dirty clothes around the room), the robot can not possibly ever hope to complete the given task and would be stuck in a never-ending loop trying to pick up non-existing laundry.

An alternative name to Decision Trees is CART (Classification and Regression Trees). CART refers to ML methods used for data-driven predictive modelling (Loh, 2011). Learning DTs in order to convert said DTs to BTs using their shared equivalence has been done by French et al. (2019), as discussed in Section 2.4.

Advantages

Some of the advantages of DTs can be listed as:

- **Modular:** One of the major advantages of DTs is their modular nature. With modularity we reference the fact that the tree structure of a DT allows for the independent development of each branch found within the tree, which in turn can be sequenced or placed wherever in the tree it needs to be.
- **Reusable:** Independent development of subbranches also leads to reusability of these branches in other DT implementations and that they can be applied to a plethora of different (robotic) systems. For example the subbranch for *picking an object* can be used in a multitude of different robotic pick-and-place tasks.
- **Hierarchical:** The tree structure and the manner of executing a DT (*top-to-bottom* and *left-to-right*) all contribute to predicates, a question which can either be answered with yes or no, being evaluated in a hierarchical fashion.
- **Intuitive:** the simple but straightforward design of a DT makes it very easy to understand. DTs are thus easily interpretable by the human and allows for good insight in the decision making process of a DT.

Disadvantages

Some of the disadvantages of DTs are:

- **No information flow:** No information flows out of the nodes of a DT. This proves to be difficult for error handling, as no information is present as to what might have caused errors or can be done to tackle errors found during execution of a DT. The lack of information flow can also result in the disadvantage as mentioned in the next item.
- **Risk of getting stuck in a loop:** In the case that the desired outcome of a predicate in order to satisfy the overall goal of the DT can not be attained, an endless loop might occur wherein the goal will never be reached. This phenomenon was described in the prior Section 2.3 in which an example was portrayed using the DT in Figure 2.2 and the scenario that no laundry is present in the laundry bin. A possible solution to avoid getting stuck in a loop is the addition of a global variable that could warn that the DT is being executed in a continuous loop. But as mentioned prior, due to the absence of information flow out of nodes, this can not be implemented in DTs. An alternative is using BTs over DTs, which will be discussed in the upcoming Section 2.5.

*Most of the mentioned (dis)advantages of DTs were retrieved from Colledanchise and Ögren (2020). However, due **note** that the detail given to the (dis)advantages in Colledanchise and Ögren (2020) is only marginal and has been explained in broader context in this section.*

2.4 How Behaviour Trees Generalize Other Behavioural Programming Ideas

From Colledanchise and Ögren (2017) and Colledanchise and Ögren (2020) we find that BTs generalize other behavioural programming ideas including most importantly Finite State Machines and Decision Trees. This section discusses the generalization of these programming ideas in short. The generalization of these programming ideas by BTs are of importance, as it enforces the choice of using BTs over said other methods, as most benefits are carried over from those programming ideas.

A Finite State Machine (FSM) is a method for representing a system by a finite number of states. States can be nested FSMs themselves and are then referred to as Nested or Hierarchical FSMs (HFSMs). From Marzinotto et al. (2014) we find that all the workings of a HFSM can be represented by a BT. Colledanchise and Ögren (2020) states that HFSMs are the closest architecture to BTs concerning working principles and the use thereof. The generalization of Finite State Machines (FSMs) by BTs is emphasized with the direct comparison between BTs and Hierarchical FSMs, as found in Section 2.5.1.

Decision Trees (DTs), as discussed prior in Section 2.3, are graph structures that represent nested *if-else* statements used for decision making processes. From Colledanchise and Ögren (2017) we find the equivalence of DTs and BTs, which allows for the conversion of one policy representation method to the other. Mainly the conversion from DTs to BTs are of interest. This is further clarified when comparing the advantages of using BTs in comparison to DTs, discussed in Sections 2.3 and 2.5.1 respectively. Most importantly, we thus find that since BTs generalize DTs, we can rewrite every form of DT into an equivalent BT. It has to be noted that as DT designs inherently do not have feedback in the form of a return status for the given actions found within said tree, it has to be assumed that all *leaves* of a DT return `Running` (Ögren, 2020a). An example of a DT converted to a BT is depicted in Figure 2.3.

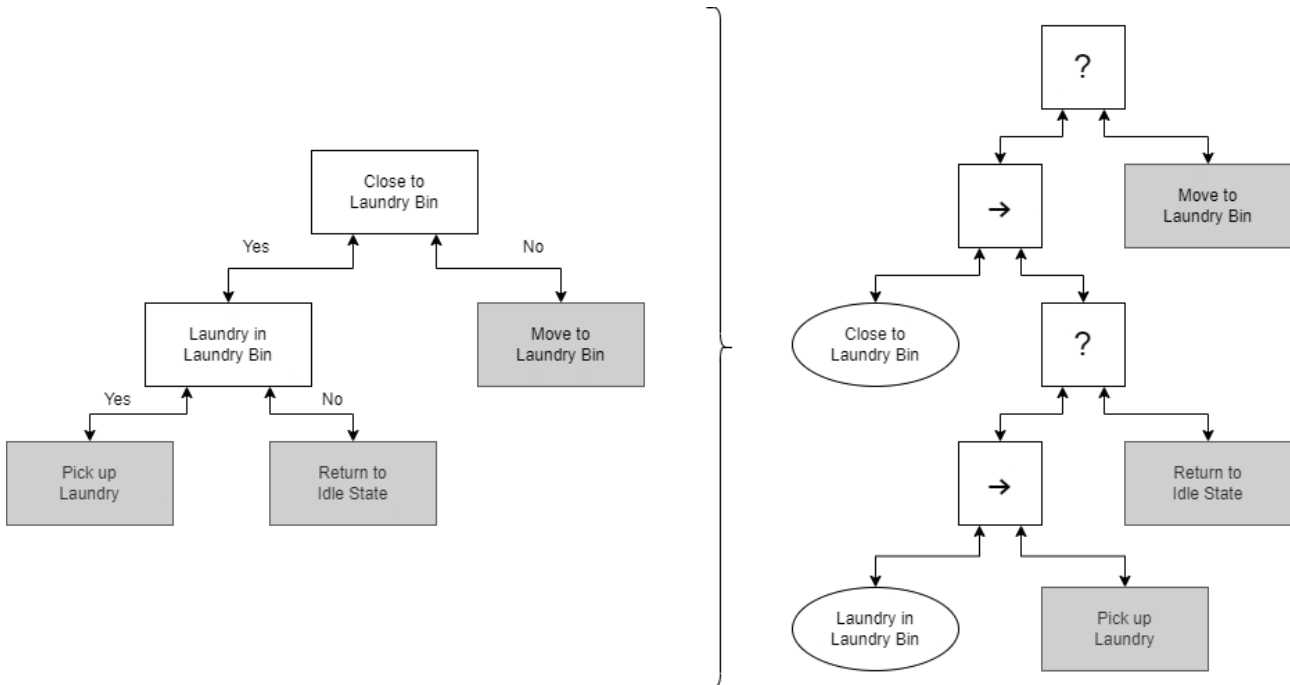


Figure 2.3: Example conversion of the DT for picking up laundry as previously given in figure 2.2 into a BT as portrayed from **left** to **right**. This example is based on an example made by Ögren (2020a).

The conversion of DTs to BTs is of interest as it helps in understanding the structuring and execution flow of BTs. For Figure 2.3 it can be noted that the issue regarding the *return to idle state* still persists, resulting in an endless loop in the case that no laundry was to appear in the laundry bin. A method for resolving this, is adapting or expanding the DT or BT in particular to account for this problem. Adapting BTs is extensively discussed in this report in Chapter 3.

2.5 Behaviour Trees

Note that some of the information as found within this section is based on lectures by Petter Ögren given at KTH Royal Institute of Technology Stockholm retrieved from YouTube which are in turn based on his book co-written with Michele Colledanchise (Ögren, 2019a)(Ögren, 2019b)(Ögren, 2020a)(Ögren, 2020b)(Colledanchise and Ögren, 2020).

2.5.1 Overall Concept of Behaviour Trees

Behaviour Trees (BTs) are a high-level representation of behaviours or task-plans that are able to represent complex robot tasks. Mostly used within the videogame industry since the launch of the popular videogame Halo 2 in 2005, all the while making Hierarchical Finite State Machines (HFSMs) obsolete in said industry, some implementations/tools that aids in the simple creation of complex high level behaviours via BTs exist for that particular industry (Isla, 2008)(Isla, 2005). An example of a tool for creating BTs is the built in BT creator in the videogame physics engine Unreal Engine 4 (Lee, 2016). BTs are equivalent to Decision Trees (DTs) and generalize other behavioural programming ideas, as discussed in Section 2.4. The closest other behavioural programming architecture to BTs, aside from sharing a lot of equivalence with DTs, are HFSMs.

Similarly to DTs, BTs are structured using a tree structure where the execution of a BT works using *ticks* which are sent from the tree down to the root, which in turn triggers a return status which can either be **Success**, **Running** or **Failure** which are all sent back up the tree. The execution of a BT is therefore *tick*-driven in comparison to being *event*-driven such as HFSMs. A BT consist out of small building blocks, or nodes, which all form to represent complex behaviours upon expansion and particular sequencing of said nodes.

We can state that BTs are a hierarchical method encapsulated in a modular and reactive architecture for switching between low-level tasks, forming complex high-level behaviours.

Node types

BTs are created from small building blocks or nodes. Nodes are either referred to as a *parent* or *child*. The root node is the first node in a BT positioned at the top of the BT and thus has no parent of its own. The BT propagates from top-to-bottom with one or more children originating from the root node. These children are in turn referred to as parents of their subsequent children when propagating further down the BT and henceforth. There exist five standard node types, which are the sequence, fallback, parallel, action and condition nodes. An additional sixth node, the decorator, can be tailored custom to the users' need. The five standard node types found in BTs described by Iovino (2020) are found in Table 2.1 alongside the decorator node:

Node type	Symbol	Succeeds	Fails	Running
Sequence	→	If all children succeed	If one child fails	If one child returns running
Fallback	?	If one child succeeds	If all children fail	If one child returns running
Parallel	⇒	If $\geq M$ children succeed	If $> N^*M$ children fail	else
Action	shaded box	Upon completion	When impossible to complete	During completion
Condition	white oval	If true	If false	Never
Decorator	◇	Custom	Custom	Custom

Table 2.1: BT node types. Adapted from Iovino et al. (2020). Please refer to Figure 2.3 for an example of the node type usage as depicted in this table.

Note that a different categorization of the nodes is presented in Colledanchise and Ögren (2020) which states that the classical formulation of the nodes are subdivided into two different categories being the *control flow* and the *execution* nodes. The control flow nodes include the sequence, fallback, parallel and decorator nodes. Whereas the execution nodes consist of the action and condition nodes. The definition and inner workings of each of the node types are equal for both Iovino et al. (2020) and Colledanchise and Ögren (2020). This report will refer to sequences and fallbacks as the control flow nodes, whereas actions will be either referred to as is or as leafs. Finally conditions are plainly referred to as conditions.

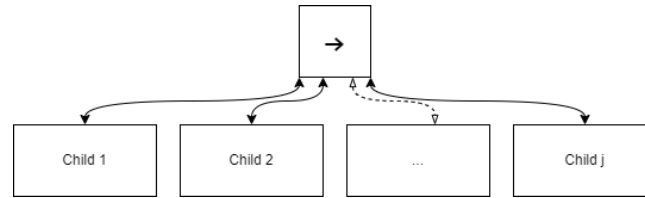


Figure 2.4: Basic structure of a sequence node with an N amount of children T_1, T_2, \dots, T_j . The sequence succeeds if all N children return **Success**.

A sequence node consist of two or more children (T_1, T_2, \dots, T_j), ticking them from left to right. Once the first child T_1 returns **Success** the second child T_2 positioned directly next to T_1 is ticked. The sequence returns **Success** only if all children T_{ij} return **Success**. Once a child returns **Failure** the execution of the rest of the sequence is preempted and a return status of **Failure** is given. The pseudocode for the sequence algorithm is found in Figure 2.8 as Algorithm 1. An example of a sequence is given in Figure 2.4.

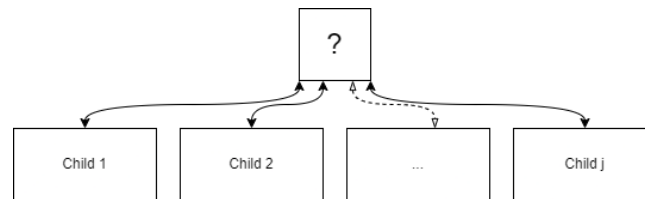


Figure 2.5: Basic structure of a parallel node with an N amount of children T_1, T_2, \dots, T_j . The fallback succeeds if one child returns **Success**.

A fallback node, similar to a sequence, consist of two or more children (T_1, T_2, \dots, T_j), ticking them from left to right. The fallback returns **Success** if one of its children T_{ij} returns **Success**. For example T_2 is only ticked if T_1 returns **Failure**. Consider a fallback that consists of three children T_1, T_2 and T_3 . If T_1 returns **Failure**, T_2 is ticked. If T_2 returns **Success**, then the entire tree succeeds. Thus, in this scenario T_3 is never ticked nor executed. The pseudocode for the fallback algorithm is found in Figure 2.8 as Algorithm 2. An example of a fallback is given in Figure 2.5.

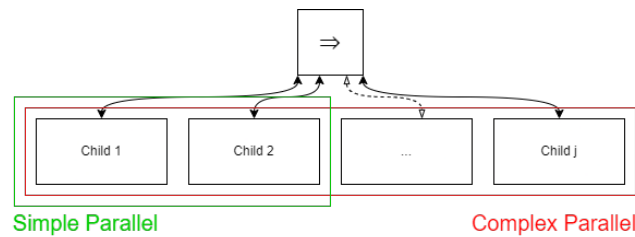


Figure 2.6: Basic structures for both a simple parallel node that has two children T_1, T_2 and a complex parallel node with a N amount of children T_1, T_2, \dots, T_j . Both types of parallel node succeed if the user given threshold for amount of children that have to return **Success** $M \leq N$ is not violated.

A parallel node consist of at least two children (T_1, T_2, \dots, T_j), ticking all children T_{ij} at the same time. Most often parallel nodes with a maximum of two children are used, also referred to as a simple parallel node in the Unreal Engine implementation of BTs (Batelsson, 2016). Parallel nodes with more than 2 children are coined as complex parallel nodes. A parallel node succeeds if M children return **Success**, of which $M \leq N$ is a user defined threshold over the total number of children N . The parallel returns **Failure** if $N - M + 1$ children fail. The pseudocode for the fallback algorithm is found in Figure 2.8 as Algorithm 3. An example of a parallel node is given in Figure 2.6.

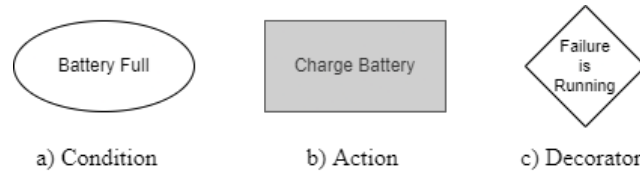


Figure 2.7: **a)** A condition. This example condition contains the predicate *Battery Full*, which can be either answered with *yes* or *no*. **b)** An action. This action executes a behaviour for an agent to charge its battery. **c)** A decorator. The example decorator in this case is a *FailureIsRunning* node, implying that the return status of the decorator’s child is set to *Running* if it were to output *Failure*.

An action node or leaf executes a command, function or script upon receiving a tick from up the tree. Upon receiving a tick, an action node immediately returns *Running* until the action execution has finished resulting in either *Success* or *Failure*. The return status of the action node equals *Success* if the associated action has been executed successfully, *Running* if the action is still in the processing of being executed and *Failure* if it has failed to do so properly. An example of an action is given in Figure 2.7.a.

A condition node is a node that checks a certain proposition or predicate. The condition node can return either *Success* or *Failure*, equivalent to the Boolean values of *True* or *False*, and is dependent on associated control and execution nodes that influence the outcome of said specific predicate. An example of a condition is given in Figure 2.7.b.

A decorator node is a custom node that can perform operations based on the return status of a single child or alter said node’s return status. A few examples of decorator nodes as found in *py_trees*, a BT library for *Python*, are those with a very specific functionality and those that belong to the *X is Y* family. The former include decorators that implement a oneshot pattern where a child is only ticked until completion and a decorator that returns a node’s status to the Blackboard. The latter form of decorators include decorators that switch a return status *X* to another return status *Y* such as the decorator *SuccessIsFailure*, wherein a *Success* return statement is altered to *Failure*. An example of a decorator is given in Figure 2.7.c.

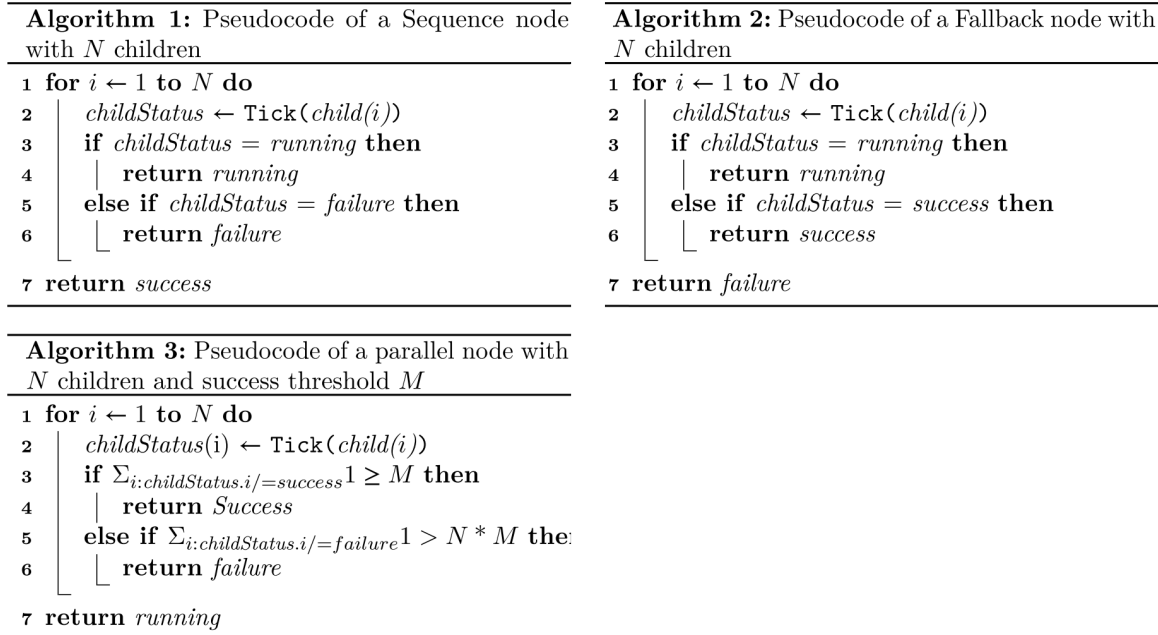


Figure 2.8: Pseudocode for the *sequence*, *fallback* and *parallel* nodes. Retrieved from Iovino et al. (2020).

An example BT using all of the node types as explained in this section is given in Figure 2.9. Starting from the root of the tree we find a parallel node that has two children, T_1 being a branch for emptying a laundry bin and T_2 being a branch to charge the agent’s battery. Both branches or subtrees are ticked at the same time.

Going into the *Empty Laundry Bin Branch* we find a fallback F_1 with two children, being a condition C_1 and a sequence S_1 that in turn has two children, actions A_1 and A_2 . Going into the *Charge Battery* branch we find a similar structure, with the exception of the fallback F_2 's second child being a decorator D_1 with an action A_3 as only child. Given the example scenario where we define for the parallel node that $M = N$, we tick both branches simultaneously. Finding out that the laundry bin is not empty, we fall back into S_1 . Successfully performing A_1 and A_2 in consecutive order by opening the laundry bin, picking up the laundry and thus emptying the bin, S_1 returns **Success**. The successful return status of S_1 moves up to F_1 , returning **Success**, moving up its return status all the way back up to the root of the BT. At the same given time instance we find that the agent's battery has been nearly depleted, thus failing condition C_2 of F_1 . This results in both D_1 and A_3 being ticked. After some time t the agent fails to continue charging the battery due to a short power outage. D_1 however alters A_3 's return status **Failure** to be **Running**, thus without moving the return status back up the tree A_3 keeps executing until the battery has been fully charged. This leads to the second branch of the BT eventually succeeding as well as the root parallel node ultimately returns **Success**.

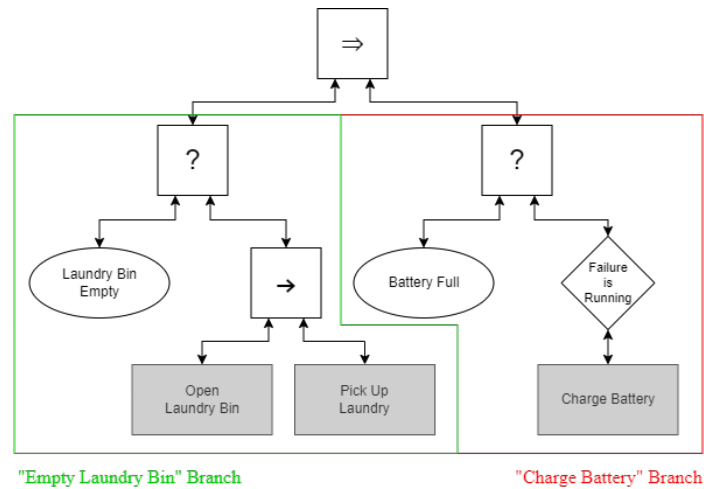


Figure 2.9: BT example using all node types as found in the classical formulation of BTs.

Advantages

As we discussed earlier in Section 2.4 we now know that BTs generalize DTs. This means that due to their similarity most of the advantages of using DTs hold true when using BTs as well (Colledanchise and Ögren, 2017)(Colledanchise and Ögren, 2020). The main advantages of BTs can be listed as follows, including the four main advantages of DTs as discussed previously in Section 2.3 being modularity, reusability, hierarchy and intuitivity:

- **Modular:** Similar to DTs, BTs are modular due to the fact that their structure allows for independent development of (sub-)branches and trees, which can all be interwoven into one high-level framework. Independent development includes the facilitation of different programmers, different programming languages and different styles of programming to work with one another as the execution of leaf nodes is independent from the logic of ticking a BT.
- **Reusable:** The modularity of BTs and independence between leaf node execution and the BT structure allows for a high level of reusability. Independent development of behaviours enables the reuse of trees in a plethora of different projects as BTs are structured in such a way that different coding styles do not affect the overall workings of a BT. Reusability and modularity both improve development time and ease the design and development of high-level task plans.
- **Easy to design and develop:** Mainly thanking the easy structure of BTs and the reusability of trees or branches, new BTs can easily be constructed e.g., with the additional use of a behaviour/skill or BT library.

- **Hierarchical:** A BT is ticked from *top-to-bottom* and from *left-to-right*. Branches and nodes with higher priority are placed more to the top-left side of a tree, while branches and nodes with less priority (that mostly serve as alternative measures to attain a certain task goal) are placed more to the lower-right side of the tree. A control architecture is deemed hierarchical if at least multiple decision making levels exist or the existence thereof is facilitated. Control hierarchy allows for prioritizing actions within and the expansion of a given behaviour.
- **Reactive:** Similarly to DTs we find that BTs are directed trees. In comparison to directed acyclic tree structures such as HFSMs that only facilitate one-way control transfer, directed trees can have two-way control transfer that flows back and forth between the vertices of two nodes. Two-way control transfer in between nodes allows for improved *reactivity*. Reactivity refers to the ability to react to changes in the world environment in a quick and efficient manner. Reactivity is desired in partly-unknown environments, as changes in the world's variables might lead to unsafe or undesirable situations for an agent to be in. Actions within a BT can be preempted during execution depending on the world variables as saved in a *Blackboard*, a given action node's return status and the corresponding pre-conditions.
- **Intuitive:** Intuitiveness or human interpretability refers to the ease of understanding a given concept. The simple BT structure and modularity that stems forth from upholding said structure throughout the expansion of such a tree, allows humans to easily read and understand such a type of high-level task plan. The degree of human readability contributes to the ability of being able to understand, develop and debug such a high-level task plan, especially when human designed.
- **Expressive:** Expressiveness or the power of expression is regarded as the ability to implement and represent a plethora of ideas by means of programming. The level of expressiveness verdicts the scale of ideas that can possibly be implemented using that certain method. BTs are deemed to be expressive as they generalize other behavioural programming methods such as HFSMs, discussed prior in Section 2.4.
- **Suitable for analysis:** Robustness, efficiency, reliability and composability are all factors that need to be taken into account when designing and developing robotic systems where safety is considered crucial.
- **Suitable for automatic synthesis and learning:** BTs due to their structural consistency and construction alongside two-way control transfer are suitable for automatic synthesis using machine learning techniques as initial synthesis and expansion is straightforward. This is opposed to e.g., HFSMs that quickly become too complex for an algorithm to synthesize and/or expand due to them having many different one-way transitions between states that need to be specified. The HFSM structure increases the computational time and efficiency during automatic synthesizing and expansion of said model.

A quick comparison can be made between BTs and Hierarchical Finite State Machines (HFSMs) which have been the industry standard for programming high-level task plans for the past decades. A Finite State Machine (FSM) is a model representing a system as a finite number of states that utilizes a one-way control transfer to travel from one state to another. Within each state another state can be nested. These types of FSMs are called Hierarchical Finite State Machines (HFSMs) (Sullivan et al., 2010)(Colledanchise and Ögren, 2017). The past two decades starting with the release of the all-time popular video games Halo 2 and 3, BTs have fully come to replace HFSMs in the video game industry (Isla, 2005)(Isla, 2008). The use of BTs, albeit being introduced by Iske and Rückert (2001), is far less mature in the robotics field and has only been gaining traction over the past few years (Bagnell et al., 2012)(Colledanchise and Ögren, 2014)(Colledanchise and Ögren, 2017)(Axelsson and Skantze, 2019)(Iovino et al., 2020). The main reasons for choosing BTs over HFSMs can be listed in short as:

- HFSMs use one-way control transfer, whereas BTs have two-way control transfer. The two-way control transfer is similar to a *function call*. One-way control transfer can lead to many state transitions needing to be explicitly specified in between the states of a HFSM. This makes using HFSMs for behavioural expansion less suitable than BTs.

Especially when considering automatic synthesis or incremental adaptation, HFSMs tend to become increasingly more complex upon expansion and are prone to error when expanded due to the sheer number of required transitions in between states.

- The increasing complexity of expanded HFSMs can result in problems regarding debugging, updating due to the increasing clutter of the states and their related state transitions. For large task plans that are learnt, BTs tend to be more computational efficient as they can be kept minimal in comparison.
- Reactivity in the form of a **Running** return statement improves the reactivity of BTs in comparison to HFSMs that have to rely on concurrent state machines.
- The addition of a fallback control flow node in BTs provide alternative approaches to achieving the same given task goal, which in turn can improve task completion time as well increase computational efficiency.
- The main difference between BTs and HFSMs is that in HFSMs the hierarchy has to be defined explicitly, whereas every module in a BT can be regarded as independent from one another and the execution of such a module is considered similar to the execution of an atomic action, or the smallest action possible.

Disadvantages

In Section 2.3 we discussed some of the disadvantages of using DTs, being the lack of information flow out of nodes and the possibility of getting stuck into a loop. This raises the question whether these disadvantages hold true for BTs as well due to their equivalence to one another. This section shortly discusses the disadvantages of BTs, which can be listed as:

- **Maturity:** The use of BTs is less mature than HFSMs, both of which are direct competitors to one another. This simply means that there, as of current, is more research regarding HFSMs as well as more tools available for actual implementation thereof. BTs can still be viewed as a recent upcoming field of research, with lots of opportunities and improvements upon its definition, approach and implementation. This thesis hopes to build forth on the continuing advancement of BTs for use in the robotics industry.
- **Computational cost:** Albeit computational efficiency was deemed as one of the advantages of BTs in the previous section, computational cost of BTs can run high in some cases. Considering a BT is ticked with a certain frequency T_{freq} , the tree is executed several times over all the while continuously checking the pre- and postconditions while actions are left **Running**. This continuous traversal of the tree, although improving *reactivity*, can result in a higher computational cost compared to running a singular action command for an agent. In the case of automatic synthesis of a BT there exist the disadvantage of **uncontrolled inefficient expansion** of the tree. In other words, we can say the the tree *blows up* out of proportion upon expansion. This leads to a large, complex and inefficient tree. A solution for this, however, is using logic minimization to minimize such a tree.
- **Tick driven:** BTs are executed via ticks that propagate *top-to-bottom* and *left-to-right* starting from the root node. The BT sends signals that have to traverse down the entirety of the tree every time-step $t + 1$, meaning that the execution of a BT is not event driven. This leads to the problem that more conditions might need to be checked when using BTs in comparison to HFSMs that only need to check one or several states at a give time instance t .
- **Non-explicit state representation:** In comparison to HFSMs there exists no explicit state representation for BTs. Thus, instead of checking the state, conditions are checked. Condition checking however, as stated previously, can be expensive as we do not have an overview of the current state and the actions are purely based on checking the conditions (Pezzato, 2021).

2.6 Methods for Learning and Adapting Behaviour Trees

This section provides a short introduction of methods for learning and adapting behaviour trees. Both, methods for automatic synthesis of BTs using an existing skill or behaviour library as well as learning and updating BTs from scratch using Learning from Demonstration (LfD) are discussed. LfD was highlighted prior in Section 2.1.

Two approaches for automatic synthesis of BTs while using a pre-existing skill or behaviour library are either implementations that utilize automated planning methods alongside BTs or Machine Learning (ML) approaches which represent a learnt policy using BTs. Both methods are shortly discussed in Sections 2.6.2 and 2.6.4 respectively. Automated planning refers to replanning a BT from scratch on a regular basis using (conventional) planning algorithms. It focuses on two main challenges referred to as *hierarchically organized deliberation* and *continual planning and deliberation*. Loosely explained as, continuously monitoring and online updating by refinement or partly removal of the agent's task plan in a hierarchical manner (Ghallab et al., 2014).

Three approaches for creating and/or expanding upon (existing) BTs using ML methods exist, which is either done via Genetic Programming (GP), Reinforcement Learning (RL) or learning DTs and the subsequent conversion thereof to create BTs. All three approaches are discussed in the subsequent Sections 2.6.3, 2.6.4 and 2.6.5.

Most of the mentioned methods utilize a form of backward chaining, which is discussed in Sections 2.6.1 3.2.4.

Note that a larger section on updating BTs can be found in Section 3.3.7 that discusses the actual implementation of backward chained behaviour trees. As mentioned earlier, this section only glosses over the general methods for learning and adapting BTs.

2.6.1 Backward Chained Behaviour Trees

Backward chaining expands on existing BTs by substituting a condition node with a (sub-)BT or module that achieves said condition. This can be done, for instance, in the case that a **Failure** status is returned after checking a precondition that needs to be satisfied in order to execute a specific action, expanding the overarching BT and making that specific behaviour more robust (Ögren, 2019b). Backward chaining can be regarded as solving high-level planning problems by means of working backward from the goal state. By means of adding sub-trees to an existing BT we can create larger reactive tree structures. An example of backward chaining for BTs is depicted in Figure 3.6. It has to be **noted** that actions within a tree should not conflict with one another or else this could result in later actions breaking prior satisfied conditions (Ögren, 2019b).

In short, backward chained (BC) BTs are BTs of which certain behaviours or modules are made more robust by substituting failed conditions or predicates, with a specific branch structure that includes additional preconditions alongside additional actions in order to satisfy the original predicate of the backward chained condition node. Backward chaining is discussed in more detail in Section 3.2.4

2.6.2 Automated Planning & Behaviour Trees

Two approaches for automated planning using BTs exist, being the *Planning and Acting* approach and *planning using A Behaviour Language*.

The Planning and Acting (PA-BT) approach is a combination of a Hybrid-Backward Forward (HBF) algorithm, which is a task planner that can cope with infinite state spaces and BTs (Garrett et al., 2015)(Colledanchise et al., 2019a). This combination allows the framework to take advantage of BTs and plan tasks for infinite state spaces. PA-BT utilizes backward chaining, discussed in Sections 2.6.1 and 3.2.4, to replace conditions with additional branch structures in order to fulfill said conditions.

The Planning using A Behaviour Language (ABL) approach consists of several components including a behaviour library, working memory, the tree itself, the world and sensors to perceive the world. The behaviour library contains pre-existing skills that can be sequenced into a tree given what the sensor perceive in the agent’s world, which are written to the working memory, as seen from Figure 2.10. It has to be noted that ABL is a behavioural programming language on its own and that BTs are used in combination with the language purely as a tool for representing the conditions and actions taken.

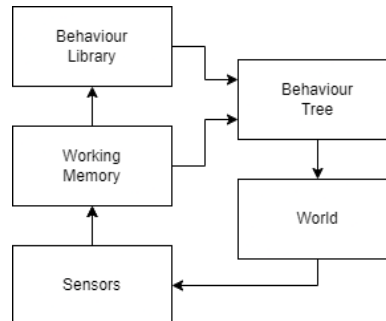


Figure 2.10: ABL approach for programming robotic behaviours, as adapted from Colledanchise and Ögren (2020).

The main difference between PA-BT and ABL is that the former focuses on creating BTs using a planning approach while the latter is a programming language on its own that merely adopt BTs as a representational tool. It has to be **noted** that this approach of continuous replanning for every new execution or attempt to achieve a certain task goal is computationally expensive. The computational cost is therefore a major drawback for using continuous automatic synthesis or re-planning of BTs.

2.6.3 Genetic Programming & Behaviour Trees

Genetic Programming (GP) is an optimization algorithm that uses multiple so-called *generations* to evolve and optimize a specific policy (Rechenberg, 1994). Applied to BTs, the GP-BT approach utilizes three different operations being crossover, mutation and selection. The first operator, crossover, randomly swaps a sub-tree with another at any level within the main tree. The second operator, mutation, replaces a node within a sub-tree with another node of a similar type. The third operator, selection, selects a set of individuals created through the evolution phase that will be used to create the next evolution set of individuals.

In short, GP-BT mixes an evolutionary algorithm to optimize an initially attained policy via a greedy learning algorithm (Colledanchise et al., 2019b). I.e., a large BT is created at first that is minimized and optimized over time.

2.6.4 Reinforcement Learning & Behaviour Trees

Within Reinforcement Learning (RL) a function, coined as the Q -function, is optimized that looks at the estimated future weights of each of the agent’s state-action combination given an agent’s learnt policy $\pi : S \rightarrow A$ that maps from state to action (Colledanchise and Ögren, 2020). In short, the RL algorithm attempts to maximize a reward function or Q -function at any given state, thus converging to an optimal policy given that all state-actions combinations of an agent can be incrementally updated until infinity (Barto and Mahadevan, 2003).

The RL-BT approach uses a manually designed tree as a starting point for gradual expansion via *learning action nodes* (which encapsulates a Q -function with states s , actions A and a reward r) and *learning control flow nodes* (which is a fallback wherein the children’s order is flexible and can be switched with one another based on priority).

2.6.5 Conversion of Learnt Decision Trees to Behaviour Trees

As discussed earlier in Section 2.4, we know that DTs are equivalent to BTs. This equivalence allows for a conversion of every type of DT to an equivalent form of BTs. This has also been portrayed with Figure 2.3 as an example.

French et al. (2019) proposed a method for learning high-level robotic manipulation task plans using LfD. Within his method he uses his so called *BT-Espresso* algorithm to convert a DT to a minimal BT. The DT to BT conversion is done, by creating a decision tree wherein motion trajectory data for a mobile robot is embedded via a `.yaml` file that can be read by a conventional robot controller. This DT is then converted to a BT using a script that is based on the DT and BT equivalence first identified by Colledanchise and Ögren (2017).

2.7 Chapter Discussion

The major difference between the discussed continuous automatic (re-)synthesis of BTs using either planning algorithms or ML, as discussed in Sections 2.6.2, 2.6.3 and 2.6.4 and the creation and incremental expansion (when needed) of BTs, Sections 2.6.1 and 2.6.5, is the difference in computational cost. The first set of methods continuously reiterate and resynthesize a BT given a certain task plan and a prior available skill/behaviour library. The second set of methods focuses on the expansion and/or creation of reusable BTs without the need for continuous re-synthesis, only expanding if and when necessary. Considering that this thesis revolves around the initial construction and further incremental adaptation of *reusable* BTs for robotic manipulation tasks, it can be easily concluded that the second set of methods for constructing and updating BTs is preferred. Both backward chaining and DT to BT conversion allow for the incremental expansion of BTs containing base behaviours to form more complex behaviours over time for solving robotic manipulation tasks. It has to be **noted** that this thesis assumes the presence of a human operator with limited to no background knowledge with respect to programming and/or robotics. The human operator provides aid to a robotic agent if help is needed in completing certain tasks that are unsolvable for the agent on its own. DT to BT conversion, as found from French et al. (2019), poses as a good approach for supervisory learning or learning from demonstration purposes. This is in comparison to the other approaches that e.g., use GP and RL, which can be regarded as unsupervised learning methods. However, after looking at the code repository of the framework as proposed by French et al. (2019) it was opted to not go into the direction of applying ML methods within the in this thesis proposed *Behaviour Tree Update Framework*. The main reasoning for this was the difference in research objective. French attempted to use knowledge of prior taken actions, to infer and learn the possible best next action to reach a certain goal state autonomously, of which the high-level task plan is represented by a BT. The research goal of this thesis report is to conjure a framework for the initial construction and continued incremental adaptation of reusable BTs via human input to create simple and complex behaviours that are associated with performing ADL for use in domestic and healthcare robotic agents, without requiring the human operator to have an extensive programming background. Thus, incremental adaptation via human input using tele-selection is proposed as the main method for creating behavioural trees that enable healthcare robots to execute behaviours that assist humans with performing ADL, as discussed extensively in Chapter 3.

The use of BTs and the expansion thereof while creating and adapting a behaviour specific to an individual service/healthcare robot allows for reactive performance of constructed behaviours during real-time execution, thus upholding safety. Reactivity comes from the ability to run *action* nodes, also called *leafs*, in a separate thread while the BT is ticked and cycled through continuously in the chance that the execution of another action has to be inferred and prioritized over the already running node. In other words, this means that a running action with lower priority can always be subsumed by an action of higher priority if the corresponding conditions are met. As a practical example we take a robot performing a picking task. The picking motion can be terminated and subsumed by an action that avoids possible collision of the robot's end-effector and a world object if need be. Thus, the robot can prioritize safety over task completion via reactivity.

Existing BTs can be re-used as modules or sub-branches in other manipulation or more general task applications that share the need for a similar set of skills. Thus, BT modules are interchangeable, reusable and therefore BTs are considered to be modular. Backward chaining allows for expanding upon different base behaviours by substituting failed condition or action nodes with additional skills encapsulated in a sub-branch. More of which will be discussed in Section 3.2.4.

All in all, BTs provide a clear overarching structure in the form of a high-level task plan, also easily understood by human observers in contrast to the more popularly and commonly used low-level robot motion policies in the form of a set ODEs (Ordinal Differential Equations). These policies make a lot of sense to a robotic agent, but less to an ordinary human albeit being in the possession of robot programming knowledge or not.

Chapter 3

Proposed Method

This chapter proposes a novel framework for creating and incrementally adapting a Behaviour Tree (BT) from scratch for Learning from Demonstration (LfD) purposes in solving complex robotic manipulation tasks associated with Activities of Daily Living (ADL).

Section 3.1 discusses the basic working principles of such a framework. Section 3.2 proposes a BT adaptation pipeline based on these working principles in the form of the *Behaviour Tree Update Framework* (BTUF). Section 3.3 discusses a practical implementation of the proposed BTUF pipeline. The experimental validation, overall system acceptance and proof-of-concept of BTUF is discussed in the next chapter, Chapter 4.

3.1 Basic Principles

A research gap for creating high-level task plans using BTs with minimal programming knowledge for LfD purposes e.g., by means of a simple-to-use framework is identified.

This thesis proposes a simple reproducible framework for creating and incrementally adapting a BT for programming a robot how to perform complex tasks associated with Activities of Daily Living (ADL) without requiring the user to have much programming language. The main functions of the framework as proposed in this thesis are summed up as:

- **Utilizing existing robot skills of which the manner or language of coding does not matter for the operational use thereof within the framework**
- **Adding an arbitrary amount of robot skills that are sequenced correctly in order to perform complex tasks**
- **Adding robot and world state checks** that influence a proper execution of programmed behaviours
- **The ability to undo prior made choices** which allows for full flexibility in the creation of high level task plans.
- **Making robot actions more robust**, which can be done with backward chaining as is discussed in Sections 3.2.4 and 3.3.13.
- **Saving and loading existing trees**, which enables reusability and modularity of learnt behaviours by facilitating further adaptation.
- **Creating alternative or parallel behaviours**, to combine learnt behaviours into a larger framework capable of performing multiple behaviours or by enabling alternative behaviours that reach a similar overarching task goal.
- **Creating an *user-friendly* interface which allows operation of the framework by users with little to no background in robotic engineering and/or computer programming**

A more comprehensive and in-depth description of the implementation of the mentioned basic principles in a working framework is introduced as the *Behaviour Tree Update Framework* (BTUF) in Section 3.3. A simplified framework, the *Behaviour Tree Creation Framework* (BTCF), is discussed in Section 3.3.14. BTUF and BTCF are compared to one another for the purpose of experimentally validating the former framework. The general system acceptance and overall proof-of-concept of BTUF as a result of this experimental validation is discussed in Section 4.4.

The general reasoning and implementation of the basic principles as mentioned prior in this Section are argued below:

- One of the major benefits of using BTs is the ability to utilize and sequence scripts written using different code conventions and/or programming languages. Therefore a framework that utilizes BTs for high-level task planning can easily make use of a plethora of available code/robot skills written by different people to create complex robot behaviours. An example of a practical implementation is found in Section 3.3.
- As was discussed prior in Chapter 2, we find that complex behaviours such as ADL are composed of smaller behaviours. I.e., small behaviours can be compounded to form larger more complex behaviours. The ability to add and sequence a finite amount of actions/leafs, which can vary in range of complexity with respect to the embedded robot skill, can create increasingly complex behaviours over time. Therefore the ability to incrementally update/add actions to a BT is a necessity for creating increasingly complex behaviours, that over a period of time and amount of adaptation attempts, can account for alternative/novel instances on an initial given task. A tangible example would be that a simple behaviour as pressing a button, can also correctly respond to a scenario where said button can not be located due to hindrance of an obstacle, if the behaviour has been adapted to do so upon an earlier encounter of such a situation. An example of a practical implementation is found in Section 3.3.5.
- Similar to how it is in the natural world, performing actions oftentimes entails a form of reasoning for executing said actions. I.e., certain conditions exist for performing certain actions. In the case of performing robot skills, we can think of conditions as a form of checks that can semantically be represented as a (set of) question(s). An example condition would be to check whether a light has been turned on, before rationalizing whether the skill to turn on the light has to be actually executed or not. In short, it can be said that adding conditions in correlation to their associated actions allows a form of inference or reasoning for performing said action. An example of a practical implementation is found in Section 3.3.6.
- Robustness can be referred to as the ability to execute a given action without an error or problem occurring during the execution thereof. In the case that an action was to be performed without checking first whether that action can be executed at all, one is unsure whether that action will execute successfully or will fail before or during the execution thereof. Backward chaining, as discussed in Section 3.2.4, is a method for reaching a goal given the expected outcome and working backwards to achieve said goal. In the context of BTs, backward chaining introduces a certain structure or type of branch that consists of a fallback of which the children are in respective order a condition and a (sequence of) action(s). This structure allows for first checking a condition before executing a specific (set of) action(s), which increases the robustness of said execution in comparison to doing so without prior condition checking. An example of a practical implementation is found in Section 3.3.7.
- One other beneficial feature of BTs is the reusability thereof. This implies that if a BT is constructed for one system it should be easily transferable to another system. In this scenario in particular, saving and loading of constructed trees can therefore be regarded as a necessity in order to do so. The idea of the proposed framework is to incrementally adapt a BT in order to program complex robot behaviours. In the case that a novel instance on the initially learnt behaviour arises, it can so happen that the execution of the programmed behaviour can not be guaranteed anymore. In this case, the ability to load these behaviours or BTs in order to further adapt them to accommodate said particular novel instance will improve the reusability of the BTs in general. An example of a practical implementation is found in Section 3.3.9.
- As it has been said a multitude of times, a BT can be used for representing complex robotic tasks. A plethora of BT structures can be thought of since there exists little to no comprehensive academic research regarding an optimal structure given certain situations as mentioned in Section 3.3.3. As most tasks can be approached in different manners, alternative approaches to solving

a specific task goal can increase the successful attainment of said task. Thus, a parallel tree which can be referred to as a completely separate branch can pose as an alternative behaviour if appended to a fallback that is placed high up in the BT. An example of such an implementation is found in Section 3.3.10.

- Readability and human interpretability pose as a major advantage and reason for using BTs over their counterpart HFSMs. As was discussed in Chapter 2, BTs consist out of only a few nodes, being control flow, leaf and condition nodes. As opposed to black-box methods such as Machine Learning (ML) methods like Convolutional Neural Networks (CNNs) and conventional robot motion policies such as Dynamic Movement Primitives (DMPs), as discussed in Chapter 2, BTs can be considered as a white-box method as the behavioural reasoning for executing specific skills in order to reach an overarching task goal is clearly readable/interpretable from the BT structure itself. If the inherent easy readability of BTs was to be combined with an easy method for constructing said BTs, the possibility would arise to allow users with little to no programming knowledge to program complex robot behaviours. This advancement is necessary with the increasing use of robotics in domestic and healthcare environments, which brings along the increasing need to locally program/adapt said robots. Therefore a simple-to-use or user-friendly framework for constructing BTs with little to no programming knowledge would open up the doors for the further and larger scale acceptance of adopting robotics inside domestic and healthcare environments. An example of a practical implementation is found in Section 3.3.12.

3.2 Proposed Pipeline

An easy-to-use framework that allows human users with limited programming knowledge to create complex robot behaviours associated with performing ADL is proposed of which the pipeline is depicted in Figure 3.1.

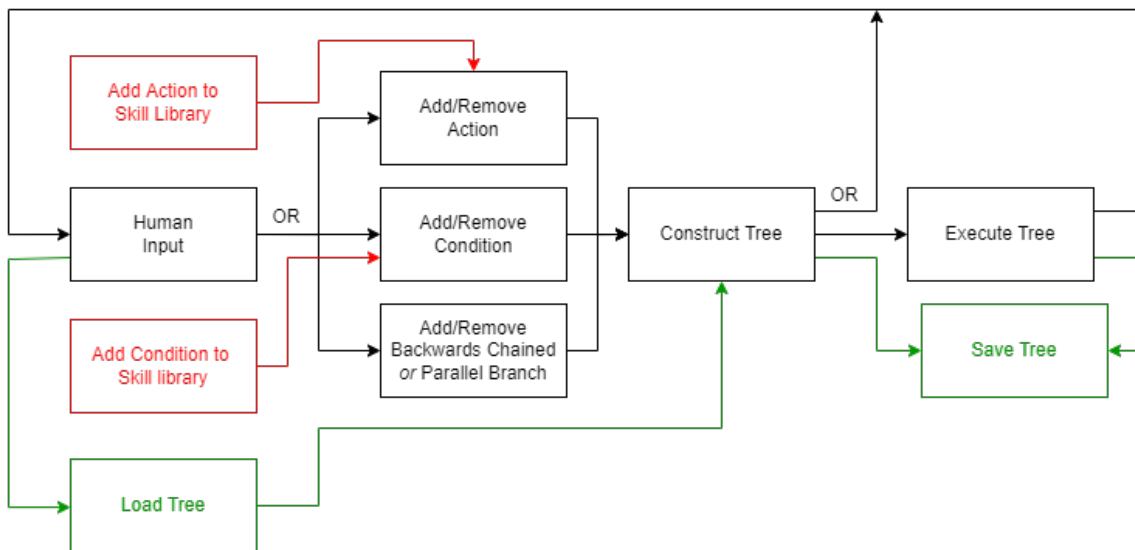


Figure 3.1: Proposed BT pipeline wherein **black** boxes indicate the main pipeline, the **red** boxes indicate expansion of the available action and condition library and **green** boxes indicate saving & loading constructed BTs. **Note** that the method of **expansion of actions and conditions** is open for future work, as discussed in Section 5.2.

This thesis focuses on the implementation of the proposed behaviour tree update pipeline as portrayed in Figure 3.1. This particular implementation will be dubbed and referred to as *the Behaviour Tree Update Framework*, or BTUF in short.

The proposed framework, BTUF, focuses on the initial creation and the incremental adaptation of BTs by appending robots skills to the aforementioned base behaviour in order to account for novel instances thereupon. In much more simpler words, we add novel skills to a base behaviour to account

for alternatives found for said behaviour. For instance, if a base behaviour is to pick up an object, an additional (sub-)BT can be added that tackles the removal of an obstacle interfering with the completion of the picking-task. This should hold true for a plethora of skills, including but not only limited to e.g., moving the robot base to circumvent an obstacle, to ask for help to the human operator if the picking behaviour fails etcetera.

Thus, BTUF should allow for the gradual expansion of a learnt base behaviour to account for the increasing complexity of an initially given task. This method of adapting a simple base behaviour to allow for more alterations and novel instances on the initially given task, should allow for an easier approach to constructing complex robot behaviours. This method has been validated in Chapter 4.

The two main methods for expanding upon existing BTs are forward and backward chaining, which are both discussed in the proceeding Sections 3.2.1 and 3.2.2 respectively.

3.2.1 Forward Chaining

Forwarding chaining is an algorithm used in inferencing engines that uses known facts as a starting point, to trigger all rules to add to the conclusion based on all known facts. I.e., forward chaining looks at all given facts and rules as found within a behaviour tree and deduces all conclusions and solutions therefrom. Forward chaining is referred to as a so-called down-up method, where one moves from the bottom to the top.

Using forward chaining one can deduct all conclusion or solutions as given by the initially given facts and rules. This is also referred to as a data-driven method (Perotti et al., 2015).

Advantages

Some advantages of using forward chaining in inferencing engines are

- multiple paths can be constructed for reaching certain conclusions
- by being a data-drive method a reasonable basis is constructed for reaching specific conclusions
- as it is a data-driven method, the base of reasoning for the reached conclusions are not limited as might be the case with depth-first methods such as backward chaining

Disadvantages

Some disadvantages of using forward chaining in inferencing engines is that

- the reasoning structure for reaching certain conclusion might be unclear to the human reader
- forward chaining is only applicable in the case that a single starting point and multiple end points can be identified. For methods where there is a single endpoint and several starting points for reasoning to reach said endpoint as a conclusion, the method is not utilizable. In the case of the latter scenario, backward chaining can be used

3.2.2 Backward Chaining

Backward chaining is an algorithm used in inferencing engines that uses a form of reasoning in which one starts from a goal and works backwards, chaining rules together to find known facts that support finding the initially stated goal. This method is also referred to as a top down approach, where we take the goal and work in a backwards fashion to find the means to attain said goal. The backward chaining algorithm is based on the modus ponens inference rule. Modus ponens means implication elimination of antecedent affirmation of which an example can be given by the following logical expression

$$P \rightarrow Q \quad , \quad P \vdash Q \tag{3.1}$$

That states if P implies Q and P is true, then Q must also be true.

Within backward chaining the end-goal is divided into sub-goals to prove that the facts that lead to reaching the end-goal are true or satisfied. The method is also referred to as a goal driven approach which utilizes a depth-first search strategy, rather than a data-driven which is the case for forward chaining as seen in the previous Section 3.2.1. A depth-first search strategy refers to an algorithm for traversing or searching a tree structure, wherein the exploration starts at the root node and traverses as far or deep as possible along each of the branches before backtracking (Tarjan, 1972).

Advantages

Some of the main advantages of using backward chaining are that:

- A goal-drive method is a computationally efficient way to reach a desired solution.
- It is an advantageous method to used if the desired result is known a-priori and inferences are to be deducted.
- Backward chaining is quicker and more computationally efficient than forward chaining. as it only checks the required rules to reach a desired goal.

Disadvantages

Some of the disadvantages of backward chaining are:

- That the desired goal has to be known a-priori for backward chaining to be utilized
- That backward chaining is less flexible than forward chaining as no multiple answers or solutions are derived for the given task. Thus, the user of the algorithm is limited to only one and not multiple conclusion which might be a deal-breaker in a decision making process.

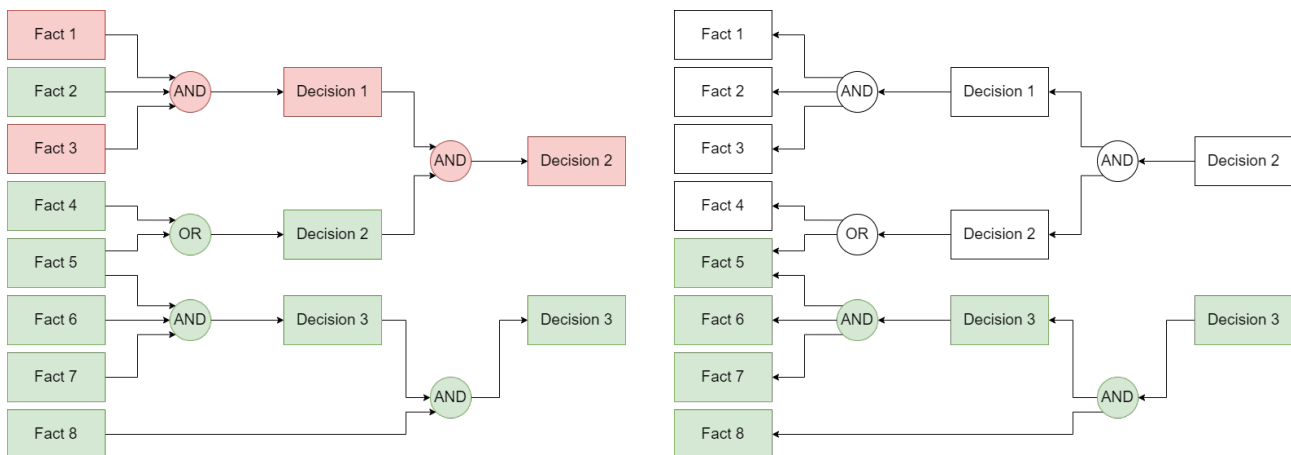


Figure 3.2: Forward chaining (**left**) versus backward chaining (**right**) example. Forward chaining is a breadth-first search method that checks all inference rules or facts before inferring a decision, making it computationally costly. Backward chaining is a depth-first search method that works backwards from the goal. This goal-driven approach works top down from the goal and does not check all inference rules or facts, making it more computational efficient. The structure of the forward chained inference engine is taken from (Nugroho, 2019) and adapted by the author to fit the portrayed example.

3.2.3 Forward versus Backward Chaining

The major differences between forward and backward chaining are that:

- Forward chaining can be utilized as an exhaustive search method, whereas by using backward chaining any unnecessary path of reasoning is avoided. This also stipulates the fact that backward chaining is referred to as a goal-driven approach.
- Thus, this leads to the fact that backward chaining uses fewer questions, as to to say, to reach a specific goal.

- Forward chaining is a slower algorithm as it checks all rules, whereas backward chaining only checks the few required rules necessary for reaching a goal, resulting in it being a much faster algorithm
- This however can result in the fact that, although faster, backward chaining might only come up with one approach to reaching a goal, whereas forward chaining can result in finding alternative paths of reasoning for reaching a certain goal.

Thus, the biggest consideration between choosing either forward or backward chaining is the computational time or efficiency versus the window of reasoning or amount of checked approaches for reaching a certain goal (Isermann, 1993)(Rupnawar et al., 2016).

As an **intermediate conclusion** it can be said that for this framework only backward chaining is utilized. The adaptation process assumes that an initial robot behaviour is constructed via human input and is adapted over time to facilitate and account for novel instances and variations thereupon. In other words, a human operator distinguishes the required actions to form a certain behaviour capable of reaching the desired task goal. Therefore the end-point or goal is known a-priori by the human operator. This means that a data-driven method such as forward chaining, that although being able to reach multiple endpoints, is not required nor necessary. Backward chaining, however, is more beneficial as it tries to tackle an a-priori known endpoint from the goal to start. Therefore backward chaining might prove to be helpful in the construction of BTs as done within the proposed framework. Section 3.2.4 discusses the implementation of backward chaining in BTs in the form of backward chained behaviour trees.

3.2.4 Backward Chained Behaviour Trees

Backward chained (BC) behaviour trees refer to BTs that have been expanded/adapted with additional subtrees by means of backward chaining. To avoid checking all conditions present in a BT all the time, we have two methods for ticking these types of behaviour trees. One of which is frequency based ticking, whilst the other is event based ticking. The former ticks the tree, as the name implies, based on a certain frequency and thus checks the entire tree all the time, whilst the latter only ticks the tree when an action returns either success or failure. I.e., a tick is sent down the tree only when an event occurs. Event-based ticking is a method for going down the tree whilst only ticking if necessary to react to a change of events. Active sensing on the other hand refers to adapting an existing BT when the uncertainty or risk is too big. Meaning that if too long of a time has transpired since the last condition check or that the consequence of a possible error can be big, that we want to update the current behaviour tree to circumvent said uncertainty/risk. A more tangible example can be given with the example wherein a mobile robot fails to open a door within a certain specified time window that one can assume that execution of the task is set out to fail. Thus, it can be opted to adapt the tree in order for the execution to be performed successfully within the specified time window. If conditions are difficult to check, we can think of them as certain *beliefs*. In that case we can add a small subtree to check these beliefs to better estimate the fulfilment of the initial check/condition (Ögren, 2021a).

As stated prior, we find from Ögren (2021b) that a backward chained behaviour is constructed out of recursively substituting conditions with smaller behaviour trees that achieve said substituted conditions. Four design criteria can be identified that need to be checked in order to guarantee reaching all top level goals which are, assuming that we have conditions A , B and an action C :

1. An action should achieve a post-condition in finite time
 - Action C achieves condition B
2. An action does not violate prior achieved sub-goals
 - Action C does not violate condition A
3. An action does not violate conditions needed to execute actions placed further ahead in the tree

- Action C does not violate condition B
4. An action should not invoke a previously failed subtree
- E.g., an action to move a robot base within an object-picking subtree should not invoke a previously failed subtree that also attempted to move the robot base. As a solution to not invoke a previously failed subtree we can add a precondition that deactivates the prior failed subtree.

A post-condition of a certain action A_i and the corresponding Active Constraint Conditions ACC_i can be for example given as seen in the following table as adapted from Ögren (2021b) that resembles a mobile robot pick-and-place task.

Action A_i	Postconditions of A_i	ACC_i
Move to Safe Area	In Safe Area	-
Move to Object	Near Object	In Safe Area
Grasp Object	Object Grasped	In Safe Area
Move to Goal	Near Goal	In Safe Area AND Object in Gripper
Place Object	Object at Goal	In Safe Area

Two ways for finding sub-goals to preserve within a larger backward chained BT is by during execution finding non-preconditions that return success and by analytically studying a BT to check:

- the sequence nodes placed in between the actions and the root of the tree
- older children of those nodes placed to the left
- the pre-conditions of the checked action that can be violated at the instant of achieving the postcondition...
- ... whereafter we deduce that the violation of those pre-conditions are OK if the goal is achieved ...
- ... while keeping in mind that violation of ACC s is not tolerated.

To check for conditions that are needed for execution of later tasks, we can also create a so-called *AND-OR-Tree* which shows the kind of disturbances that can be handled within the BT wherein a sequence becomes AND and a fallback becomes OR (Ögren, 2021b).

Finding Suitable Behaviour Tree Structures

In order to successfully go through a BT as efficiently as possible we need to define a suitable structure for our BTs. From Ögren (2019b) we find a method on how to update BTs via backward chaining as mentioned prior in Section 3.2.4. Consider that we have a simple base tree as depicted in 3.3 wherein we have two conditions connected to a sequence at the root.

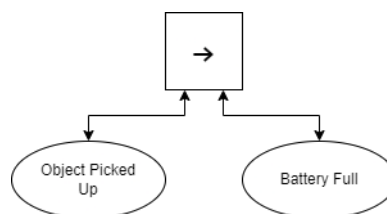


Figure 3.3: Simple BT that consists of a sequence with two conditions C_1 and C_2 as its children.

In order to update the first condition after failure, we substitute said condition with the structure as seen in Figure 3.4 where the condition itself is placed as the first child of a fallback node, of which the subsequent children of the fallback node all satisfy the initially failed condition upon successful execution.

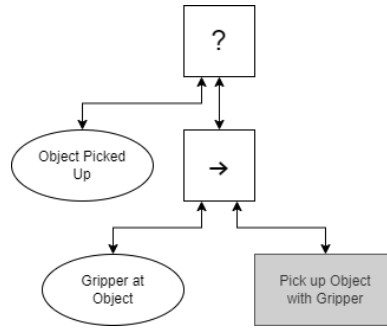


Figure 3.4: Backward chained branch consisting of a fallback R_{BC} at the root of which its children T_1 is a condition and T_2 is a sequence of which the children $T_{2,a}$ and $T_{2,b}$ are a condition and action respectively. This structure first always checks whether T_1 returns **Success**. If not, then T_2 is ticked. If $T_{2,a}$ returns **Success**, then $T_{2,b}$ will be executed. Upon successful execution thereof, R_{BC} will return **Success**. Also, R_{BC} returns **Success** if T_1 returns **Success**.

As can be seen from the example as depicted by Figure 3.4, condition C_1 *Object Picked Up* is also satisfied upon successful completion of sequence T_2 if C_1 is to initially return **Failure**.

This method of adapting BTs using backward chaining, adding condition-action pairs that satisfy a(n initially failed) condition, leads to gradual expansion of the tree with subtrees that satisfy the prior failed conditions as seen in Figure 3.5. In other words, backward chaining certain conditions and/or actions within a BT increases the robustness or the chance of successfully fulfilling said condition or executing said action.

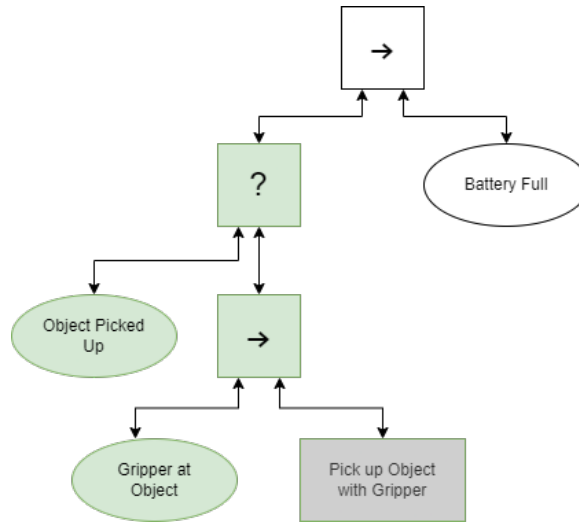


Figure 3.5: Backward chained BT where C_1 from Figure 3.3 is substituted with R_{BC} from Figure 3.4, effectively backward chaining the *Object Picked Up* condition from C_1 to be more robust.

We have seen several general examples of expanding BTs to facilitate increasingly complex behaviours that can react to different scenarios/variations of the original task in Section 2.6. An example of a high-level task plan which utilizes BTs specific to our topic would obviously portray a behaviour commonly encountered with mobile robots which in this case is a pick-and-place task, as shown in Figure 3.6.

From Figure 3.6 it can be seen that a BT that consisting solely out of a *sequence* and two *conditions* is prone to failing easily. In the case that the *sequence* returns **Failure**, it is desirable to ask a human operator for help. This desire results in our initial BT to include a *fallback* for both conditions that results in asking a human operator for help upon failing said conditions. With this BT as a baseline, the initial behaviour can be adapted and expanded throughout several cycles of trial and error, finally resulting in a BT capable of solving the initial pick-and-place task given novel instances or variations on the task, as depicted in Figure 3.7.

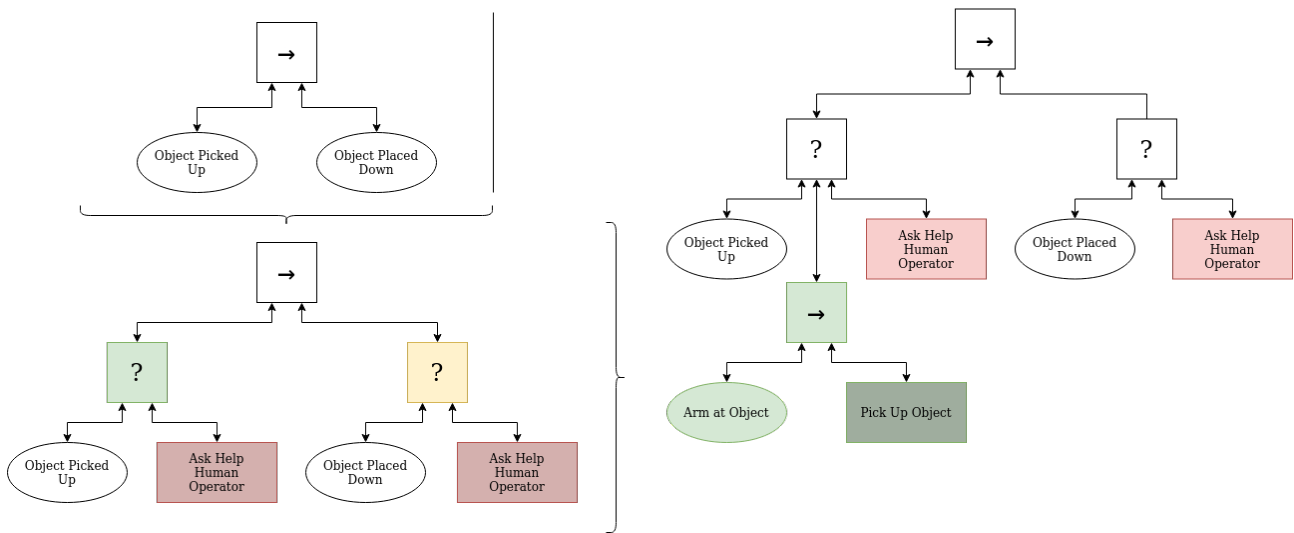


Figure 3.6: Example 1 of backward chaining to create a larger BT by substituting conditions with sub-trees.

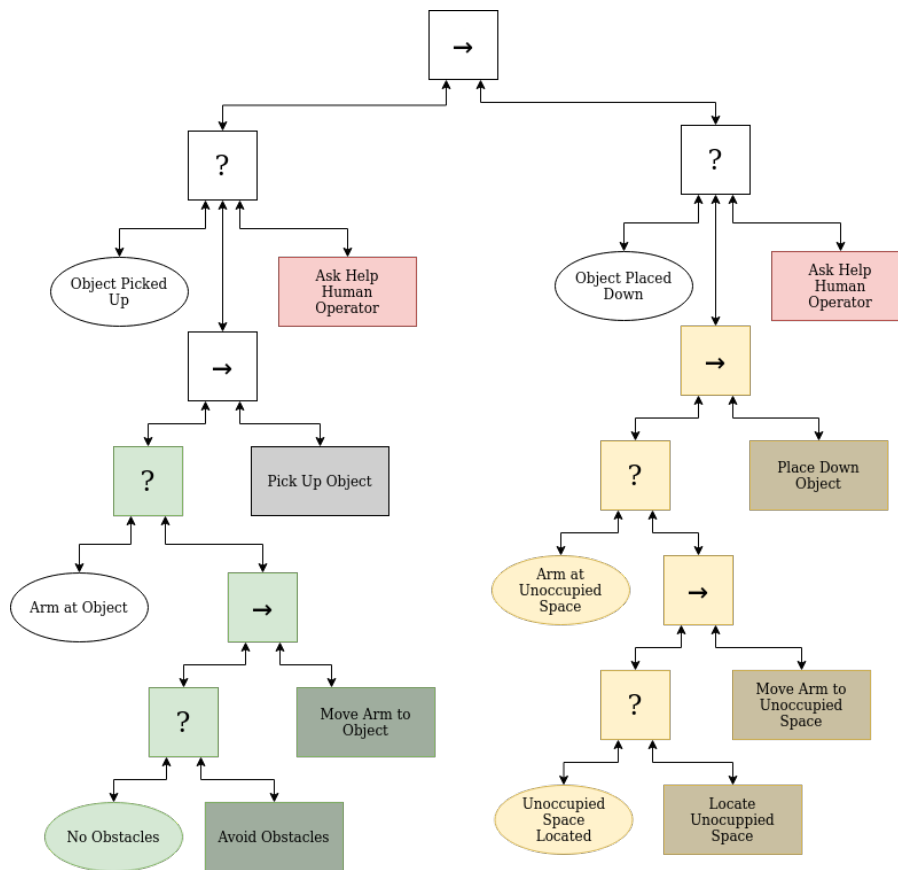


Figure 3.7: Example 2 of backward chaining to create a larger BT by substituting conditions with sub-trees.

Note that the practical implementation of BTUF discussed in Section 3.3 does utilize a specific node that asks a human operator for help, but rather uses a so-called update loop embedded within a Text-Based User Interface (TUI) that asks for human input after every adaptation/execution step in the BT creation process as found in Algorithm 15. Moreover, it has to be **noted** that the example BT as shown in Figure 3.7 concerns a sequential BT. The introduction of an additional node, the *parallel* node, results in faster execution of a BT as it allows for parallel execution of multiple actions at the same given time instance, previously discussed in Section 3.3.2. However, within the proposed framework these node types are not considered. Section 3.3.3 discusses the BT structures that are constructed with the proposed *Behaviour Tree Update framework*.

3.3 Practical Implementation

This section discusses a practical implementation of the methods as described in the prior sections of this chapter in the form of the own proposed *Behaviour Tree Update Framework* (BTUF).

Section 3.3.1 discusses the proposed core features of BTUF. Section 3.3.2 discusses the different node types as used within BTUF. Section 3.3.3 discusses different BT structures as found in literature and discusses which one is utilized within BTUF. All of the features as found within BTUF, as well as an overview of the full framework itself, are discussed by means of pseudo-code in Sections 3.3.4, 3.3.5, 3.3.6, 3.3.7, 3.3.8, 3.3.9, 3.3.10, 3.3.11 and 3.3.12. Section 3.3.13 discusses methods implemented in BTUF to ensure robustness. A conventional framework in the form of *Behaviour Tree Creation Framework* (BTCF) is proposed in 3.3.14 which has been used as a comparative framework to BTUF for the experimental validation as discussed in Chapter 4.

3.3.1 Proposed Framework Features

BTUF makes use of a terminal based UI (User Interface) or TUI (Text-Based User Interface), wherein human inputs are required to create a BT either from scratch or by adapting a prior existing BT in order to teach an agent specific behaviours with the prospect of successfully executing certain robotic tasks. Core features of BTUF are:

- Adding/inserting actions.
- Adding/inserting conditions.
- Backward chaining (substituting existing leafs *or* inserting entire backward chained branches).
- Removing actions, conditions, backward chained branches and/or parallel branches.
- Saving created BTs.
- Loading created BTs (to either execute, to further adapt or to use as an extension to the current tree in the form of a parallel branch).
- Creating parallel branches/behaviours.
- Executing BTs.

3.3.2 Node Types

The node types used within BTUF are found in Figure 3.8 below.



Figure 3.8: (From Left to Right) **Sequence**: Succeeds if all children N succeed. **Fallback**: Succeeds if one child M of N children succeeds. **Condition**: Checks if a certain condition or world state has been achieved. **Leaf**: or **action** node, represents an executable file or robot skill.

Note that other node types such as parallel nodes (of which all children N are ticked at the same time instance) and decorators (which change the output of a node given a custom function) exist, but will not be further discussed as these are not used within BTUF.

3.3.3 Behaviour Tree Structures

Three different types of BT structures can be identified as:

- **Deep** - This structure is mostly used when creating backward chained behaviour trees. Backward chaining is a goal-solving method by working backwards from the goal state to the initial state. In the case of backward chained BTs, a failed condition can be substituted with a particular BT branch structure that achieves the prior failed condition. This is mostly done by substituting a failed condition, with a branch consisting of a fallback, of which the children are from left to right 1) the initial condition 2) a singular action or a sequence of actions meant to achieve the same goal condition as child 1) of the overarching fallback node. In other words, backward chaining is used in order to make the successful execution and fulfillment of a condition more robust. **To summarize:** a deep BT structure allows for making a condition more robust. However, expanding a condition too deep might cause implications on readability of the tree as it increases the complexity thereof, which is one of the most important benefits of using BTs.
- **Wide** - This structure allows for adding additional condition checks and/or actions that form behaviours associated with novel task instances that would have to be executed before being able to successfully execute the initial base behaviour upon their occurrence. One example being that before an object can be grasped successfully by the dexterous manipulator of a mobile robot, that it first has to circumvent an obstacle by moving the robot base around said obstacle. This structure, albeit being very minimalistic, would require the tree to be ticked several times before successfully satisfying the base behaviour, ultimately making the perceivability of the BT confusing. In the case that we want to tick the BT as minimally as possible when using a wide BT structure, the adapted branches that tackle novel instances should contain the adapted skill within a sequence alongside a duplicate of the base skill. In other words, parallel branches will have to be constructed. Using duplicate nodes, however, is considered to be bad practice and thus should not be considered. **To summarize:** a wide structure, albeit looking minimalistic, can become confusing when becoming too large as it is required to go through the BT several times before being able to execute the base behaviour by means of successful execution of the adapted behaviours a-priori in a satisfactory fashion.
- **Hybrid** - The hybrid structure refers to going both into the depth and width when adapting a BT. Using this BT structure we can make robot skills more robust by adapting into the depth of a particular branch via backward chaining, whilst we can adapt alternatives of the base behaviour by going into the width of the tree by adding parallel branches. In other words the hybrid structure mitigates the downsides of both the deep and wide structures out of which it is constructed.

Given the structures above, it is hypothesised that the *hybrid* structure will be preferred by the participants as it combines the merits of both the *deep* and *wide* structure, whilst not sacrificing the human readability and understanding of the robot task as portrayed by the BT. Therefore, BTUF makes use of all three different structures by allowing the user to either adapt into the depth (via Backward Chaining) or into the width (either extending the base behaviour or creating a parallel sub-branch with an alternative or different behaviour) of which the combination of the former structures form an overall hybrid BT as the outcome. In short, BTUF is a framework that provides easy creation and subsequent adaptation of BT structures.

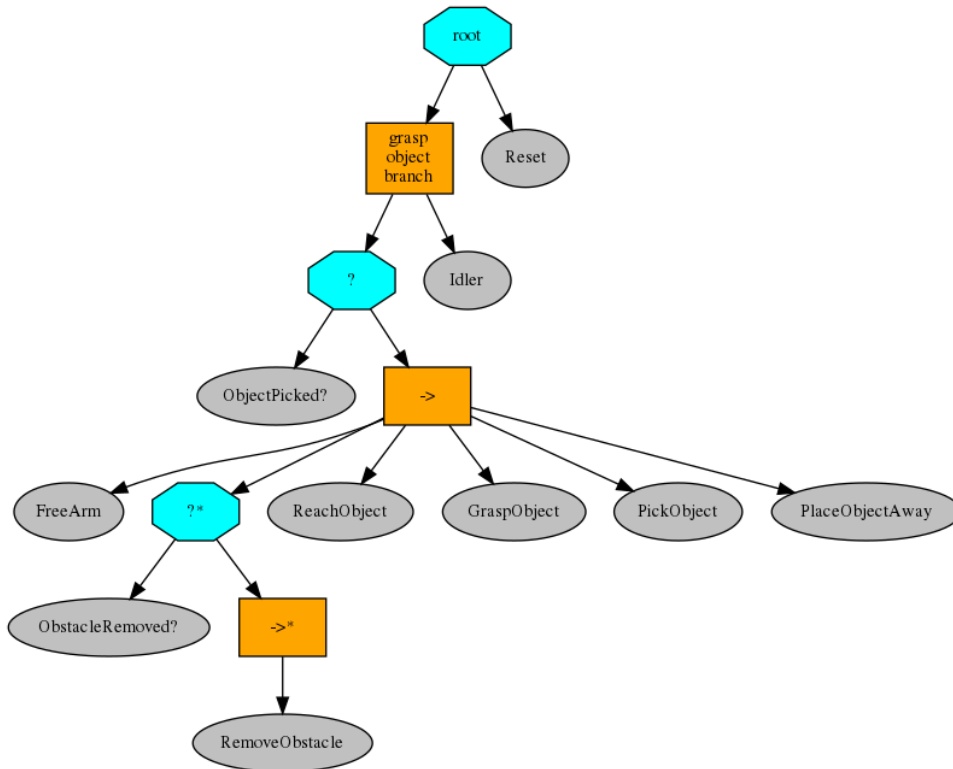


Figure 3.9: Example BT constructed using BTUF for executing a pick-and-place task while accounting for a possible obstacle that has to be removed if present. **Note** that this exact BT structure is a possible solution to the task goal asked to the experiment participants as discussed in the experimental design in Section 4.1.2 and found back in the participant handout in Appendix 6.

Within the practical implementation of BTUF as done in this thesis, the BT structure is created as seen in Figure 3.10. A base structure is automatically generated for all BTs constructed within BTUF. As discussed in Section 3.3.1, the functions of BTUF include adding condition and action nodes as well as inserting backward chained branches and parallel behaviours or branches. All of these node or branch types are, if added to a BT under construction, always added at a pre-specified location within the tree as seen in Figure 3.10 and Algorithm 4. The reasoning for this specific structure as found from Figure 3.10 is explained per node as follows:

- The root R_0 is a fallback. Parallel branches can be manually inserted as children to R_0 by the user via Algorithm 14. The fallback structure allows for parallel branches to be executed as alternative behaviours if a prior behaviour branch fails within the BT. R_0 has two children: a sequence S_0 and an idler skill I_0 dubbed reset. I_0 returns the robot to its original idle state if S_0 returns **Failure**.
- Sequence S_0 denotes the robot behaviour as represented by that specific branch, which in this case is to grasp an object. S_0 has two children: a fallback F_0 and an idler skill I_1 . I_1 returns the robot to its original idle state if F_0 returns **Success**.
- F_0 forms a structure similar to a backward chained behaviour tree, discussed in Section 3.2.4. Users can add one or multiple condition nodes $C_{i..j}$ as children of F_0 that, if added, form condition checks before attempting to execute sequence S_1 .
- S_1 , standard a child of F_0 , has as children either action nodes $A_{i..j}$ or backward chained (BC) branches $BC_{i..j}$ as added by the user.

In short, the manner of BT construction within BTUF is set up in such a way that if both the desired conditions and actions are specified by the user for performing a certain behaviour, the BT is structured as a BC BT. This structuring allows for some level of robustness as discussed in Section 3.3.13.

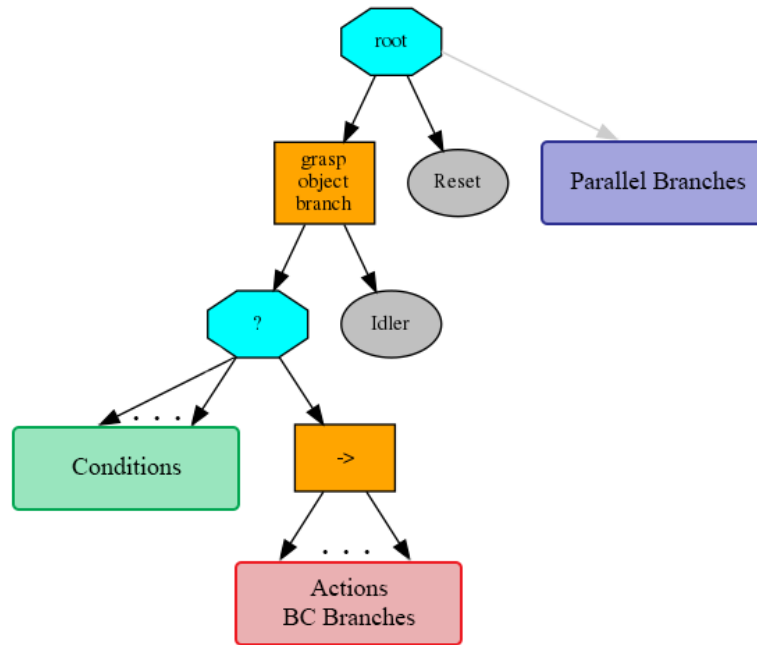


Figure 3.10: Base BT structure automatically generated when creating behaviours using BTUF. The **green** box indicates the location where conditions are placed via Algorithm 8. The **red** box indicates the location where actions and backward chained (BC) branches are added via Algorithm 7 and 9. The **blue** box indicates the location where parallel branches are added via Algorithm 14.

With respect to the BT structures created using BTUF, some remarks can be given in terms of the current limitations which are summed up as:

1. As the BT structures are mostly premeditated within BTUF, not a lot of room exists for the user's creative freedom with respect to the structuring of BTs. In other words, there is a trade-off between a framework that facilitates easy to construct behaviours versus one that allows full constructional freedom of BT structures, but requires more background knowledge. For BTUF the former is chosen as the end-goal is to achieve a framework that is easily operable by users with little to no programming knowledge.
2. BTUF, as of now, does not allow for backward chaining action nodes found within a backward chained branch. In other words, there is a limitation to the level of *depth* that one can achieve with BTs created using BTUF.
3. BTUF, as of now, does not allow for the adaptation of individual nodes as found within parallel branches.

The main reasoning for limitations 2 and 3 was that the time window of this thesis did not allow for the addition of the two discussed features, which would add new levels of complexity to the framework with respect to the programming thereof.

3.3.4 Constructing and Printing a Behaviour Tree

The main Python package used for creating BTs within BTUF is the `py_trees` BT library, of which most of the algorithms discussed in the sections below utilize some functionalities thereof.

Note that although all of the algorithms assume that Python and `py_trees` is used, the reader can choose to use any other BT library or programming languages such as those found in Iovino et al. (2020).

Algorithm 4: Constructing base structure (`create_root_structure()`)

```

input : None → automatically generated
output: Uncompiled base structure of BT

1 create fallback  $R_0$ ;                                /* root node */
2 create sequence  $S_0$ ;
3 create fallback  $F_0$ ;
4 create sequence  $S_1$ ;
5 create idler  $I_0$ ;
6 create idler  $I_1$ 

7 add  $S_1$  as child of  $F_0$ ;
8 add  $F_0, I_1$  as children of  $S_0$ ;
9 add  $S_0, I_0$  as children of  $R_0$ ;
10 return  $R_0$  as tree

```

Algorithm 4 creates the base BT structure as discussed in Section 3.3.3 and as seen in Figure 3.10. Line 1-6 create the control flow nodes and idler nodes of which the base structure consists of, after which line 7-9 properly structure them. The base structure as created with Algorithm 4 is the standard used for all BTs created using BTUF for which the reasoning has been discussed prior in Section 3.3.3.

Algorithm 5: Compiling and constructing Behaviour Tree (`construct_tree()`)

```

input : Uncompiled BT structure
output: Compiled BT structure

1 construct tree as BehaviourTree;
2 add pre_tick_handler to tree;
3 add post_tick_handler to tree;
4 add debug_visitor to tree;
5 add snapshot_visitor to tree;
6 setup tree;

```

Algorithm 5 compiles a `py_trees` BT with R_0 as root. This allow the BT to be executed, visualised via Algorithm 6 and to be incrementally adapted via Algorithm 15 line 20-47. `pre_tick_handler` and `post_tick_handler` are `py_trees` functions that allow for logging and extracting data from the blackboard. A behaviour tree blackboard (BB) is similar to a list containing global variables. Thus, a blackboard contains variables that are accessible by all node instances of a given BT. `debug_visitor` allows for printing logging messages, while `snapshot_visitor` collects run-time data that is used to visualize the constructed BT with Algorithm 6.

Algorithm 6: Printing Behaviour Tree (`print_tree()`)

```

input : Compiled BT structure (construct_tree())
output: Renders a visual representation of the constructed BT

1 render tree;
2 print tree;

```

Algorithm 6 renders the constructed BT as a dot graph after which it is visualized in a `.png` format.

3.3.5 Adding Robot Skills

Algorithm 7: Adding robot skills (`add_action()`)

```

input : Key value corresponding to robot skill in action dictionary
output: Robot skill embedded within an action node inserted into BT structure

1 [INPUT] Do you want to [1] Append [2] Insert Action?;
2 print action_dictionary;
3 [INPUT] Which robot skill do you want to add? → user_action_input;
4 if user_action_input = int in range action_dictionary then
5     embed action_dictionary[user_action_input] as action node;
6     if [1] Append Action then
7         append user_action_input to user_action_list;
8         append action_id to action_id_list;
9         add action node as child in tree;
10        print_tree();
11    end
12    else if [2] Insert Action then
13        [INPUT] Specify index to insert node → action_index_input;
14        if action_index_input = int in range user_action_list then
15            insert user_action_input in user_action_list at action_index_input;
16            insert action_id in action_id_list at action_index_input;
17            insert action node as child in tree at action_index_input;
18            print_tree();
19        end
20 else
21     return to line 3;

```

Algorithm 7 encapsulates robot skills found in `action_dictionary` into `py_trees` actions nodes, which in turn can be inserted into the BT being constructed/adapted. `user_action_list` contains all actions take by the user via so-called tele-selection, which is updated every time a new action node is added to the tree. Every node within a BT constructed using `py_trees` has a unique child identity (ID). `action_id_list` contains the IDs of all the actions taken by the user. Thus, `user_action_list` and `action_id_list` have the same length of which the elements target the same specific action nodes, but of which the lists have a different purpose respectively. The reason for using two different lists is that `user_action_list` is used in Algorithms 12 and 13 to save and load a BT respectively, while `action_id_list` can be used to target a specific node within that list for removal with Algorithm 10 or to be backward chained with Algorithm 9.

Note that the scripts listed in `action_dictionary` are Python scripts, but do not have to be so in particular and can be written in a plethora of languages in different coding styles. Therefore, these scripts should be able to be executed as action nodes within a BT as long as the scripts themselves are able to run on the used robotic agent in question. The dictionary containing the robot skills can be manually expanded by the user. The reader is referred to Section 5.2 for future work that proposes an automated expansion of the `action_dictionary` with actions learnt from demonstration.

3.3.6 Adding Conditions

Algorithm 8: Adding conditions (`add_condition()`)

```

input : Key value corresponding to condition in condition dictionary
output: Embedded condition node inserted into BT structure

1 [INPUT] Do you want to [1] Append [2] Insert Condition?;
2 print condition_dictionary;
3 [INPUT] Which condition do you want to add? → user_condition_input;
4 if user_condition_input = int in range condition_dictionary then
5     embed condition_dictionary[user_condition_input] as condition node;
6     if [1] Append condition then
7         append user_condition_input to user_condition_list;
8         append condition_id to condition_id_list;
9         add condition node as child in tree;
10        print_tree();
11    end
12    else if [2] Insert condition then
13        [INPUT] Specify index to insert node → condition_index_input;
14        if condition_index_input = int in range user_condition_list then
15            insert user_condition_input in user_condition_list at condition_index_input;
16            insert condition_id in condition_id_list at condition_index_input;
17            insert condition node as child in tree at condition_index_input;
18            print_tree();
19        end
20 else
21     return to line 3;

```

Algorithm 8 allows for the insertion of condition nodes found in `condition_dictionary` into the BT being constructed/adapted. `condition_dictionary` contains Python scripts that construct a specific `py_trees` condition node. These conditions check specific variables that either represent world states or variables that correlate to whether certain robot skills have been successfully executed or not. An example is a condition node that checks firstly whether an obstacle is present in the world and secondly whether the robot skill to remove said obstacle has executed successfully or not. Similar to `action_dictionary`, `condition_dictionary` can be manually expanded by the user. Similarly to Algorithm 7, two different lists are created, being `user_condition_list` and `condition_id_list`, of which the functionality are similar to their action counterparts as discussed in Section 3.3.5 with the exception of the condition ID not being used as a target for backward chaining.

3.3.7 Backward Chaining

Algorithm 9 allows for backward chaining existing action nodes or to insert a newly created backward chained (BC) branch within a BT. If an existing action node is targeted to be BC, the algorithm goes into line 13-22. These lines in Algorithm 9 remove the original action node ID from `action_id_list`, remove the targeted node from `user_action_list` and reinsert the root ID of the BC branch `backward_chaining_target_id` and `backward_chaining_action_list` in those locations respectively instead. Simply said, a BC BT's `user_action_list` is a nested list of which the nested elements indicate a BC branch. These nested lists can in turn be identified by Algorithm 13 to successfully reload and reconstruct a BC BT. Backward chaining has been discussed prior in section 3.2.4.

Note that all BC branch root IDs (all instances of `backward_chaining_target_id`) are added to `action_id_list` containing all action nodes. When targeted for removal, BC branches are removed in full. When targeted for backwards chaining, a BC branch is removed and substituted for the new BC branch. Thus, we only have to save the root ID of the BC branch for targeting purposes.

Algorithm 9: Backward chaining (`backward_chaining()`)

```

input : Index of action node to be substituted for backward chaining or index wherein to
         insert new backward chained branch
output: Backward chained BT

1  [INPUT] Do you want to [1] Update Existing Action/Branch [2] Create & Insert New Branch?;
2  create fallback node  $R_{BC}$  → backward_chaining_target_id; /* root ID of BC branch */
3  add_condition() → lines 2-5 & 9; /* condition appended as child  $T_1$  of  $R_{BC}$  */
4  append user_condition_input to backward_chaining_action_list;

5  create sequence node  $T_2$ ; /*  $T_2$  is a child of  $R_{BC}$  */
6  print action_dictionary;

7  [INPUT] Specify number of actions → backward_chaining_action_input;
8  for i in range backward_chaining_action_input do
9  | [INPUT] Enter Action i → user_action_input;
10 | append user_action_input to backward_chaining_action_list;
11 | add_action() line 5 & 9; /* action appended as child  $T_i$  of  $T_2$  */
12 end

13 if [1] Update Existing Action/Branch then
14 | [INPUT] Specify index of child for backward chaining → backward_chaining_index_input;
15 | if backward_chaining_index_input = int in range action_id_list then
16 | | remove child at action_id_list[backward_chaining_index_input];
17 | | pop action_id_list[backward_chaining_index_input];
18 | | pop user_action_list[backward_chaining_index_input];
19 | | insert backward_chaining_target_id in
20 | | | action_id_list[backward_chaining_index_input]; /* contains IDs of all actions
21 | | | and BC branch roots, which are in turn children of  $S_1$  */
22 | | insert backward_chaining_action_list in
23 | | | user_action_list[backward_chaining_index_input];
24 | | insert  $R_{BC}$  as child in tree[backward_chaining_index_input];
25 | | print_tree();
26 else if [2] Create & Insert New Branch then
27 | [INPUT] Specify index where to insert backward chained branch: →
28 | | backward_chaining_insert_input;
29 | | insert backward_chaining_target_id in
30 | | | action_id_list[backward_chaining_insert_input];
31 | | insert backward_chaining_action_list in
32 | | | user_action_list[backward_chaining_insert_input];
33 | | insert  $R_{BC}$  as child of  $S_1$  in tree;
34 | | print_tree();
35 end

```

3.3.8 Removing Actions, Conditions, Backward Chained Branches and Parallel Behaviours

Algorithm 10: Removing actions, conditions or backward chained branches (`remove_nodes()`)

```

input : Keyboard input corresponding to node/branch type and specific index of the to be
         removed node/branch
output: Diminished BT structure by removal of indicated node/branch

1  [INPUT] Remove [1] Condition [2] Action/Backward Chained Branch: →
   remove_condition_or_action;
2  if [1] Remove Condition then
3  | [INPUT] Please type index of the top level condition node you want to remove: →
   | user_condition_remove_input;
4  | if user_condition_remove_input in range user_condition_list;
5  | then
6  | | remove_child_by_id(condition_id_list[user_condition_remove_input]);
7  | | pop condition_id_list[user_condition_remove_input];
8  | | pop user_condition_list[user_condition_remove_input];
9  | | print_tree();
10 | end
11 else if [2] Remove Action/Backward Chained Branch: then
12 | [INPUT] Please type index of the action node you want to remove: →
   | user_action_remove_input;
13 | if user_action_remove_input in range user_action_list;
14 | then
15 | | remove_child_by_id(action_id_list[user_action_remove_input]);
16 | | pop action_id_list[user_action_remove_input];
17 | | pop user_action_list[user_action_remove_input];
18 | | print_tree();
19 | end
20 else
21 | return to line 1
22 end

```

Algorithm 10 allows for the removal of actions, conditions and BC branches if they are present in the current BT. All action node IDs are found within `action_id_list`, whereas all condition node IDs are stored within `condition_id_list` as discussed in Sections 7 and 3.3.6 respectively. Moreover, as discussed prior in Section 3.2.2, the root ID of all BC branches are saved within `action_id_list` in Algorithm 10. When targeting a specific action node or BC branch for removal, that specific node ID (or root node ID in the case of a BC branch) is targeted using the `remove_child_by_id()` function. Thereafter, their respective elements are removed from both `user_action_list` and `action_id_list` respectively. This is true for conditions nodes as well. Upon removal of the targeted condition node, their respective elements are removed from `user_condition_list` and `condition_id_list`. **Note** that when targeting the root of a BC branch for removal, the entire branch is removed from the tree.

Note that the `pop()` function used in Algorithms 9 and 10 remove a specific element from a targeted list.

Algorithm 11: Removing parallel branches (`remove_nodes()`)

input : Index of parallel branch root
output: Diminished BT structure by removal of indicated parallel branch

```

1 [INPUT] Specify index of the parallel branch to be removed from root: →
   user_parallel_branch_remove_input;
2 if user_parallel_branch_remove_input in range parallel_branch_id_list then
3   remove child_by_id(parallel_branch_id_list[user_parallel_branch_remove_input]);
4   pop parallel_branch_id_list[user_parallel_branch_remove_input];
5   print_tree();
6 else
7   return to line 1
8 end

```

Algorithm 14 creates parallel branches, which come in the form of saved BTs loaded as children of the root R_0 of a BT currently under construction. Algorithm 11 allows for the removal of these parallel branches. The root ID `parallel_branch_id` of all respective parallel branches added to the BT under construction are saved to `parallel_branch_id_list` in Algorithm 14. In turn, Algorithm 11 can target each of the parallel branches for removal using this same ID list.

Note that Algorithms 10 and 11 are found underneath the same function name. The reason for their separation into two different Algorithms in this section is to have an easier description of both functions using pseudo-code. Algorithms 10 and 11 are separated with a simple **input** statement in line 36 of Algorithm 15 that asks the user what type of node or branch he/she wants to remove. Moreover, parallel branches are added and removed in full. The current implementation of BTUF does not allow going into parallel branches and targeting individual nodes therein as this adds to the framework's complexity. The addition of parallel branches within BTUF only stipulates the ability to add prior constructed BTs as parallel or alternative behaviours within one BT.

3.3.9 Saving & Loading Behaviour Trees

Algorithm 12: Saving Behaviour Tree (`save_tree()`)

input : Constructed BT
output: Pickle list containing all nodes of saved BT

```

1 [INPUT] Save file as: → file_save_name;
2 save user_action_list as file_save_name;
3 save user_condition_list as file_save_name + '_condition';

```

The `save_tree()` function in Algorithm 12 saves the `user_action_list` and `user_condition_list` in a local directory on the user's computer using the `Pickle` Python package under a user specified name. These so-called pickle files can then be loaded into Algorithm 13 or 14 and unpacked to their original list format. Algorithm 13 fully reconstructs a saved BT from the loaded `user_action_list` and `user_condition_list`. Algorithm 14 reconstructs a saved tree via Algorithm 13 and adds it as a child to the root R_0 of a BT under construction, creating a parallel branch or behaviour.

Algorithm 13: Loading saved Behaviour Tree (`load_tree()`)

```

input : Pickle list containing all nodes of saved BT
output: Constructed BT

1 [INPUT] Specify name of file to load: → file_load_name;
2 open file_load_name as user_action_list ;                               /* try */
3 open file_load_name + '_condition' as user_condition_list ;           /* try */

4 create_root_structure();
5 if user_action_list not empty then
6   for i in range user_action_list do
7     if item in user_action_list is list then
8       backward_chaining_loaded_tree = user_action_list[item];
9       backward_chaining_condition_list = backward_chaining_loaded_tree[0] ; /* used
10      in add_condition() to reconstruct BC branch condition */
11      pop backward_chaining_loaded_tree[0] → backward_chaining_action_list ;
12      /* remove element 0 correlating with condition node to form list of
13      actions */
14      backward_chaining() line 2,3,5,8,10,11;
15      append backward_chaining_target_id to action_id_list;
16      append backward_chaining_action_list to user_action_list;
17      append  $R_{BC}$  as child in tree;
18     else
19       add_action() line 5, 7-9;
20     end
21   end
22 if user_condition_list not empty then
23   for i in range user_condition_list do
24     add_condition() line 5, 7-9;
25   end
26 construct_tree();
27 print_tree();

```

The `load_tree()` function in Algorithm 13 reconstructs a saved BT using two pickle files created with Algorithm 12 containing its respective list of actions (`user_action_list`) and conditions (`user_condition_list`). Line 6-17 of Algorithm 13 iterate through each element in `user_action_list`. At every iteration it checks whether the element is a nested list, indicating a BC branch. If an element in `user_action_list` is identified as a nested list, and thus a BC branch, then Algorithm 13 goes into line 7-14 to reconstruct that specific BC branch. If no nested list is found, the element is used to construct an action node in line 16. No actions, BC branches or conditions are added if the corresponding lists `user_action_list` or `user_condition_list` are empty.

3.3.10 Creating Parallel Behaviours

Algorithm 14: Creating parallel behaviours (`create_parallel_branch()`)

```

input : Pickle list containing all nodes of saved BT
output: Saved BT reconstructed as parallel branch on root node

1 [INPUT] Specify branch name: → branch_name;
2 load_tree() → set parallel root as  $P_0$  ;                               /* name=branch_name */
3 append parallel_branch_id to parallel_branch_id_list;
4 add  $P_0$  as child of  $R_0$  in tree;

```

Algorithm 15: Full Behaviour Tree Update Framework

```

input : User keyboard inputs to target and operate specific functions
output: A visualized and executable BT

1 [INPUT] [1] Create tree [2] Load tree [3] Exit Program: ;
2 if [1] Create Tree then
3   | create_root_structure();
4   | construct_tree();
5   | [INPUT] [1] Add condition [2] Add action: ;
6   | if [1] Add condition then
7   |   | add_condition();
8   | else if [2] Add action then
9   |   | add_action();
10 else if [2] Load tree then
11 | load_tree();
12 else if [3] Exit Program then
13 | [Input] Save tree? [1] Yes [2] No: ;
14 | if [1] Yes then
15 |   | save_tree();
16 | else if [2] No then
17 |   | pass;
18 |   exit BTUF;
19 end
   /* start update loop (line 20-47) */
20 [INPUT] Do you want to [1] Execute Behaviour [2] Update Behaviour [3] Exit Program?: ;
21 if [1] Execute Behaviour then
22 | tick() tree; /* tree is ticked once */
23 else if [2] Update Behaviour then
24 | [INPUT] Do you want to [1] Add new Conditions/Actions [2] Backward Chaining [3]
   | Parallel Branch [4] Remove Leafs/Branches?: ;
25 | if [1] Add new Conditions/Actions then
26 |   | [INPUT] Add [1] Condition [2] Action;
27 |   | if [1] Condition then
28 |   |   | add_condition();
29 |   | if [2] Action then
30 |   |   | add_action();
31 | else if [2] Backward Chaining then
32 |   | backward_chaining();
33 | else if [3] Parallel Branch then
34 |   | create_parallel_branch();
35 | else if [4] Remove Leafs/Branches then
36 |   | [INPUT] Remove [1] Conditions/Actions/Backward Chained Branches [2] Parallel
   | Branches?: ;
37 |   | if [1] Conditions/Actions/Backward Chained Branches then
38 |   |   | remove_nodes() → Algorithm 10;
39 |   | else if [2] Parallel Branches? then
40 |   |   | remove_nodes() → Algorithm 11;
41 else if [3] Exit Program then
42 | go to line 13;
43 [INPUT] Do you want to execute/update the tree again? [1] Yes [2] No: ;
44 if [1] Yes then
45 | return to line 20; /* start new update loop */
46 else if [2] No then
47 | go to line 13; /* go into save statement before exiting BTUF */

```

3.3.11 Executing a BT

After constructing a BT with Algorithm 5, the tree can be ticked and executed over an user specified amount of iterations and frequency. Within this thesis, the tree is ticked once at every execution run using the tick() function as seen in line 22 of Algorithm 15. An execution run is defined from start to finish as the user input to tick the tree until the time instance where the root of the BT returns a status output of either **Success** or **Failure**. After every execution run, the user is asked whether to execute or update the tree again as seen in line 43 of Algorithm 15. The BT ultimately succeeds if the root returns **Success**.

3.3.12 Full Behaviour Tree Update Framework

The full Behaviour Update Tree Framework as laid out in Algorithm 15 effectively embeds all of the earlier discussed Algorithms 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 and 14 into a terminal based UI or text-based UI (TUI). The reasoning for using a TUI is to allow for easy and intuitive operation of the framework as well as improving user experience.

BTUF has been experimentally validated using both a pilot and final test setup as found in Section 4.1. The results regarding the overall system acceptance of BTUF, as well as remarks and comments with respect to the UI have been recorded during the experiments as found in 4.4 and transcribed in Appendix 6 respectively. Future work based on these results and remarks is proposed in section 5.2.

3.3.13 Ensuring Robustness

Robustness with respect to BT execution can be referred to as the ability to successfully execute behaviours. Thus, increased robustness decreases the chance of preempted behaviours or failed executions thereof due to erroneous occurrences. In the case of the practical implementation of BTUF within this thesis project, several methods of warranting some level of robustness have been applied, which are summarized as:

- Backward chaining, discussed in Sections 3.2.4 and 3.2.2, allows for a degree of robustness due to a condition always being checked before attempting to execute a certain behaviour. Only upon failure of the initial condition check will the behaviour be executed. This particular behaviour in turn, upon successful execution thereof, results in a similar outcome to the failed condition check. It has to be **noted** that users have to warrant this type of robustness that utilize conditions checks using BC branches themselves when constructing BTs. This has also been discussed in Section 4.5.
- The BT structure created within BTUF has a similar structure to a BC BT, as discussed in Section 3.3.3, at least warranting an overall condition check of the overarching goal-task if implemented by the user.
- For the sake of robustness during the experimental validation, as discussed in Chapter 4, the used robot skills are dependent on certain blackboard (BB) variables sending out BB variables themselves upon (un)successful execution. E.g., the *reach object* robot skill is dependent on the *free arm* robot skill having executed successfully prior. The *free arm* skill, as the name suggests clears the robot arm and end effector from underneath a table, allowing it sufficient free space to execute the *reach object* skill to reach the goal object for grasping. Without executing the *free arm* skill prior, the *reach object* skill would inevitably slam the end-effector within a table. Thus, to avoid such a collision, the *reach object* skill is dependent on a BB variable that is sent out upon successful execution of the *free arm* skill, circumventing the possibility of said collision occurring. BB variable dependency holds true for all robot skills as used within the conducted experiments. In short, to construct a correct pick and place pipeline using BTUF as was required of participants in the experiments, discussed in 4.5, robot skills had to be sequence in a proper fashion as to avoid possible world collisions. The reasoning for the robot skills being BB variable dependent is to uphold safety by minimizing world collisions.

3.3.14 Conventional Framework

For the purpose of experimental validation as done in Chapter 4, a comparative framework to BTUF has been made, the *Behaviour Tree Creation Framework* (BTCF). BTCF is referred to as a more *conventional* framework and can be regarded as an immensely simplified version of BTUF in terms of available features. It is designed such that it should resemble a more conventional way of creating BTs which is *manual programming*. Aside from it being hugely impractical and time-consuming, it is simply not realistic to ask participants to create a BT from scratch by means of manual programming. Therefore, BTCF makes use of a Text-Based User Interface (TUI). The major difference is that BTCF does not allow incremental adaptation, meaning that the user has to create the entire BT from start to finish in one run. Basic functions such as adding nodes (including control flow nodes, which is not possible within BTUF), visualizing the BT after finishing the construction thereof and executing the completed BT are present. In other words, BTCF allows a human user to create a BT from scratch till finish by placing control flow and leaf nodes wherever desired and allowing BT execution thereafter, albeit no function for node removal is given.

To summarize, the main features of BTCF are:

- Adding actions and/or conditions.
- Adding control flow nodes (sequences and fallbacks)
- Executing the constructed BT (of which a visualisation is given)

It is important to **note** that BTCF is not directly comparable to conventional manual programming as many differences are found in between. However, BTCF can pose as a comparative framework to BTUF considering that, as opposed to BTUF, many of the decisions regarding node placement and overall BT structure are all left in the hands of the user similar to manually programming a BT. BTUF on the other hand, although allowing a lot of human input, most crucially constructs the general structure of the BT upon which entire behaviours, simple or complex, can be built. Both BTUF and BTCF are used within the pilot and final experimental validation setup as discussed in Chapter 4.

Chapter 4

Validation

This chapter discusses the validation or proof-of-concept of the Behaviour Tree Update Framework (BTUF). Both a pilot study and final experiment has been set up and conducted to gather subjective and objective metrics from a small group of participants. This was done in order to draw general conclusions about the initial overall user performance of the inner workings of the proposed framework, whilst also looking at the average rate of acceptance thereof by human operators.

Section 4.1 discusses the setup of the conducted experiment. Section 4.2 discusses the hypotheses and initial expectations. Section 4.3 presents the results thereof, after which an overall conclusion with respect to the experimental validation is drawn in Section 4.4. The results and conclusions are further discussed in Section 4.5. After this section the reader is lead to continue to the last chapter, Chapter 5, which gives an overall summary of the methods applied, the obtained results and the conclusion as taken within this thesis project. Future recommendations given the results of the conducted experiments as discussed in this chapter are given in Chapter 5 as well.

4.1 Experiment Setup

Note: the structure of Sections 4.1 and 4.2 is based on the guide on writing a research proposal by Field (2013), as well as the guide on human subject research for engineers by De Winter and Dodou (2017). The experiment proposal, as accepted by this thesis' supervisor Dr. Jens Kober, is found in Appendix 6. The hand-out given to the participants during the experiments is found in Appendix 6.

4.1.1 Introduction

The conducted experiments look at the performance of the newly proposed BTUF for incrementally updating BTs with a specific structure in order to effectively teach an agent how to perform tasks associated with ADL without having the user require programming knowledge. Performance of the usage of BTUF is compared to a minimalistic substitute framework, BTCF, mimicking a more conventional method of programming BTs.

4.1.2 Method

Participant Selection

Both a pilot study and final experiment have been conducted. The pilot study involved five participants drafted from the Technical University of Delft (TU Delft). The final study focused on participants that were drafted from the employees of this thesis' facilitating robotics company Heemskerk Innovative Technologies (HIT). The final experiment had seven subjects participating. The participants were **not necessarily** required to have knowledge regarding Behaviour Trees and similar high-level robot policy representation, but the presence thereof was regarded as helpful. Neither was knowledge in the field of robotics a necessity for participation in this experiment. Considering the nature of HIT's

employees however, most of the final experiments participants had some degree (from a medium to very high level) of background knowledge in robotics engineering in general. The pilot study participants on average had no background knowledge or experience with respect to robotics. As the only goal of the experiments was to provide a proof-of-work of the proposed framework in the form of some objective and subjective metrics, having a small test group for the final experiment has been regarded as sufficient for drawing meaningful conclusions towards that end as found in Section 4.4.

The median participant age for both the pilot and final experiments was 25-30. It has been assumed that age and the ability to perceive and understand BT structures do not have a negative correlation with one another. Moreover, in the case that there is a negative correlation (as might be proven in future studies or not) it does not affect the overall outcome of this experiment, as we were merely trying to find a general first-impression of the usability, performance and subjective human system acceptance of BTUF over more *conventional* methods as represented by BTCF. Participants were given a hand-out with the necessary information with respect to their participation in the study, which includes a form of written consent, prior to their actual participation. Participants have been let known that they were able to forfeit their participation at any given time before, during and after the experiment. Personal information, aside from age, gender and experience regarding BTs and robotics overall, were not asked nor published as they have no academic relevance in the case of this experiment. The participant handout is found in Appendix 6.

Materials

Materials used for this experiment were:

- A printed or digital hand-out used for briefing the participants as well as for asking their written consent.
- A printed or digital hand-out showcasing the tasks that had to be completed by the participant using either one of the two frameworks as described in Sections 3.3.1.2 and 3.3.14.3 respectively.
- A personal computer on which the participant had to perform the experiment tasks.
- A printed or digital hand-out on which to draw a BT and for answering some general subjective questions regarding the experiment including a Van der Laan scale, which has been used as a simple assessment of system acceptance.

Experimental Design

Considering that this study used a small group of participants, a within-subject design has been used for the experimental setup. In the case of a within-subject designed experiment, each participant tests all of the experimental conditions. In the case of this particular study, every participant had to teach a robotic agent (TIAGO by Pal Robotics) within a simulated environment how to perform a simple pick-and-place task using BTUF and BTCF in randomized order. Before working within the simulation, participants were asked to draw out a BT by hand in order to visualize what they approximately expected to create using either of the two frameworks based on some simple background knowledge regarding BTs and their general working principles. This was done in order to probe the participants' general understanding of BTs. It can also infer something about the difference between the time to completion of manually drawing a BT by hand versus doing so using one of the two presented frameworks. This inference, however, has not been made, which is also discussed in Section 4.5. When using BTUF, participants were not specifically let known what a supposedly *good* tree would look like, but were made aware of the working principles of a BT and the functions of the used BT nodes. Thus, participants were expected to perform the experiments mostly based on trial-and-error and logical reasoning in order to achieve successful completion. This was done so similarly for when participants used BTCF. With this similar manner of briefing in advance of using either one of the two frameworks, it was hypothesized that overall performance, as measured by the metrics as found in Section 4.1.2, of BTUF would be higher in comparison to BTCF. The rest of the initial hypotheses and expectations are found in Section 4.2 of which the answers are discussed in Section

4.4. After construction of a BT capable of executing the goal behaviour during the first phase of the experiment, participants were expected to adapt the BT to account for a novel unforeseen instance on the task in the second phase (e.g., accounting for the sudden presence of an obstacle that hinders successful completion of the initial pick and place task). Because a within-subject design has been used, each participant served as their own control (De Winter and Dodou, 2017).

Metrics

The objective metrics recorded during the pilot study and the final experiments are:

- *Time to Completion (TC)* - The amount of time necessary for the participant to construct a BT capable of solving the given task goal, taken over all runs from the start to finish thereof.
- *Runs to Completion (RC)* - The amount of runs necessary for the participant to construct a BT capable of solving the given task goal, of which a run is defined as the time in which the user is operating a framework from initialization to preemption thereof or until the successful execution of a constructed BT.
- *Total BT nodes in final tree structure (TBTN)* - The amount of control flow, condition and leaf nodes present in the final tree structure capable of solving the given task goal.
- *Amount of actions taken to completion (AC)* - The amount of actions taken from start to finish, of which an action is determined as an input by the user that results in the addition or removal of one or a multitude of control flow, condition and/or leaf nodes. As an example, if the user gives an input to add a condition node to a BT, the TUI might prompt the question as to what amount of conditions should be added at once given that one input. The total amounts of nodes added given that single function input is in that case disregarded and the amount of actions taken by the user is always considered to be equal to one user action. I.e., irrespective to the amount of nodes added with a singular function input, is the singular function input counted as such.

The objective results for the pilot study and final experiments are found in Section 4.3.1.1 and Section 4.3.2.1 respectively. The subjective metrics recorded during the pilot study and the final experiments are given by a Van der Laan scale, also depicted in the participant handout form as found in Appendix 6. The Van der Laan scale consists of nine different subjective variables that are measured or ranked in a similar fashion as a Likert scale, being: *Useful-Useless*, *Pleasant-Unpleasant*, *Bad-Good*, *Nice-Annoying*, *Effective-Superfluous*, *Irritating-Likeable*, *Assisting-Worthless* and *Raising Alertness-Sleep-inducing*. The subjective results are found in Section 4.3.1.2 and Section 4.3.2.2 for the pilot study and final experiment respectively. Lastly, participant feedback has been recorded and is found in Appendix 6.

Procedure

The experiments were conducted at Heemskerk Innovative Technologies (HIT). The participants were briefed by means of a printed or digital hand-out explaining the outline and purpose of the experiment, as found in Appendix 6. The briefing included a description of the task the participants had to perform and how much time they had to do so. All participants were asked to fill out a form of written consent after the briefing finished, before actually commencing and partaking in the experiment. The participants were informed that they could cancel their participation at any given moment before, during and after the experiment. The participants were also let known that they hold the right to retract any of the experimental data gathered from their respective participation within the experiments. The participants identity as well as other personal information regarded as non-vital for the academic relevance of the experimental outcome will only be known to the researchers in question and will not be published to the public. Only after the briefing was done, the participant had been fully made aware of the experimental procedure and the participant's consent had been obtained, did the experiment start. The experiment itself had a duration of approximately 40 minutes. The total experiment duration varied per person, as some took 25 minutes while others needed approximately

70 minutes. The subjects have been asked to fill out a Van der Laan system acceptance scale for both BTUF and BTCF, giving the researcher(s) insight in some subjective metrics such as usefulness and desirability of using a framework such as BTUF over more *conventional* methods. This Van der Laan scale is found in Appendix 6. The subjective results thereof are found in Section 4.3.1.1. Additional comments and findings by the subjects were asked for as well, in the case that the subjects desired to provide these. These comments are found in Appendix 6. After a subject finished the experiment task, if so desired by the subject, he/she was debriefed in a broader context about the experimental goals, which could initially influence the potential outcome of the study if it were done so before commencing the experiment. At last, after the experiments had been conducted for the specific individual in question, the subject was made aware of the completion of the study. All participants, if they so desire, will be notified in the future if any of the experimental results are published or made public. All recorded data has been digitally stored by the researcher.

4.2 Hypotheses & Initial Expectations

The reason for conducting the experiments is for the experimental validation or proof-of-concept of the proposed framework. Several hypotheses were made prior to conducting the experiments. These hypotheses can be summed up as:

1. Participants will have a faster time to completion using BTUF.
2. Participants will require less runs to completion using BTUF.
3. Participants will require less actions to completion using BTUF.
4. Participants will require less nodes in the final BT structure using BTUF.
5. Participants will prefer the framework wherein an intermediate output is given that visually represents the BT under construction during the creation/adaptation process of the tree, being BTUF.
6. Participants will prefer the framework wherein the BT structure is mostly given to the user. I.e., the framework that automatically generates the overall BT structure will most likely be preferred by human users, being BTUF.

All of these hypotheses were sought to be proved or disproved given the results of the pilot and final experiments. The initial expectation was that BTUF would exceed BTCF in all aspects, except for the amount of BT nodes present in the final BT structure. The reasoning for the latter statement was that since BTUF prescribes and automatically generates most of the BT structure, a lot of extra nodes will be added to BTs created using BTUF, whereas users might not account for such a similar and possibly more complex structure when utilizing BTCF. The next section, Section 4.3, discusses the experimental results. These results are then used to prove or refute the initial hypotheses and expectations in Section 4.4, which draws an overall conclusion towards the validation of the own proposed framework. Lastly an overall discussion of the experimental validation is given in Section 4.5.

4.3 Experimental Results

4.3.1 Preliminary Study Results

Prior to the actual experiment, a pilot study was conducted with five participants. Four participants were students drafted from the Technical University of Delft who had some background experience/-knowledge in robotics engineering and/or BTs. One participant is a long-time employee of HIT with an extensive background in robotics engineering. These pilot studies were meant to check whether the proposed experimental setup was reliant enough for gathering the desired objective and subjective metrics as described earlier in Section 4.3. Moreover, the pilot studies were meant to indicate any changes that had to be made before moving on to the actual experimental tests. Data was recorded for all five participants. Data recorded for two of the five participants were omitted as they did not fully finish or succeed their participation in the pilot study.

The pilot study tasks, which differ slightly from the tasks that were given in the final iteration of the experiment, involved asking the participants to create an executable BT capable of solving the given task goal using both of the presented frameworks respectively. Afterwards, three subtasks were given that involved the need for adapting pre-made BTs that were unable to execute properly, using the BTUF framework. These subtasks were mostly meant to check the ability of participants to read and interpret (the given) BTs and whether they were able to logically argue what adaptive measures had to be taken in order to alter the BT in such a fashion as to complete the given sub-tasks. One of the subtasks included adapting a BT capable of achieving the initial task goal (*pick object* → *inspect object* → *place object*) to also account for a possible obstacle that has to be removed if detected. It has to be **noted** that these subtasks were solvable using a minimum of one action, which was also let known prior to the participants.

Objective Metrics

Several preliminary findings regarding the objective metrics are displayed in Table 4.1. The description of the main and subtasks is given in the participants handout, as found in Appendix 6.

Framework	TC [s]	RC	TBTN	AC
BTUF	501	1.3	13.7	9.7
BTCF	438	3.3	8.7	20

Table 4.1: Objective results for the main task retrieved from three participants in total. All numbers have been rounded of to one decimal point.

Looking at the preliminary objective results it can be seen that on average it takes 57 seconds longer to construct an executable BT using BTUF in comparison to BTCF. The final BT constructed with BTUF have on average 5 nodes extra. On the contrary only a single run is necessary to complete the task goal using BTUF on average, whereas BTCF requires approximately 3. Moreover, the amount of total actions taken from start to finish was twice as much when using BTCF in comparison to when using BTUF, being 20 and 10 actions respectively.

As a **preliminary conclusion** it can be said that BTUF takes slightly longer to operate and constructs larger BTs, but at the same time requires only one run and less actions to create an executable BT that is capable of solving the given task goal.

BTUF	TC [s]	RC	AC
Subtask 1	94	1	2.7
Subtask 2	245	1	1
Subtask 3	33	1	1

Table 4.2: Objective results for the subtasks retrieved from three participants in total.

The subtasks, as found in the participant handout as seen in Appendix 6, involved the following three tasks in respective order:

1. A BT is shown that is missing a specific action within the required sequence of actions to successfully execute the pick-and-place pipeline as represented by the specific BT. This subtask is designed to inform participants of the node insertion function as found within BTUF.
2. A BT is shown that accounts for the detection and removal of an obstacle alongside the base pick-and-place behaviour. Participants have to remove a superfluous backward chained branch in order to successfully execute the tree. This subtask is designed to inform participants of a *proper* way to account for possible obstacles in the pick-and-place behavioural pipeline.
3. A BT is shown that is missing a condition check, that the participants have to add. Without said condition check, the entire pick-and-place pipeline will re-execute in full every time the BT is rerun. This subtask is designed to inform participants about condition-action pairs and the advantage of the added robustness of using such pairs.

From table 4.2 it can be seen that as intended, all participants were able to solve the three given subtasks within one run. However, for subtask 1 the average AC was 3. A clear reasoning for this has not been found and is therefore not further formulated.

Subjective Metrics

Subjective metrics were recorded using a Van der Laan scale, depicted in the participant handout form as found in Appendix 6. The preliminary results regarding the subjective rate of acceptance are depicted in Figures 4.1 and 4.2 for BTUF and BTCF respectively. Participants were neutral regarding the ease of understanding of BTUF, but did prefer this framework over BTCF.

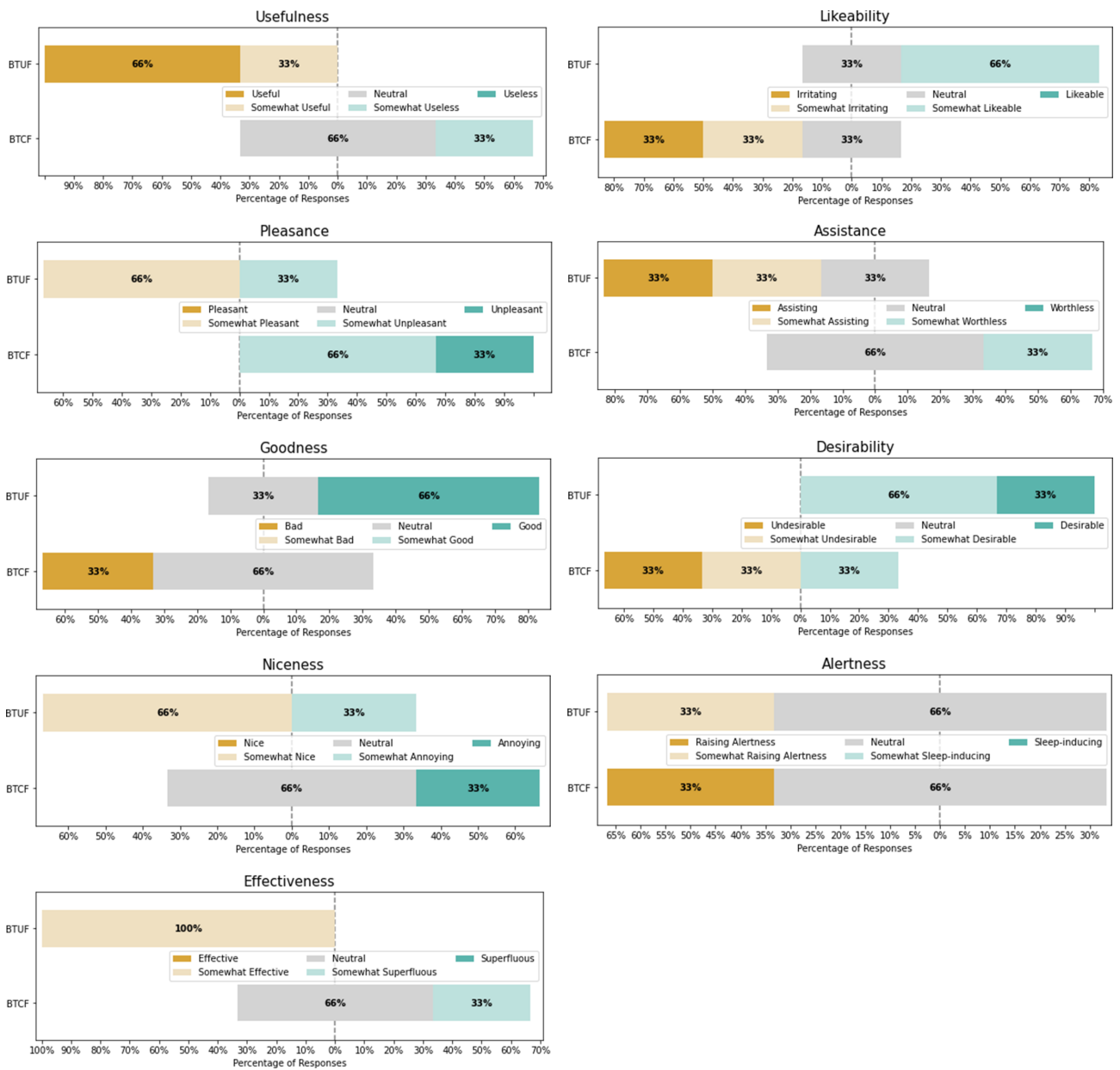


Figure 4.1: Mean preliminary Van der Laan scale results concerning system acceptance for BTUF and BTCF as retrieved from three pilot study participants.

Overall it can be seen that BTUF is subjectively viewed as positive. The subjective metrics imply that the framework is viewed as useful, good, effective, likeable, assisting and desirable.

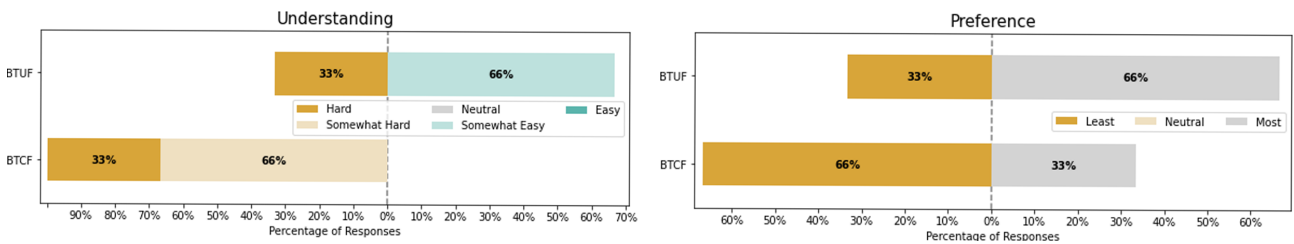


Figure 4.2: Mean preliminary subjective results concerning preference and understanding of BTUF and BTCF as retrieved from three pilot study participants.

On average BTCF was subjectively viewed in a more negative light in comparison to BTUF. The subjective metrics imply that the framework is unpleasant, irritating and undesirable. However, it does seem to be regarded as nice and raising alertness. The latter metrics seems to be contradicting

to what the results of the former metrics imply. The reason for this discrepancy is most likely the fact that these results were gathered from only three participants and more data is needed to have a better indication of system acceptance. The reader is thus referred to the subjective results as found from the final experiments as discussed in Section 4.3.2. The ease of understanding of BTUF was considered to be somewhat hard by the participants, alongside the fact that participants were neutral about whether they did or did not like this framework. Overall it was found that two out of three participants favored BTUF over BTCF. One participant favored BTCF as it gives the user full freedom over the creation of the BT structure.

4.3.2 Final Experiment Results

Seven subjects participated in the final experiments. The data from one of the participants was omitted due to not finishing the full-length of the experiment. Both the objective and subjective metrics recorded during the final experiments are the same as those recorded during the pilot study, as discussed in Section 4.3. Two notable changes were made for the final experiments in comparison to the prior conducted pilot experiments. Firstly, participants were given an additional task-phase where a (by the human participant) constructed BT had to be adapted to account for a novel instance that involved the removal of an obstacle before executing the pick-and-place pipeline. Secondly, participants were given training before using either framework in the form of a demonstration by the researcher. This was done in order to minimize a learning effect observed during the pilot. This learning effect involved participants demonstrating a significant overall improvement in TC as a result of prolonged operation of either framework.

Objective Metrics

The results for the objective metrics is displayed in Table 4.3.

BTUF	TC [s]	RC	TBTN	AC
Phase 1	520	1	13.8	9.3
Phase 2	296	1	17	1.8
BTCF	TC [s]	RC	TBTN	AC
Phase 1	553	2.7	8.7	15.8
Phase 2	279	2	9.6	11

Table 4.3: Final objective results retrieved from 6 participants. Phase 1 refers to creating a BT capable of solving the initial task goal. Phase 2 refers to adapting or re-creating a BT to account for a novel instance of the initial given task.

Looking at the results, we find that the total operation time to successfully perform phase 1 and 2 of task 1 does not differ much between both frameworks, being 814 seconds and 832 seconds for BTUF and BTUF respectively. A bigger difference can be found in time per phase per framework. Using BTUF the adaptation process takes significantly less time compared to the construction of the initial pick-and-place BT. For BTUF, participants required less time to create a BT to account for the novel instance. Using BTUF this had to be done from scratch. This finding is attributed to the fact that during phase 1 it becomes evident to participants what type of BT passes phase 1 successfully. This realisation leads to an improvement in TC during phase 2. Most simply, the addition of an action to remove the obstacle to the initial BT successfully completes the task goal, albeit not robustly so. Although the TC is similar for both frameworks, there is a clear difference found in the other metrics. Participants completed phase 1 and 2 on average in a single run using BTUF, whereas for BTUF multiple runs were required to construct a BT capable of achieving the task goal. Participants were given intermediate feedback in the form of a visual image of the BT they were constructing at every step of the construction process when using BTUF. Moreover, BTUF allows for undoing previously added nodes/branches via its removal function. On the other hand, BTUF neither showed the intermediary tree during construction nor did it allow for undoing previous taken actions or removing placed nodes. This major difference between the two frameworks resulted in participants having to rerun BTUF and to restart from scratch in the case that a mistake was made or if the, or so they

thought, final BT structure did not execute successfully or as expected. This functional difference also results in the amount of actions taken (AC) from start to completion, to be significantly higher for both phases when using BTCF compared to BTUF. Lastly, it can be seen that the total nodes in the final executable BT structure is higher when utilizing BTUF in comparison to BTCF. This is attributed to the fact that BTUF pre-constructs essential parts of the BT structure. This is done in order to *guide* the user to easily construct an executable BT without having to focus too much about proper structuring. BTCF does not automatically account for a proper BT structure during operation. This leaves users with all constructional freedom with respect to the BT structure. This resulted in most participants being able to create a minimal tree using BTCF, that while although most likely was not robust, correctly achieved the given task goal. The more minimal a BT is, the more computationally efficient it is. However, it has to be **noted** that the given tasks did not ask the human to add or account for condition checking, leaving the robustness of the constructed BTs to be desired for. This occurrence is further discussed in Section 4.5.

BTUF	TC [s]	RC	AC
Subtask 1	57	1	1.1
Subtask 2	75	1	1.1
Subtask 3	66	1	1

Table 4.4: Objective results for the subtasks retrieved from three participants in total.

With respect to Table 4.4 it can be seen that, as initially intended from the subtasks, all participants were able to solve the three given subtasks within one run using only one action on average. However, it was seen that the intended solution sometimes was not given but an alternative, yet correctly executable tree, was proposed by the participant instead. An example would be that instead of removing a particular node, another was added to a certain location that yielded the same desired effect for solving the sub-task goal, albeit making some nodes or branches a bit superfluous due to node duplication. This further specifies the necessity for future research about optimal BT structures, of which the amount of academic research is currently lacking. This research gap has been mentioned prior in Section 3.3.3 and is proposed as future work in Section 5.2.

All in all it can be **concluded** that the total operation time when utilizing either framework is similar. The most notable differences are found within the amount of runs to completion, the total BT nodes in the final structure and the total amount of taken actions until completion. BTUF takes only a single run and less actions overall to create an executable BT for solving the task goal. On the other hand, BTCF can create smaller but also more complex BTs that are coherent to the users wildest imagination.

Subjective Metrics

The mean of the subjective results regarding the system acceptance of BTUF and BTCF using a Van der Laan scale taken over six participants in total is displayed in Figures 4.3 and 4.4 respectively.

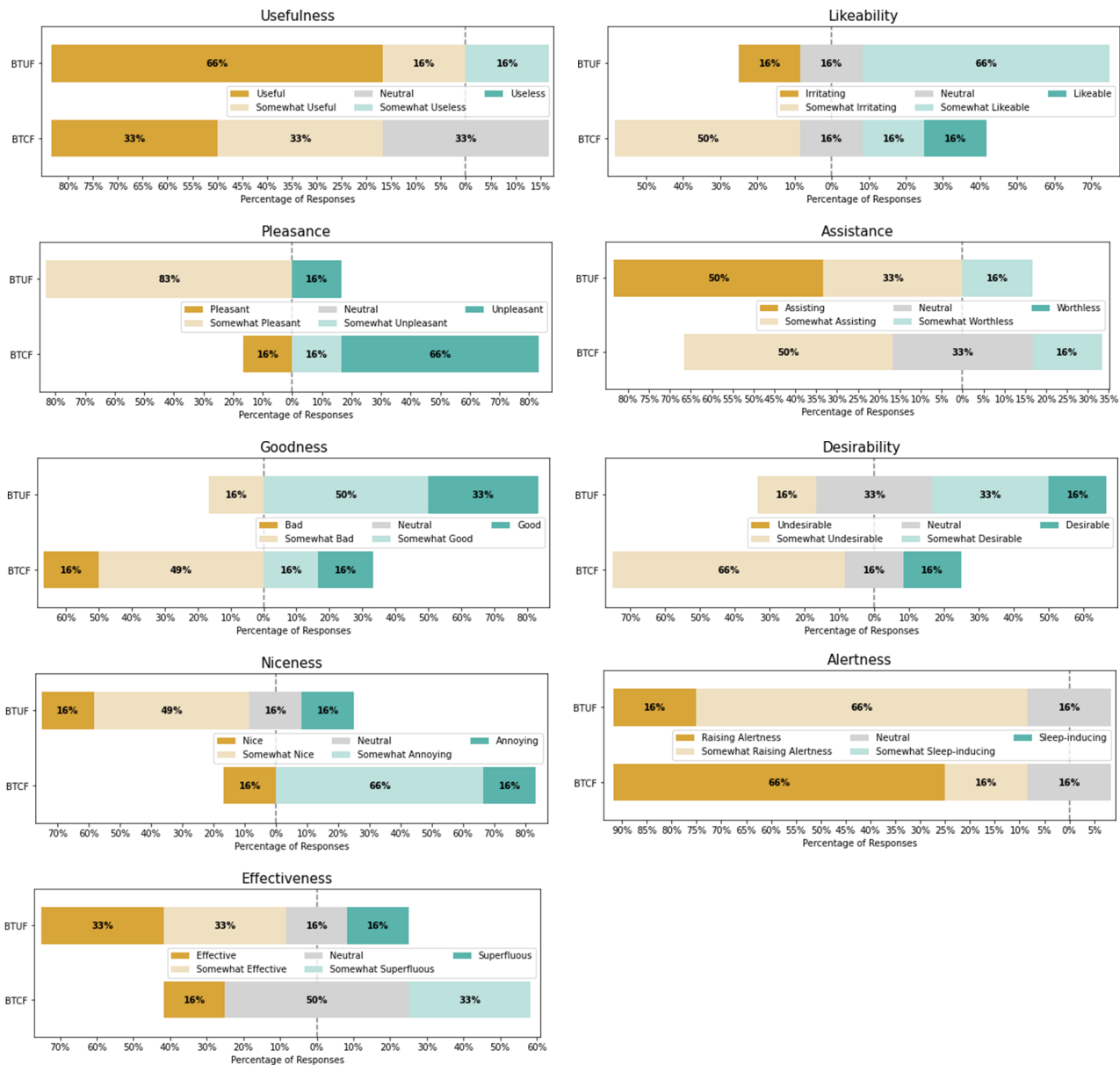


Figure 4.3: Average subjective Van der Laan scale results for BTUF and BTCF, taken over 6 participants.

From Figure 4.3 we find that participants view BTUF positively overall. On average it is considered to be useful, pleasant, good, nice, effective, assisting, desirable and that it raises alertness. From Figure 4.4 we find that participants are more neutral about BTCF. On average it is considered to be useful, but unpleasant and annoying to work with. However, working with BTCF does raise alertness. From feedback taken from one of the participants, as transcribed in Appendix 6, we however find that this raising of alertness was not per se considered to be a good thing. It was suggested that due to the nature of BTCF, one should have more attention and focus during the operation thereof compared to when using BTUF. This would subsequently result in the human operator not being able to perform longer operation time due to fatigue and loss of attention over time.

Given the subjective results we can **conclude** that on average participants seemed to prefer using BTUF, considered the usage thereof more favorable and to be somewhat easy. BTCF was preferred the least and was considered to be somewhat hard to operate, while having a higher mental workload due to the need for increased focus during the operation thereof.

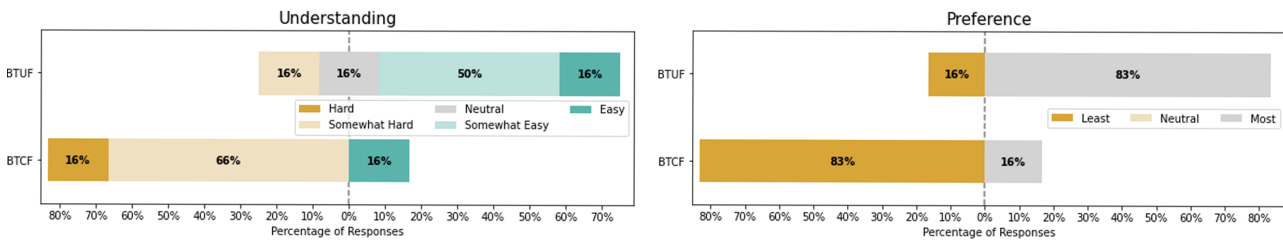


Figure 4.4: Average final subjective results concerning preference and understanding of BTUF and BTCF, taken over 6 participants.

4.4 Overall Conclusion Towards the Validation of the Own Proposed Method

In this section, the experimental results as discussed prior in Section 4.3 are used to answer the initial hypotheses and expectations as found in Section 4.2.

- **Hypothesis 1:** Participants will have a faster time to completion using BTUF.
 - From preliminary findings it was found that it takes 64 seconds longer to operate BTUF compared to BTCF for creating a base behaviour (phase 1). On the contrary, from the final findings it was found that on average participants required 33 seconds less for creating the initial BT when using BTUF. This differs greatly from the initial findings of the pilot study. A difference of 97 seconds between the preliminary and final findings was found. This difference might be credited to the fact that participants were given a training/demonstration phase in the final experiments. The addition of such a phase might have resulted in the difference in TC between the two frameworks to shift more in favor of BTUF than it did prior without, as was the case during the pilot experiments. For phase 2, the novel instance phase, it was found that on average participants seemed to be able to complete the given task 17 seconds faster on average when using BTUF. All in all, it was found that on average human participants were only 16 seconds faster in operating BTUF compared to BTCF. This is not a significantly large difference considering that the entire task took on average 817 and 831 seconds to complete using BTUF and BTCF respectively.
- **Hypothesis 2:** Participants will require less runs to completion using BTUF.
 - One of the differences between BTUF and BTCF, as found from the preliminary findings, was that the average participant could create the initial BT in a single run while three runs were needed in total when doing so using BTCF. It can be seen that this observation holds true for both the pilot and final experiment. From the final experiment results we find that phase 2 of the main task, to account for a novel instance, was on average completed in a single run when using BTUF. It took an average of 2 runs to complete the same task when using BTCF. All in all, it can be concluded that hypothesis 2 is therefore correct. Participants only require one single run to create an initial behaviour, as well as a single run to adapt said initial behaviour to account for a novel instance. This stands in comparison to BTCF which requires on average 3 runs to create the base behaviour and 2 runs extra to accommodate for a novel instance thereon as found from the final results.
- **Hypothesis 3:** Participants will require less actions to completion using BTUF.
 - From the preliminary findings, as shown in Table 4.1, we see that the AC while using BTUF equals 10. This is equal to half the AC required to complete the given task using BTCF, which stands at 20. From the final finding, as found in back in table 4.3, we see that the average AC while using BTUF equals 7 actions. This is less in comparison to the AC when using BTCF for creating the initial BT. Moreover, when accounting for the novel instance BTUF only takes 2 actions on average to adapt the the initial BT. Compared to the on average 11 actions needed while utilizing BTCF, it can be said that this is a large difference.

This finding is easily understood by the fact that BTUF allows for the saving, loading and further adaptation of prior created BTs. This functionality allows the user to simply load a prior built tree, add a backward chained branch to account for possible obstacles to be removed all the while retaining the original behaviour created earlier. BTCF, being more unforgiving, requires the user to recreate the tree from scratch, which in turn clearly explains the higher AC when using BTCF. All in all it can be concluded that hypothesis 3, which states that participants will require less AC using BTUF is therefore correct.

- **Hypothesis 4:** Participants will require less nodes in the final BT structure using BTCF.
 - We find from the preliminary results in Table 4.1 that the TBTN equals 14 on average when created with BTUF and 9 when created with BTCF. From the final results in Table 4.3 we observe the same for phase 1. The TBTN for BTs created for phase 2 using BTUF count 17 nodes on average, whilst only 10 nodes are counted on average for the BTs created using BTCF. Several remarks can be made about this observation. In general, participants were not asked to account for robustness. Robustness could be partially accounted for by adding condition-leaf pairs. By adding a condition check before attempting to execute a specific action, the BT will always check to see whether that corresponding action can actually execute or not before even attempting to do so. The task goals can be satisfied by simply adding sequences of required actions, as participant do not have to account for condition checks. However, it was found from most of the participants' BTs created with BTUF that a specific backward chained branch was used to account for the removal of the obstacle. This means that most participants used a condition-action pair in the form of a BC branch to check and account for possible obstacles, before attempting to execute the pick-and-place pipeline associated with the base behaviour. This type of BT executes successfully in both the situation that an obstacle is or is not detected. On the other hand, most trees constructed by participants using BTCF consist mostly out of a simple action sequence, wherein the *remove obstacle* skill was inserted into the pick-and-place action sequence. This was all done without the addition of any condition check. Robustness of the BTs could not be checked due to setup of the experiment and the overall formulation of the task goals, as discussed in Section 4.5. This means that despite having a higher amount of nodes, most BTs created using BTUF that account for obstacle removal using a condition-action pair in the form of a BC branch are therefore more robust than their BT counterparts created using BTCF. The reader is also referred to Section 4.5 for a further discussion regarding this exact issue with respect to the robustness of BTs created during the pilot and final experiments. Matters aside, it can be concluded that hypothesis 4, which states that participants require less TBTN using BTCF is therefore correct given the obtained results.
- **Hypothesis 5:** Participants will prefer the framework wherein an intermediate output is given that visually represents the BT under construction during the creation/adaptation process of the tree, which is BTUF.
 - From both the pilot study and final experiments we find that on average participants preferred the BTUF framework over the BTCF framework. From verbal and written feedback as transcribed in Appendix 6 we find that one of the main reasons is the addition of intermediary visual feedback during the creation process of the BTs when using BTUF. It can be concluded that this particular hypothesis, which states that participant prefer the framework wherein intermediate visual output is given, holds true.
- **Hypothesis 6:** Participants will prefer the framework wherein the BT structure is mostly given to the user. I.e., the framework that pre-constructs parts of the BT structure will be preferred by human users, which is BTUF.
 - Repeating what has been said in the answer to hypothesis 5, we found from preliminary and final results that the majority of the participants had a preference for BTUF over BTCF. Another reason for this preference is that BTUF automatically constructs the most essential

parts of the BT structure. All participants were mostly unfamiliar with or had no hands-on experience in creating BTs prior to the experiments. Most participants seemed to appreciate the help that BTUF provides in creating BT structures, given that the available knowledge of the participants were limited to what was provided during the experiment. Comments and feedback from participants have been gathered and are transcribed in Appendix 6. From these comments we find that most participants appreciate this particular feature. It has to be **noted** that the opposite belief has also been found in some participants. The reader is referred to Section 4.5 for a further discussion regarding the overall acceptance of this particular feature that pre-constructs most of the BT structure. All in all, it can be concluded that the majority of participants favor BTUF, which automatically generates the overall BT structure.

Overall it can be concluded that, from both the preliminary and final data, that all six hypotheses have been proven to be correct. In short it can be said that BTUF is received fairly positive with human users, while participants are more neutral or slightly negative towards BTCF. Thus, the use of the proposed *Behaviour Tree Update Framework* has hereby been validated. The reader is referred to Section 5.2 in Chapter 5 which discusses possible future work to further solidify this claim.

4.5 Discussion

A pilot study with five participants was conducted prior to the final experiments. Out of five participants, three participants provided usable subjective and objective results. These results are discussed in Sections 4.3.1.1 and 4.3.1.2. The final study was conducted with seven participants out of which only one participant's data was omitted due to not fully completing the study by dropping out prematurely. These results are discussed in Sections 4.3.2.1 and 4.3.2.2.

A major difference between the pilot study and the final experiments is the addition of a 5 to 10 minute demonstration of each framework as done by the researcher during the final experiments. With the initial absence of a training/demonstration phase during the pilot study, it was found that there was a strong learning effect. This learning effect has been mentioned prior in Section 4.3.2. As the experiment focused mostly on the difference between the two different frameworks for incremental adaptation of BTs based on the objective performance and subjective acceptance by human users, it was decided to add a training/demonstration phase in the hope to quell this learning effect. However, from the preliminary and final objective results it can be seen that the values for RC, TBTN and AC were mostly similar between the pilot and final study. The TC for task 1 to create the initial BT was lower on average during the pilot study in comparison to the final experiment. This difference can be due to the fact that there is a discrepancy in the amount of participants during the pilot and the final study, being three and six respectively. Another possible reason could be that because a training/demonstration was given that showcased all of the framework's specific functionalities, people sought to utilize more of these functions. During the pilot study the participants mostly approached the task goal by relying heavily on trial-and-error, which might possibly attribute to a shorter TC. However, all of this is just speculation and therefore it is heavily suggested that future work includes redoing the aforementioned experiments using a larger pool of participants in order to find more conclusive results regarding this difference in TC between the pilot and final study group of participants that did not and did receive training/demonstration respectively.

Participants were initially only required to create a BT to execute a simple pick-and-place pipeline during the pilot study. This pick-and-place pipeline involves picking, inspecting and placing an object located on an elevated surface or table. Thereafter, subjects were presented with a prefabricated BT, capable of solving said initial task and were asked to account for a novel instance thereupon. This novel instance being a suddenly spawned obstacle that had to be removed upon detection before successfully being able to execute the previous learnt behaviour. In the final experiment setup this was changed such that participants now first had to successfully create an executable BT capable of solving the initial task goal, after which it had to be adapted to account for the novel instance. I.e., no

prefabricated tree was given for the adaptation step, but instead the BT created by the participants themselves were reused. This held true for BTUF. When using BTCF, participants were asked to create the BT yet again from scratch, however this time taking into account a possible obstacle that had to be removed. It has to be **noted** that a huge advantage persisted in accounting for the novel task instance while utilizing BTUF due to it having the additional function to save and load constructed trees. This crucial function resulted in participants only having to perform a minimal amount of 2 actions to account for the novel instance, which involved loading the previous built tree and inserting a backward chained branch to account for the obstacle removal into it. The BC branch was then to be placed such that before attempting to execute the pick-and-place pipeline the obstacle would be removed if detected. This occurrence was also hypothesized prior to conducting the pilot and final experiments in hypothesis 3 as discussed in Section 4.4. Again, as BTCF does not have a feature for saving and loading BTs, participants were asked to create a new BT from scratch to account for the novel instance. This lack of a saving and loading function resulted in a far higher AC on average when using BTCF compared to the AC when operating BTUF.

Looking at these changes made to the final experiments given the feedback from the pilot study and looking at the overall experimental setup in general, several final remarks and overall points of improvements can be summed up as following:

- Both the pilot and final experiments focused on identifying system acceptance of a proposed framework that allows for incremental adaptation of reusable BTs for performing robotic manipulation tasks associated with ADL. The experiment was set up in such a fashion as to compare one framework that allows incremental adaptation with continuous visual feedback and input fault tolerance in the form of undoing wrongfully taken actions by the user, with another framework that does not support such intermediate visual feedback nor has input fault tolerance. However, except for the fact that there was an inherent difference in the process of constructing a tree, where one was incrementally adapted over time with an intermediary visual output at every step of the process versus one where it had to be constructed in one go from start to finish, it was found from user preference as well as participant feedback and comments, as transcribed in Appendix 6 that a lot of preference was given to the framework with the most *easily operable* user-interface (UI). In short, the complexity of the UI infers a lot about the participants preference towards said framework. Some participants considered having less input options to be more favorable in comparison to having more. Others favored it exactly the opposite way around. In other words, a lot of the recorded subjective results are heavily influence by a difference in UI per framework. Considering that the intention of the experiments was to focus on the back-end of the two presented frameworks, i.e., the working principle or algorithms behind the curtains, rather than the UI itself, it is proposed that a future study has to be conducted where the UI of both frameworks are kept as similar as possible to one another, to counter this recorded phenomenon. In that scenario the obtained data will be more relevant when trying to draw conclusions regarding the difference in working principles behind both frameworks, instead of it being heavily subjugated under the participants' UI/front-end experience.
- A change made to the final experiment setup was the addition of a training/demonstration phase before using either framework. During the pilot study, participants were only given some general background knowledge about BTs and the overall context of the experiment. Trial-and-error was required for participants to successfully work with either framework as background knowledge and experience was lacking for all participants. However, as the intuitiveness of working with the system and the speed per participant for grasping and understanding how to operate either framework differed greatly from one individual to the other, it was opted to add a training/demonstration phase to the final study. This training/demonstration phase was intended to provide a somewhat equal amount of background knowledge to all of the participants, in the hope that the total experiment duration could be diminished and the background knowledge/-experience could be somewhat normalized between all of the participants. However, as BTs were still a foreign concept to nearly all of the participants, it is suggested that longer periods

of training/demonstration should be given if future experiments with either framework is to be conducted. A remark from one of the final experiment participants, as transcribed in Appendix 6, stated that if he was given a couple of hours to spend with either framework, that he would definitely be able to operate either framework, both the one he did and did not favor, in a quick and accurate manner.

- Robustness of a BT can for example be referred to as the ability to cope with situations that would result in a, partially or fully, failed execution thereof. I.e., a BT can be considered to be more robust if the tree has the ability to account for situations where successful execution thereof can not be safeguarded. A way to increase robustness in BTs is backward chaining, as discussed in Section 2.6.1. Simply said, checking a condition prior to executing a robot action can increase the robustness of successfully executing said action. In the way that both the pilot and final experiments were set up and the manner in how the task goals were formulated, participants were tasked with the goal to simply construct and execute a sequence of actions. These actions were simply to pick, inspect and place an object. After successful execution thereof, people were also asked to account for a novel instance wherein an obstacle had to be removed if detected. The way that the first task goal was worded, it was sufficient to construct a BT that consists simply out of a simple sequence placed at the root of which the children are all of the required robot actions. The robot skills are constructed in such a way that, if placed in the correct order, these actions will always execute successfully. However, this resulted in the fact that there was no necessity to add conditions to account for possible mishaps. Therefore, although some participants did attempt to make the constructed BTs more robust by implementing backward chaining or just by pairing conditions with their associated actions, most did not. Therefore, a clear statement regarding the difference in robustness between the two presented frameworks, BTUF and BTCF, can not be made at this point of time. For instance, a BT constructed using BTUF that albeit having some pre-constructed BT structure only has actions added manually to said structure and a BT constructed using BTCF in the form as described earlier that consists of only a sequence of actions, will nearly amount to the same tree if executed. The only conclusion then could be that the BT created using BTUF automatically implements a return statement in the form of the `Idler` action as seen in Figure 3.9, which reset the robot pose irrelevant of whether the added actions execute successfully or not and will therefore be a bit more robust in comparison to a BT created using BTCF where one has to manually account for this to behaviour to occur and is not automatically added to the BT structure. All in all, it is proposed that if future experiments are conducted, participants should be asked to account for possible erroneous executions, which should then prompt and encourage them to add condition checks within the tree. Only then, as participant will keep in mind to account for condition checks, can a valid conclusion be made on the differences between the BTs made with either framework with respect to the robustness thereof.
- During both the pilot study and final experiments, participants were asked in advance to draw a BT by hand given that they were provided with background knowledge about the working principles of BTs while providing context about the to be performed task. Aside from the purpose of the drawing being that participants had to think about a proper BT structure that would be required to achieve the specified task goal, the time to draw such a BT by hand was also recorded. It was the intention to then compare the time needed to construct a tree by hand versus the time needed to construct a tree using either of the presented frameworks. However, some flaws can be indicated in the manner as to how this comparison was set up and why it finally was opted to not make this comparison. Firstly, most of the participants at this stage of the experiment still did not fully understand how a BT properly executes, as they were only given a demonstration of either framework after completing their drawing. This resulted in some of the drawn BTs simply not being able to execute if they were programmed as such. Secondly, the frameworks were presented in a random fashion per participant. It was found that some participants who did seem to grasp the theory behind BTs fairly quickly felt limited into constructing the exact BTs using BTUF in accordance to what they had drawn-up prior

in the drawing phase. These participants were left confused when using BTUF as most of the BT structure is automatically generated therein and the freedom to construct their envisioned BTUF structures that came forth from their understanding of how a proper BT structure should look like, was simply not possible. It has to be **noted** that the researcher did make clear that participants should have been open minded when using either framework and that the BT structures constructed via either framework might differ from their initial (drawn) expectations. However, to avoid this confusion a different setup is proposed for future experiments. In the case that the researcher wants to compare the necessary time to draw a *correct* BT by hand versus the time necessary to construct the same BT using a computer, a drawing phase should be added in the future experimental validation of both frameworks respectively. The demonstration/training phase per framework should be given before both the drawing phase and *programming* phase, to make sure the participant understand what type of BT structures can possibly be created when utilizing each specific framework. In other words, an example of such a setup would be that context is given regarding the correct approach in creating BTs for each framework in advance respectively, after which participants are expected to uphold said approach in drawing such a BT by hand prior to doing so within the framework. The beforehand drawn BT is in that case more comparable to the one constructed using that particular framework. With this setup a more relevant conclusion can be drawn about constructing a BT by hand versus doing so within a framework that either does and does not pre-construct the majority of the BT structure.

- The Behaviour Tree Update Framework proposes two methods for creating BTs from scratch via human input requiring little to no programming background. The first method is to initially construct and incrementally adapt a BT from scratch to create a base behaviour that can be expanded to account for novel instances. This method of adaptation can eventually create larger and more complex behaviours using smaller and more simple behaviours as a starting point or as building blocks. These behaviours can then, over time, be aggregated and enlarged to form complex behavioural task-plans capable of performing ADL and possible alternatives thereupon. This method of incremental adaptation is implemented by constructing and compiling the BT at every step of the construction process wherein nodes are either added or removed of which feedback is given in the form of a visual representation of the constructed tree. Moreover, the ability to continuously add, as well as to remove nodes alongside the ability to save and load all created BTs also facilitates the reusability, adaptability and expandability of these constructed BTs. The second method that allows for creation of BTs without the need for the human operator to have a programming background is the addition of a UI that is operable with ease via simple computer-keyboard inputs in combination with BTUF automatically generating most of the BT structure during the creation/adaptation process. In other words, users should not have to bother too much with the BT structure as the basis of the tree is generated by BTUF and the same structure is mostly upheld during the further construction of BTs. As said prior, it has been found that most of the subjective metrics were heavily influenced by the difference in UI between both of the tested frameworks, BTUF and BTCF. Most participants seemed to prefer having a framework wherein the structure is mostly guided by the program itself. Some seemed to dislike this feature due to the fact that it prohibits the freedom of structuring the BT fully custom to one's desires.

To test all of the implemented methods it would have been better to separate them in the experimental test setup. For instance, to check what the addition would be of a framework that automatically generates most of the BT structure and how it would be perceived by human operators, another test setup can be proposed. Within this setup two frameworks are compared to one another where the only difference is that one predefines a tree structure only to be followed, whereas the other does not. Simply said, we would compare two structures that should be nearly identical to one another in terms of functions, UI, as well as adaptive capabilities in the form of loading and saving and removing added nodes and branches. The only difference would be that one gives full control over the construction of the BT structure, where the other does

not and automatically generates most of it. In that case it is hypothesized that users who have extensive knowledge about constructing BTs will prefer a framework that allows full freedom over the construction thereof. People who rather prefer a lighter mental workload or have less background knowledge with respect to proper BT structures might prefer the framework that pre-constructs most of the BT structure automatically.

- Regarding the amount of participants drafted for both the pilot and the final experiments it can be said that having relevant data from three and six participants respectively, is on the low side. It can be said that in order to draw more conclusive results regarding the experimental validation of the proposed framework and its counterpart, more data has to be collected from a larger pool of participants. However, this was not done as the experimental validation part of this thesis project was kept minimal due to time constraints with respect to the total duration of the thesis project. It was also opted to keep the experiments in-house, which limited the possible maximum amount of participants. Albeit kept minimal, the experimental validation as done in this thesis can be regarded as a valid method for finding the proof-of-concept of BTUF. As mentioned prior, the final experiment was conducted with participants drafted only from within HIT. Future research should draft participants from a group of people with a larger difference in gender, age, engineering experience and educational or professional background to get a more conclusive and broader insight in the validity and acceptance of the proposed framework.

It is **noted** that although preference was given to BTUF, some participants appreciated and favored the freedom of creating custom BT structures while using BTCF. Therefore it is suggested that, if a future version would be made, that the next iteration of BTUF allows the user to choose whether he/she would like the framework to automatically generate the BT structures, that although limits freedom can quickly create executable and adaptable behaviour trees or that he/she would like to fully create the BT structure him/herself by placing all control flow, leaf and conditions nodes manually. This suggestion, alongside some others, is further discussed in Section 5.2 of the concluding chapter of this thesis, Chapter 5.

Overall it can be said that the conducted experiments, although giving some insight in the objective performance and subjective acceptance of BTUF in comparison to BTCF, should be redone with a larger group of participants using a different setup as proposed in the bullet points above, if more representative data is to be collected. Although kept minimal, the experiments as conducted during this thesis project can be regarded as sufficient and the conclusion made from the gathered results can be seen as relevant. The overall results with respect to the preference, subjective acceptance and better objective performance of BTUF as compared to BTCF can be viewed as a proof-of-concept of the in this thesis proposed *Behaviour Tree Update Framework*.

Chapter 5

Concluding Chapter

This final chapter concludes this thesis project report, starting with an overall summary, continuing with a section that proposes future work for the possible continuation of the academic work as done in this thesis and finally ending with some concluding remarks. The overall summary of this thesis is found in Section 5.1, possible future work is discussed in Section 5.2 and the final concluding remarks are given in Section 5.3.

5.1 Summary

The world population is ever aging. With an increasing shortage of medical personnel, the introduction and use healthcare robotics is proposed. Some improvements have to be made. Most importantly, robotic agents should be able to work autonomously without the need for constant monitoring by a remote human operator. Robots need to be programmed or taught locally how to account for different situations when executing tasks. Thus, robots should ideally have a form of high-level logic or reasoning when executing tasks assisting humans with Activities of Daily Living (ADL). To more easily integrate robotics into domestic and healthcare environment, the use of Learning from Demonstration methods is proposed to allow those without an extensive background in programming to create robot behaviours associated with ADL.

BTs are a visual high-level representation method for robot behaviours. This thesis proposes a framework for the initial construction and continuous incremental adaptation of BTs via human input for creating complex robot behaviours associated with ADL without requiring the human operator to have an extensive background in programming. These BTs are modular, reusable and human-interpretable. Parts of or entire BTs can be reused to expand or merge these into larger trees. BTs are interchangeable between different agents given the low-level scripts that are embedded within them are compatible with one robotic system and the other. BTs have a simple yet distinct structure that is continuous throughout the entirety of the tree, which is easily understood by humans. All in all, BTs are suitable for use in the construction of robot behaviours by both experienced and inexperienced human operators.

The proposed Behaviour Tree Framework (BTUF) implements several methods. Conventional BTs are constructed via manual programming and compiled at run-time after the full construction thereof is finished. Operating BTUF does not require programming knowledge as all of its functionalities are embedded within a Text-Based User Interface (TUI), that is easily operable via simple keyboard inputs. Some background knowledge with respect to how BTs are read/executed is helpful however. Therefore a user-manual can be provided to future operators, similar as to the one found in the participant handout in Appendix 6. From experiments, as conducted during this thesis, it was found that solely relying on trial-and-error and having some hands-on experience was sufficient for making the *correct* operation of BTUF clear over time. BTUF allows for incremental adaptation of BTs as it compiles the tree at every step of the construction process wherein nodes are either added or removed

from the tree. This allows for a visual output of the intermediate tree, which notifies the human user of their current progress, as well as allows for intermediate execution of the BT to check whether the desired behaviour is properly executed or not. The BT structure is mostly automatically generated by BTUF. This allows the user to focus mostly on what types of conditions checks and actions he/she wants to add. Thus, the human operator is able to construct trees using for example the following simple analogy of reasoning that reads: *"First I want to do A, to then proceed with B to finally execute C after checking whether I can do C at all."* The ability to save and load previously constructed BTs to use them for further adaptation, or merging with novel constructed BTs allows for incremental expansion of simple to complex behaviours over time. Lastly, the use of BTs allows for human readability and insight into constructed trees which can infer the embedded behaviours of the trees when executed, without having to do so in practicality.

The methods as implemented in BTUF have all been discussed in theory, but have also been realised into a physical framework that has been validated using an experimental test setup. Human subject experiments were conducted both with a pilot and final test setup, with 5 and 7 participants per setup respectively, totalling at 12 subjects. Objective and subjective data was recorded to compare the overall performance of the newly proposed BTUF versus another framework that poses as a substitute for the conventional method of manually programming BTs, being BTCF. The major differences between the two frameworks is that BTCF does not compile the BT at every step of the construction process, which leads to a lack of intermediate visual feedback of the constructed tree and therefore neither allows for the intermediate execution thereof. Moreover, BTCF does not automatically generate the most important parts of the tree structure leaving the human operator with all freedom in constructing the tree structure themselves. However, this increase in freedom in constructing the BT structure does require a larger degree of understanding of how to properly structure BTs, which raises the user requirements with respect to background knowledge and experience for using this particular framework. Lastly, some important functions that facilitate modularity and reusability of constructed BTs are absent in BTCF, such as saving and loading constructed trees and undoing previous made changes.

From the objective results it was found that overall participants were able to construct a suitable BT for executing the given pick-and-place task in a single run. At least multiple runs were necessary to construct a BT with a similar behaviour in BTCF. Moreover, the total amount of actions required to construct the final tree structure using BTUF was around half the amount of actions necessary to do so using BTCF. The adaptation process for expanding upon the base behaviour to account for a novel instance in the form of an to-be-removed obstacle required only one run and a minimum of 2 actions on average when using BTUF, as opposed to BTCF which required yet again more actions as well as more runs to completion on average.

From the subjective results gathered over all of the conducted experiments, it was found that BTUF was preferred by the majority of the participants over BTCF. BTUF was also perceived to be somewhat easy to operate, while BTCF was regarded as more difficult to use.

Several possible reasons for the objective and subjective outcomes have been raised and discussed extensively in Section 4.5 of Chapter 4 and will not be detailed out further in this summary.

In short, from the experiments it has been concluded that overall BTUF seems to be accepted by the human public and is seen as an useful tool for creating robot behaviours. BTCF, meanwhile, has some charms of its own being that the TUI is more simple and that a select few of the participants preferred the freedom in constructing custom BT structures. Future work is proposed by the writer in Section 5.2 that discusses possible merging of functionalities of BTUF and BTCF to further improve the Behaviour Tree Update Framework as proposed and implemented in this thesis, alongside some other propositions to improve the experimental test-setup and future iterations of BTUF.

In conclusion, this thesis proposes and validates a framework for the initial construction and continuous incremental adaptation of BTs via human input to create complex robot behaviours associated with executing ADL for use in, but not only limited to, both domestic and healthcare robotic agents. Future work is discussed and proposed in Section 5.2 to further advance the development of the proposed framework to one day implement a future iteration thereof in physical robotic agents to be used in the healthcare industry and domestic environments to tackle the increasing shortages in medical personnel with the ever-increasing world population.

5.2 Future Work

This section discusses possible future work with respect to the further experimental validation of and possible improvements on the in this thesis proposed *Behaviour Tree Update Framework*. Some points have been raised in Section 4.5. All of the propositions for possible improvements have been summarized and are listed below:

- **Proposition 1:** Retest the proposed framework with a new experimental setup and a larger group of participants.
 - As discussed extensively in Section 4.5, some points of improvement are suggested for future work to gather more conclusive results with respect to the framework as proposed in this thesis, aggregated from the made conclusions in Section 4.4. To summarize, when using comparative frameworks in order to gather objective and subjective results, a new test setup should account for all of the possible variables in between the presented frameworks. Lastly, the pilot and final experiments have only been conducted with 5 and 7 participants respectively, totalling to an amount of 12 subjects. To draw more meaningful conclusions that give a better indication regarding the overall performance and general system acceptance of the proposed framework, a larger pool of participants has to be drafted for future experiments.
- **Proposition 2:** A future iteration of BTUF should allow for both guidance and full freedom with respect to the creation of the BT structure.
 - During the experimental validation it was found that there were different sentiments with respect to having automatised generation of the fundamental structure of a BT as implemented within BTUF. As can be seen from the experimental results in Section 4.3 of Chapter 4 it was found that most participants of both the pilot and final experiments, being 66% and 83% respectively, seemed to prefer BTUF. A major reason being the help that BTUF provides with constructing the basis of the BT structure. A select few however did not like this *limited* approach in creating BTs and rather preferred having to study up on how to create proper BT structures and create them as customized to their own desires as possible. As BTUF promises on creating robot behaviours with little to no experience, this approach of having full freedom over the BT structure is counter-intuitive towards that promise as it requires more background knowledge and experience with respect to the manual construction/programming of BT structures. However, a future iteration of BTUF could give users both the option to either have BTUF automatically generate the essential parts of the BT structure, while the other option allows the user to have all freedom in constructing the tree if he/she so desires and has the sufficient background knowledge in creating the entire structure themselves to do so successfully. Moreover, when automatically generating the base structure of BTs while using BTUF, the framework should also allow for targeting individual nodes as found within BC branches or parallel branches. The current implementation of BTUF does not allow so, as discussed prior in Section 3.3.3. This addition will improve the possible complexity of BTs created within BTUF, but will require more background knowledge of the user. As a conclusion, this proposition is made for broadening the user audience of BTUF, so that it facilitates both users with and without programming experience.

- **Proposition 3:** A future iteration of BTUF should implement other projects as proposed by and currently being worked on at HIT and by the academic field in general.
 - BTUF has been built in such a way that it contains an action and condition library. These actions can be written in different formats by different people in different languages. In other words, if an action or executable script works for the used robotic system, then BTUF can be used to parse these actions inside of a BT in order to create (complex) robotic behaviours. In the past as well as of current, HIT has been working on low-level robot motion control policies, such as PMPs and DMPs which are robot motion policy paradigms learnt from demonstration via teleoperation. Simply said, the action dictionary of BTUF can be created and/or expanded using a LfD teleoperation method encoding demonstrated human motions via e.g., an Omni haptic manipulator into a movement primitive. This could be done manually or automatically. The addition of encoding the robotskills using LfD via teleoperation and saving these to the action dictionary of BTUF will most likely prove to be a great asset in further realising the desire to teach a robot human-like low-level behaviours and high-level reasoning. Moreover, HIT is working on a Natural Language Processing (NLP) module for having interactive human-robot interaction via speech. BTUF could be utilized for embedding this particular NLP module as a part of different behaviours within a BT. The other way around could also be true, where the NLP module makes use of BT structuring as proposed with BTUF to make a high-level task plan of the conversational flow between human and robot. Although this thesis has been facilitated by TU Delft and HIT and is thus property of both parties resulting in the code not being open-source at the time of writing, the academic community in general should, if desired, expand BTUF with integration of novel methods and ideas in order to further the advancement of using BTs for the incremental adaptation of reusable BTs in robotics.
- **Proposition 4:** Expand BTUF so that created BTs can be exchanged online and can be applied on multiple robotic systems.
 - BTUF can be applied on all sorts of different robotic platforms dependent on what low-level scripts they have embedded. With the addition of an online platform or a similar cloud type storage, BTs could be exchanged to share created behaviours, which are most likely applicable to similar or compatible robot systems, with one another. This type of sharing empowers the ideology of the reusability of BTs and can drastically slow down production time of BTs in the case that a user can find their desired behaviour online, without having to create these themselves first. Obviously, apart from only sharing the BTs themselves people will most ideally have to embed the scripts for the low-level actions alongside them. This can all be done for example via Github or similar platforms. All in all, this proposition is to increase the accessibility to knowledge and behaviours by giving BTUF an even more open-source nature (considering that this academic work, aside from the code repository, is mostly open to the public as well).

All in all it can be **concluded** that the proposed *Behaviour Tree Update Framework* forms a good basis for future research, as proposed in this section, and for the further progression towards the realization of a user-friendly framework to be utilized by people with all types of backgrounds for creating and adapting simple and complex robot behaviours to advance the autonomous integration of robotics in daily domestic life as well as in the healthcare industry.

5.3 Concluding Remarks

This thesis report concludes the work as done by the researcher and writer thereof, Rudy De-Xin de Lange, for the degree of Ir. at the Technical University of Delft, the Netherlands. This thesis proposes the Behaviour Tree Update Framework for incremental adaptation of Behaviour Trees through human input for implementation in Learning from Demonstration applications for healthcare robotics. Special thanks is given to all of those that have supported me throughout this last phase of my life as a student. Although the code created during this thesis project is not open source and is property of both the TU Delft and HIT, I would like to see future researchers carry on my work to realize the dream of creating an user-friendly framework that enables people of all ages and backgrounds to create simple and complex robot behaviours to help the advancement of robotic integration within the domestic and healthcare field. Thanks to the reader for making it this far in the report.

Chapter 6

Appendices

Appendix I: Participant Feedback & Comments

Within this appendix recorded feedback and comments retrieved before, during and after the pilot, as well as the the final, experiments are compiled/transcribed in the cases that they were provided by the respective participants. These comments and feedback have been considered when writing the discussion of this thesis' experimental validation chapter, found in section 4.5.

Note that the remarks are all transcriptions and thus are written using a grammatical structure that represent the direct personal viewpoint of the participant themselves.

Remarks w.r.t. BTUF

- "It seems very effective for adepts, but not intuitive or insightful for beginners" - *Pilot Study Participant 1*
 - "After performing the subtasks my opinion w.r.t. BTUF changed. The subtasks helped my understanding of the basics a bit more. The frameworks saves time, effort and possible mistakes by pre-filling parts of the tree."
- "Since I didn't have any reference knowledge, the first framework was confusing at first." - *Pilot Study Participant 2*
 - "However, after using the second framework I appreciated the automatic generation of the main parts. It would save a lot of time, because you can skip the base part when building the tree." - *Pilot Study Participant 2*
- "Intuitive and quick to understand." - *Pilot Study Participant 3*
- "Insightful, works well and the framework can be learned quickly. You do have to understand the logic behind the framework and behaviour trees a bit. I would need additional time to become fully skilled in operating the framework, however I think that can be done relatively quickly. Because the constructed trees can be easily adapted/changed, it can decrease my level of alertness when operating the framework. However, I consider this to be a plus." - *Final Study Participant 2*
- "The condition nodes and their working are confusing. Does the condition imply that you already have performed a certain action, or is it a prerequisite question you ask before commencing an action. Also, I do not fully agree that a structure is partially pre-generated in this framework as other structures might be more suitable in my opinion." - *Final Study Participant 4*
- "This framework worked way better (compared to BTCF). However, there are a lot of inputs that repeat themselves (e.g., the function to append an action or to insert it). It would be better to have less amounts of inputs to improve the operation of the framework." - *Final Study Participant 6*
- "Annoying interface, not allowing to directly undo a previous action, forcing you to go through a lot of steps to remove a mistake. I do not like the fact that the structure is pre-made and I rather want to construct it myself." - *Final Study Participant 7*

Remarks w.r.t. BTCF

- "For a beginner, this framework seems to promote understanding of the BT mechanics more, but more tedious/time consuming compared to the other framework (BTUF)" - *Pilot Study Participant 1*
- "It is easy to lose track of where I am in the tree, because I could not see it when building. In addition it is a bit annoying to build the base part of the tree." - *Pilot Study Participant 2*
- "Hard to visualize and understand." - *Pilot Study Participant 3*
- "The framework is usable, however you do need to write out the trees on paper beforehand or you continuously have to logically think about the steps that have to be taken and how the tree continuously looks like during construction. This makes it very work intensive, which would lead to not being able to operate the framework for extended periods of time. I consider this framework to be hard to work with, as well as not being very pleasant to operate." - *Final Study Participant 2*
- "To work via terminal inputs to construct a tree is not useful. The power of a BT lays in their visual representation, which is lacking in this framework (during the intermediary steps of the construction process)" - *Final Study Participant 3*
- "It is unclear what a fallback supposedly does if used in step one of the tree construction procedure. It would make more sense to add a condition first instead of a fallback. It is very hard to keep track of the tree that is being constructed and to clearly understand what it is going to do when it is finally executed." - *Final Study Participant 6*
- "Lots of freedom to build structures." - *Final Study Participant 7*

Appendix II: Participant Handout & Form of Written Consent

Incremental Adaptation of Behaviour Trees

For Applications in Learning from Demonstration Frameworks

Researcher: R.D. de Lange

April 11, 2022

1 Introduction

Total duration: ≈ 30-60 minutes

With an ever ageing society, the demand for healthcare improvements increases on a continuous basis. As a response to this trend, Heemskerk Innovative Technologies (HIT) is currently developing a healthcare robot to support the sick and elderly in performing everyday tasks or *Activities of Daily Living* (ADLs).

A Behaviour Tree (BT) is a high-level policy representation, which in this case is used to represent a complex robot task synonymous to ADLs. Within this study you, the participant, are asked to create a BT structure for pick and placing an object using a simulated healthcare robot using two different frameworks. At the end of the experiment, it is asked from you to fill out a questionnaire regarding your perceived user experience of both frameworks.

Important Notes

- You are free to forfeit your participation at any given time before, during and after the study.
- You are free to refuse answering any of the researcher's questions.
- You are free to ask any question that will not influence the outcome of the study.
- The information that you provide will be anonymised and used for academic research purposes only.
- Your personal identity will not be disclosed to the public nor internally and will only be known by the researcher in question.
- In the case you want to reach out to the researcher in question for what any reason in regards to your participation or the results of this study, please reach out to: rudydelange@tudelft.nl

2 Form of Informed Consent

2.1 Taking part in the study

I have read and understood the study information dated April 11, 2022, or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.

Yes No

I consent voluntarily to be a participant in this study and understand that I can refuse to answer any of the researcher's questions. I understand that I can withdraw from the study at any time, without having to give a reason.

Yes No

2.2 Risks associated with participating in the study

I understand that taking part in the study involves the risk of my personal information being known by the researcher(s) in question. However, I also do know that my personal information will not be publicly published in order to safeguard my identity.

Yes No

2.3 Use of the information of the study

I understand that the information I provide will be used for academic research purposes only and sensitive information towards my own personal identity will not be disclosed to the public and is therefore, if published, held anonymously.

Yes No

2.4 Future use and reuse of the information by others

I give permission (on April 11, 2022) to be archived in the TU Delft repository so that my anonymised research data retrieved from my participation in this particular study can be used for future research and learning purposes.

Yes No

Name Participant:

Signature:

Date:

Name Researcher:

Signature:

Date:

I, the researcher, have accurately read out the information sheet to the potential participant and, to the best of my ability, ensured that the participant understands to what they are freely consenting.

3 Experiment

3.1 Context

As previously mentioned in the introduction 1, you are asked to create a Behaviour Tree (BT) to teach a simulated robot how to pick and place an object, using two different frameworks: Framework 1 & 2. Whichever framework you are using first and second is randomized and will only be disclosed at the end of the experiment when the final hand-out is given to you.

The first half of the practical part of the experiment is two-fold (repeated for both frameworks each) and consists of three phases each

- In the first phase you are asked to draw a BT out by hand that, in your understanding given the context as provided in this experiment, should be able to reach the goal as stated in the task description.
- In the second phase you are asked to make the robot pick an object placed on a table, to then inspect it and finally to place said object back on the table.
- Phase Three of the experiment will start once the first phase is successfully completed and involves accounting for a novel instance of the original task as presented and tackled in phase one.

In the second half of the practical part of the experiment you are asked to solved three small tasks, that should be solvable in a matter of minutes each.

3.2 Background

A BT sends *ticks* or an execution signal starting from the root node (the top most node) down the entire tree. In other words, a Behaviour Tree (BT) reads from top-to-bottom and from left-to-right as depicted in figure 1. The nodes either have one of the two outputs **Running** or **Failure**, indicated either with the colours **green** or **red** as seen in 1 and in the BTUF & BTUF-L framework during the experiment.

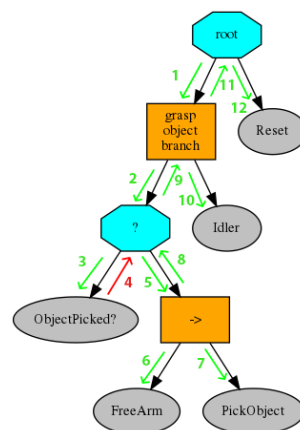


Figure 1: A BT is ticked from top-to-bottom and from left-to-right. A green arrow indicates a **Success** status for the ticked node. The red arrow indicates the node output to be **Failure**.

Four different node types are identified in the BTs as constructed with either BTUF or BTUF-L, as found in figure 2.



Figure 2: (From Left to Right) **Sequence**: Succeeds if all children N succeed. **Fallback**: Succeeds if one child M of N children succeeds. **Condition**: Checks if a certain condition or world state has been achieved. **Leaf**: or **action** node, represents an executable file or (partial) behaviour/skill.

3.3 Framework Functions

You are asked to work with two different frameworks for constructing BTs in order to teach a health-care robots how to perform ADLs. This section shortly discusses all of the available functions found within either framework. Do bear in mind that the functionality per framework differs, meaning that not all of the described functions are found back in both of the frameworks respectively. I.e. one framework might have different functions as opposed to the other.

3.3.1 Adding/Removing Actions & Conditions

Actions/Leaves correlate to robot skills, or executable skills that the robot can perform. Conditions are checks that look at given world variables. The use of conditions is e.g. to check whether a robotskill has been executed successfully or not. As an example, `ObjectPlaced?` returns *Success* if the robot successfully executed the action `PlaceObject` or `PlaceObjectAway` prior.

Actions and conditions can both be either added or removed. *Appending an action/condition* places the nodes in a sequential or consecutive order. *Inserting an action/condition* at a specific location or index allows the user to manually place said action/condition at a different location within (a specific part) of the tree. **Note**: the researcher can shed more light on the previous statement verbally.

Actions	Conditions
<code>FreeArmFromUnderneathTable</code>	<code>ObstacleRemoved?</code>
<code>ReachObjectWithArm</code>	<code>ObjectReachedWithArm?</code>
<code>GraspObjectWithGripper</code>	<code>ObjectInspectedWhileLiftedFromTable?</code>
<code>LiftObjectFromTableWithArm</code>	<code>ObjectGraspedWithGripper?</code>
<code>PlaceObjectOnTable</code>	<code>ObjectPlacedOnTable?</code>
<code>PlaceObjectFurtherAwayOnTable</code>	<code>ArmBlockedByTable?</code>
<code>InspectObjectWhileLiftedFromTable</code>	-
<code>RemoveObstacleWithGripper</code>	-
<code>Idler</code>	-

Also note: that the conditions and actions given above are given in a randomized order, of which not all of them are always available to manually choose from within either of the frameworks.

3.3.2 Adding Control Flow Nodes

Control flow nodes, are as the name implies nodes that control the flow or manner in which BTs are traversed/ticked/executed. The control flow nodes available to the user are either fallbacks or sequences as explained prior in figure 2. When having to place control flow nodes yourself, bear in mind the differences in the working mechanism: e.g. fallbacks can be used if only one child has to succeed. A sequence can be used if all of the children *need* to succeed. Another use of a fallback is

to chain a condition with an associated action to increase robustness, which brings us to the next section, section 3.3.3.

3.3.3 Backwards Chaining

Backwards chaining is a method for making actions/leafs more robust. An action is substituted for a certain structure that consists of at least a fallback, with two children being 1) a fallback 2) a (sequence of) action(s). When executing a backwards chained tree, the condition is checked first. Upon success the sequence of actions will not be executed. Upon failure, it will. This structure allows for condition checking prior to the execution of a sequence of actions that will satisfy the the condition if not done so initially.

3.3.4 Saving & Loading

All of the constructed trees can be saved & loaded for later use. When loading a tree, the user can continue to adapt the tree as and if desired.

3.3.5 Adding/Removing A Parallel Tree

A parallel tree can be created by loading in a saved tree. This parallel tree will be appended to the root of the current tree and thus forms a separate branch or behaviour that can be either an alternative to the original behaviour or a whole new behaviour on its own. These parallel trees can be removed in turn as well. **Note** that these parallel trees do *not* have to be utilized within this experiment.

3.4 Task #1

Phase 1: - Draw a BT that you believe is suitable for the task as described in *Phase 2 (Simulation)*.

Note: You can change/extend your drawing when starting with phase 2

Phase 2 (Simulation): - Pick the object on the table → Inspect the object → Place the object back on the table.

Note: avoid collisions while doing so!

Phase 3 (Simulation): - Repeat phase 1 & 2, but now account for the novel instance.

Note: if possible, you can re-use earlier saved BTs
(only holds true when using BTUF)

Important: you have only successfully completed the task goal for either phase once the researcher confirms so.

3.5 Task #2

You are presented with three small tasks that require adaptation of pre-constructed BTs within one of the two earlier shown and used frameworks.

Note: These tasks are solvable using a minimum of one action. However you are not required to do so!

Subtask 1: - The shown BT is not able to correctly execute a pick and place behaviour. Add/remove nodes in order to successfully execute said behaviour.

Subtask 2: - The shown BT is not able to remove the obstacle. Add/remove nodes in order to successfully remove the obstacle as well as pick up the object!

Subtask 3: - The shown BT will be executed in full again during every consecutive run after an initially successful first run. Add/remove nodes so that the BT only executes the entire motion pipeline in full during the first run .

4 Experiment Evaluation Form

4.1 Framework 1

Age:

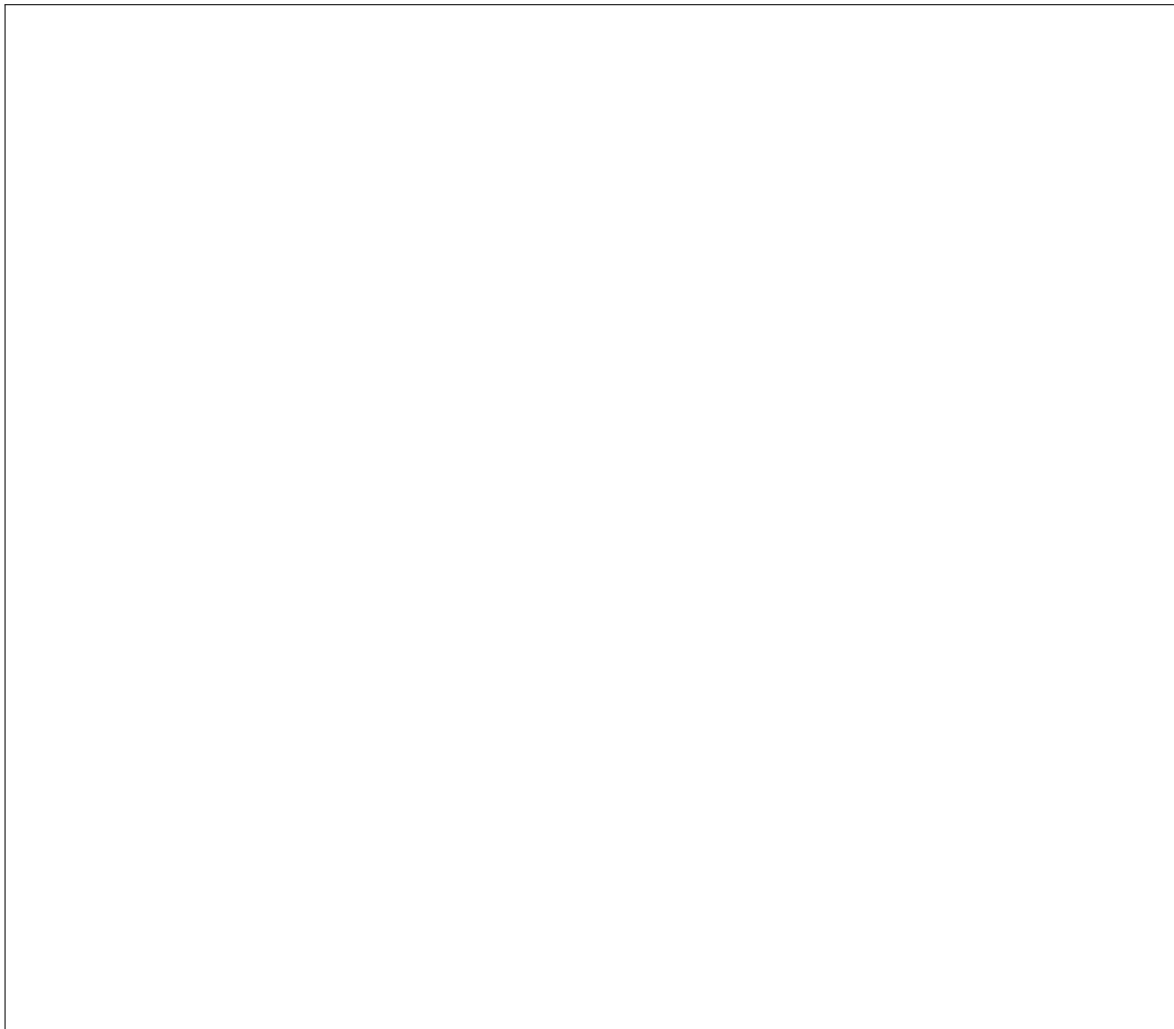
Gender:

Male Female

Years of Experience (w.r.t. Robotics):

-1 1 2 3 4 5 6 7 8 9+

Please draw a BT for picking an object → to inspect it → to place it back:



How easy could you understand and operate the framework?

Hard Somewhat Hard Neutral Somewhat Easy Easy

Rate your preference for this framework:

Least Neutral Most

Please fill out the following Van der Laan scale:

Useful	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Useless
Pleasant	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Unpleasant
Bad	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Good
Nice	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Annoying
Effective	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Superfluous
Irritating	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Likeable
Assisting	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Worthless
Undesirable	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Desirable
Raising Alertness	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sleep-inducing

Additional Comments (Not mandatory):

4.2 Framework 2

Age:

Gender:

Male Female

Years of Experience (w.r.t. Robotics):

-1 1 2 3 4 5 6 7 8 9+

Please draw a BT for picking an object → to inspect it → to place it back:



How easy could you understand and operate the framework?

Hard Somewhat Hard Neutral Somewhat Easy Easy

Rate your preference for this framework:

Least Neutral Most

Please fill out the following Van der Laan scale:

Useful	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Useless
Pleasant	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Unpleasant
Bad	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Good
Nice	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Annoying
Effective	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Superfluous
Irritating	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Likeable
Assisting	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Worthless
Undesirable	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Desirable
Raising Alertness	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Sleep-inducing

Additional Comments (Not mandatory):

4.3 Question Sheet Additional Subtasks

Did your opinion change of framework X after performing the subtasks?

Yes No

If yes, in what way did your opinion change?

Appendix III: Experimental Validation Proposal

Human Experiment Proposal #2

Incremental Adaptation of Behaviour Trees

For Applications in Learning from Demonstration Frameworks

R.D. de Lange, D. Karageorgos, J. Kober

April 11, 2022

Abstract

This human experiment serves as a means to an end for the experimental validation/proof-of-work of the author’s proposed method, as discussed in de Lange (2022), to incrementally adapt a Behaviour Tree (BT) for applications in Learning from Demonstration (LfD) frameworks [2]. The case or scenario as considered in this experiment concerns a healthcare robot, designed for assisting elderly and medical patients in performing Activities of Daily Living (ADLs), that has to be taught by a human observer how to perform such behaviours. Accounting for novel instances such as removing an *unforeseen* obstacle is done over time via incremental adaptation upon first encounter of such instances.

In short, human participants are asked to incrementally adapt a BT using the proposed *Behaviour Tree Update Framework* (BTUF) to teach a healthcare robot how to perform a single pick and place task. The use of BTUF is compared to another framework that loosely resembles *conventional manual programming* methods. Performance and usability of the framework are measured using both objective and subjective metrics.

1 Introduction

Note: the structure of the experimental set-up for this particular study is based on Field’s (2013) guide on writing a research proposal as well as the guide on human subject research for engineers by de Winter & Dodou (2017) [3][4].

A Behaviour Tree (BT) is a high-level policy representation that shares resemblance to Decision Trees. A Decision Tree (DT) is a popular supervised machine learning (ML) algorithm that is used for regression and classification analysis. BTs, which share a similar structure with the addition of some node types expanding the perceived benefits of their usage, can be used to e.g. represent complex robot manipulation tasks. One of the benefits of using BTs for high-level representations of complex robot skills is their ability to be easily perceived and understood by a human observer (white-box) in comparison to other ML methods or robot policy representations such as neural networks or reinforcement learning algorithms (black-box). Moreover in comparison to their direct high-level policy representation competitor, Hierarchical Finite State Machines (HFSSMs), we find that BTs are more suitable for learning purposes as BT structures remain constant throughout an update process and are perceived to be more modular, more easily expanded upon and less prone to becoming cluttered when enlarged over time.

Concerning the most suitable BT structures for learning or adaptation purposes and the incremental expansion thereof, there has been little to no academic publication to what is perceived

to be most desirable or efficient. This experiment compares the performance of a newly proposed framework for incrementally updating BTs with a specific structure in order to effectively teach an agent how to perform tasks associated with Activities of Daily Living (ADLs) without having the user require programming knowledge. Performance of the usage of BTUF is compared to a minimalistic substitute framework mimicking a more conventional method of programming BTs.

1.1 Node Types

The node types used within BTUF are found in figure 1 below.



Figure 1: (From Left to Right) **Sequence**: Succeeds if all children N succeed. **Fallback**: Succeeds if one child M of N children succeeds. **Condition**: Checks if a certain condition or world state has been achieved. **Leaf**: or **action** node, represents an executable file or (partial) behaviour/skill.

Note that other node types such as parallel nodes (of which all children N are ticked at the same time instance) and decorators (which change the output of a node given a custom function) exist, but will not be further discussed as these are not used within BTUF.

1.2 Behaviour Tree Structures

Three different types of BT structures can be identified as:

- **Deep** - This structure is mostly used when creating backwards chained behaviour trees. Backwards chaining is a goal-solving method by working backwards from the goal state to the initial state. In the case of backwards chained BTs, a failed condition can be substituted with a particular BT branch structure that achieves the prior failed condition. This is mostly done by substituting a failed condition, with a branch consisting of a fallback, of which the children are from left to right 1) the initial condition 2) a singular action or a sequence of actions meant to achieve the same goal condition as child 1) of the overarching fallback node. In other words, backwards chaining is used in order to make the successful execution and fulfillment of a condition more robust. **To summarize**: a deep BT structure allows for making a condition more robust. However, expanding a condition too deep might cause implications on readability of the tree as it increases the complexity thereof, which is one of the most important benefits of using BTs.
- **Wide** - This structure allows for updating additional actions that would have to be executed before being able to successfully execute the base behaviour. One example being that before an object can be grasped successfully by the dexterous manipulator of a mobile robot, that it first has to circumvent an obstacle by moving the robot base around said obstacle. This structure, albeit being very minimalistic, would require the tree to be ticked several times before successfully satisfying the base behaviour, ultimately making the perceivability of the BT confusing. In the case that we want to tick the BT as minimally as possible when using a wide BT structure, the updated branches should contain the updated skill within a sequence alongside a duplicate of the base skill. Using duplicate nodes, however, is considered to be bad practice and thus should not be considered. Another downside is that updating in the width, which basically amounts to adding more conditions that have to possibly be satisfied,

does not allow for making the existing conditions more robust per se. **To summarize:** a wide structure, albeit looking minimalistic, can become confusing when becoming too large as it is required to go through the BT several times before being able to execute the base behaviour by means of successful execution of the updated behaviours a-priori in a satisfactory fashion.

- **Hybrid** - The hybrid structure refers to going both into the depth and width while updating a BT. Using this BT structure we can make conditions more robust by updating into the depth of a particular branch, whilst we can update alternatives of the base behaviour by going into the width of the tree when adding additional branches. In other words the hybrid structure mitigates the downsides of each of the separate structures out of which it is constructed.

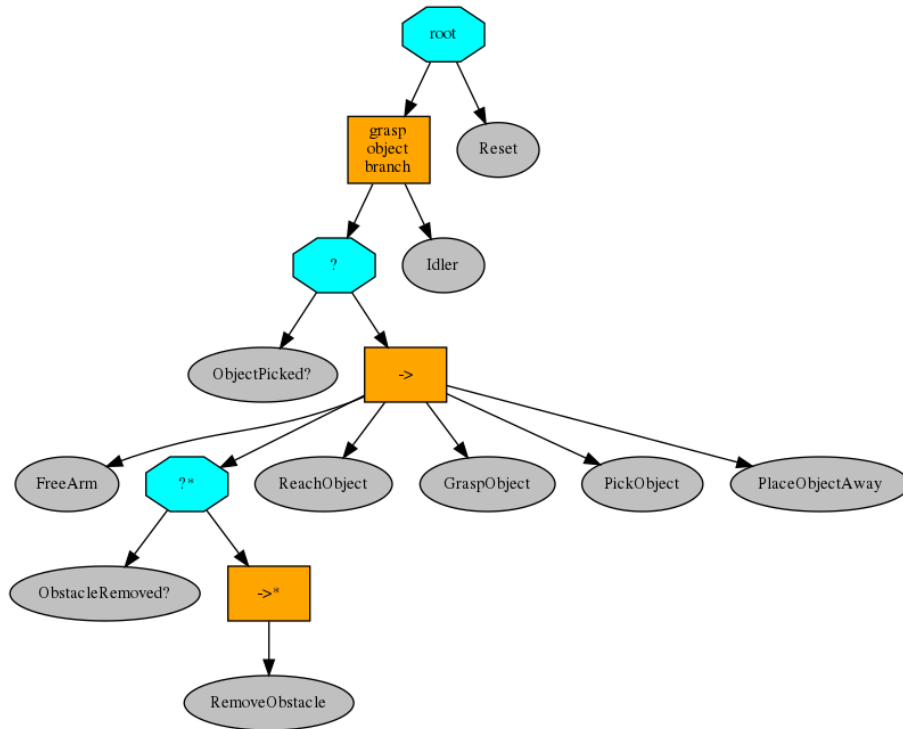


Figure 2: Example BT constructed using BTUF for executing a pick and place tasks while accounting for a possible obstacle that has to be removed if present.

BTUF makes use of all three different structures by allowing the user to either update in the depth (via Backward Chaining) or in the width (either extending the base behaviour or creating a parallel sub-branch with an alternative or different behaviour) of which the combination of the former structures form an overall hybrid BT as the outcome.

Given the structures above, it is hypothesised that the *hybrid* structure will be preferred by the participants as it combines the merits of both the *deep* and *wide* structure, whilst not sacrificing the human readability and understanding of the robot task as portrayed by the BT [1]. **In short:** BTUF is a framework that provides easy creation and subsequent adaptation of such BT structures.

1.3 Behaviour Tree Update Framework

BTUF makes use of a terminal based GUI (Graphical User Interface) or TUI (Text-Based User Interface), wherein human inputs are required to create a BT either from scratch or by adapting a

prior existing BT in order to teach an agent specific behaviours with the prospect of successfully executing certain robotic tasks.

Core features of BTUF are:

- Adding/inserting actions.
- Adding/inserting conditions.
- Backwards Chaining (substituting existing leafs *or* inserting entire backwards chained branches).
- Saving created BTs.
- Loading created BTs (to either execute, to further adapt or to use as an extension to the current tree in the form of a parallel branch).
- Removing conditions/nodes/backwards chained branches/parallel branches.
- Executing BTs.

Note that except from the root and the control flow nodes (sequences/fallbacks), all nodes within the constructed BTs are targetable for adaptation. The exemption of changing the control flow nodes is to uphold a hybrid structure for the BT as discussed prior in section 1.2.

A more in-depth discussion with respect to the inner workings/implementation and background theory of BTUF are found in de Lange (2022) [2].

1.4 Conventional Framework

The conventional framework can be regarded as an immensely simplified version of BTUF in terms of available features and is thus referred to as BTUF light (BTUF-L) or BTUF offline from now on. It is designed such that it should resemble a more conventional way of creating BTs which is *manual programming*, thus stating the reason why BTUF-L is referred to as the conventional framework as opposed to BTUF. Aside from it being hugely impractical, time-consuming and simply not realistic to ask participants to create a BT from scratch by means of manual programming, BTUF-L makes use of a TUI. The major difference is that BTUF-L does not allow incremental adaptation, meaning that the user has to create the entire BT from start to finish in one run. Basic functions such as placing and removing nodes (including control flow nodes, which is possible within BTUF) are present, as well as executing the BT after finishing the construction thereof are present. In other words, BTUF-L allows a human user to create a BT from scratch till finish by placing control flow and leaf nodes wherever desired and allowing BT execution thereafter.

To summarize, the main features of BTUF-L are:

- Adding actions and/or conditions.
- Removing actions and/or conditions.
- Adding control flow nodes (sequences and fallbacks)
- Executing the constructed BT

It is important to **note** that BTUF-L is not directly comparable to conventional manual programming as many differences are found in between. However, BTUF-L can pose as a comparative framework considering that, as opposed to BTUF, many of the decisions regarding node placement and overall BT structure are all left in the hands of the user similar to manually programming a BT. BTUF on the other hand, although allowing a lot of human input, most crucially constructs the basis of the BT on which entire behaviours, simple or complex, are built upon.

```

[START] Starting Behaviour Tree Update Framework
[INPUT] [1] Create tree [2] Load tree [3] Exit Program:
1
[INPUT] [1] Add condition [2] Add action:
1
[INPUT] Add Condition? [1] Yes [2] No: 1
[INFO] The available conditions are:
1 ArmBlocked?
2 ObjectReached?
3 ObjectGrasped?
4 ObjectPicked?
5 ObjectPlaced?
6 ObjectInspected?
7 ObstacleRemoved?
[INPUT] Which condition do you want to add?: 4
[INFO] Added condition:
ObjectPicked?
-----
Writing root.dot/svg/png
[DEBUG] root : Selector.setup()
[DEBUG] grasp object branch : Sequence.setup()
[DEBUG] ? : Selector.setup()
[DEBUG] -> : Sequence.setup()
[INFO] We have updated the tree 0 times...
[INPUT]Do you want to [1] Execute Behaviour [2] Update Behaviour [3] Exit Program?:
2
[INPUT] Do you want to [1] Add New Conditions/Actions/Branches [2] Backwards Chaining (Updating Existing Branches or Creating New Branches) [3] Remove Leafs/Branches? : 1
[INFO] BT Exists
[INPUT] Add [1] Condition [2] Action/Branch: 2
[INPUT] Add from [1] Action Library [2] Saved Trees?: 1
[INPUT] Insert Action at [1] End [2] Specific Index: 1
[INFO] Updating tree
[INFO] the available actions are:
1 FreeArm
2 ReachObject
3 GraspObject
4 PickObject
5 PlaceObject
6 PlaceObjectAway
7 InspectObject
8 RemoveObstacle
[INPUT] Select desired action:
1
[INFO] Taken action:
FreeArm
-----
Writing root.dot/svg/png
[INFO] We have updated the tree 1 times...
[INPUT]Do you want to [1] Execute Behaviour [2] Update Behaviour [3] Exit Program?:
2
[INPUT] Do you want to [1] Add New Conditions/Actions/Branches [2] Backwards Chaining (Updating Existing Branches or Creating New Branches) [3] Remove Leafs/Branches? : █

```

Figure 3: Example visual of the BTUF terminal environment. **Note** that BTUF-L uses the same text-based user interface with the main difference being the functions/features found within either of the two frameworks.

2 Method

2.1 Participant Selection

This study will focus on participants that are drafted from the employees of this thesis' facilitating robotics company Heemskerk Innovative Technologies (HIT). The participants are **not necessarily** required to have knowledge regarding Behaviour Trees and similar high-level policy representations, but is regarded as helpful. Knowledge in the field of robotics is neither a necessity for participation in this experiment, however considering the nature of HIT's employees most of the participants will have some degree (from a medium to very high level) of background knowledge regarding robotics engineering in general. As this experiment's only goal is to provide a proof of work in the form of some objective and subjective metrics, a small test group of around 5 to 10 human participants can be regarded as sufficient for drawing meaningful conclusions towards that end.

Participants are drafted and approached from HIT. This sets the median age at around 25-

30. It is assumed that age and the ability to perceive and understand BT structures do not have a negative correlation with one another. Moreover, in the case that there is a negative correlation (as might be proven in future studies or not) it does not affect the overall outcome of this experiment, as we are merely trying to find the general usability and performance of BTUF over more *conventional* methods as represented by BTUF-L. Participants will be given a hand-out with the necessary information with respect to their participation in the study, which includes a form of written consent, prior to their actual participation. Participants are let known that they are able to forfeit their participation at any given time before, during and after the experiment. Personal information, aside from age, gender and experience regarding BTs and robotics overall, are not asked nor published as they have no academic relevance in the case of this experiment.

2.2 Materials

Materials used for this experiment are:

- A printed or digital hand-out used for briefing the participants as well as for asking their written consent.
- A printed or digital hand-out showcasing the tasks that have to be completed by the participant using either one of the two frameworks as described in sections 1.3 and 1.4 respectively.
- A personal computer on which the participant has to perform the experiment tasks.
- A printed or digital hand-out on which to draw a BT and some general subjective questions regarding the experiment including a Van der Laan scale which is used as a simple assessment of system acceptance by the participants.

2.3 Experimental Design

Considering that this study uses a small group of participants, a within-subject design is used for the experimental set-up. In the case of a within-subject designed experiment, each participant tests all of the experimental conditions. In the case of this particular study, two practical parts are discerned from one another. In the first part, every participant has to teach a robotic agent (TIAGO by Pal Robotics) within a simulated environment how to perform a simple pick and place task using BTUF and BTUF-L in randomized order. Before working within the simulation, participants are asked to draw out a BT by hand in order to visualize what they approximately expect to create using either of the two frameworks based on some simple background knowledge regarding BTs and their general working principles. When using BTUF, participants are not specifically let known what a supposedly *good* tree will look like, but are made aware of the working principles of a BT, the functions of the used BT nodes. Therefore participants are expected to perform the experiments mostly based on trial-and-error and logical reasoning in order for successful completion. This is done so similarly for when participants use BTUF-L. With this similar manner of briefing in advance of using either one of the two frameworks, it is hypothesized that overall performance, as measured by the metrics as found in section 2.4, of BTUF will be higher in comparison to BTUF-L. After construction of a BT capable of executing the goal behaviour, participants are expected to adapt the BT to account for a novel unforeseen instance on the task (e.g. accounting for the sudden presence of an obstacle that hinders successful completion of the initial pick and place task).

In the second practical part, participants are asked to use BTUF to solve three additional subtasks that are solvable using only one action each. These subtasks are designed to highlight some core functions of BTUF, which are otherwise overlooked in the first part of the study. Because a within-subject design is used, each participant serves as their own control [3].

2.4 Metrics

This experiment makes use of both objective and subjective metrics as a means of validating the use of the proposed BTUF. The objective metrics which will be recorded are:

- **Time to completion** (TC)
- **Runs to completion** (RC) (measured from the amount of resets and/or executions (e.g. because of intermediate saving & loading of constructed trees, resetting BTUF to start anew *or* intermediate execution of the current BT))
- **Total BT nodes used in the final BT structure** (TBTN)
- **Amount of actions taken to completion** (AC)

The subjective metrics will be gathered by means of having participants fill out a subjective questionnaire including a Van der Laan acceptance scale as found in Appendix 2.5 [5].

2.5 Procedure

The experiments will be conducted at Heemskerk Innovative Technologies (HIT). The participants are briefed by means of a printed or digital hand-out explaining the outline and purpose of the experiment. The briefing includes a description of the task the participants have to perform and how much time they have to do so. All participants are asked to fill out a form of written consent after the briefing has finished, before actually commencing and partaking in the experiment. The participants are informed that they can cancel their participation at any given moment before, during and after the experiment. The participants are also let known that they hold the right to retract any of the experimental data gathered from their respective participation within the experiments. The participants identity as well as other personal information regarded as non-vital for the academic relevance of the experimental outcome will only be known to the researchers in question and will not be published to the public. Only after the briefing is done, the participant has been fully made aware of the experimental procedure and the participant's consent has been obtained, will the experiment start. The practical part of experiment itself has a duration of approximately 30 to 60 minutes. Before using either framework, the participants will be given a short demonstration of around 5-10 minutes about how to operate that respective framework. This demonstration is given for learning purposes and is aimed at giving insight to the participants about the overall working of BTs and the two frameworks. It is also supposed to level the playing field for all participants with respect to their prior background knowledge. During the experiment, the participants are asked to verbally state their actions and thoughts. The subject is asked to fill out a Van der Laan system acceptance scale for both BTUF and BTUF-L, giving the researcher(s) insight in some subjective metrics such as usefulness and desirability of using a framework such as BTUF over more *conventional* methods. Additional comments and findings by the subjects are asked for as well, in the case that the subjects desire to provide these. After a subject has finished the experiment task, if so desired by the subject, he/she is debriefed in a broader context about the experimental goals, which can initially influence the potential outcome of the study if it were done so before commencing the experiment. At last, after the experiments have been conducted for the specific individual in question, the subject is made aware of the completion of the study. All participants, if they so desire, will be notified in the future if any of the of the experimental results are published or made public.

References

- [1] Rudy De-Xin de Lange. Joining High-Level Symbolic & Low-Level Motion Learning Methods for Autonomous Manipulation of a Compound Task. *TU Delft Repository*, 2021.
- [2] Rudy De-Xin de Lange. Incremental Adaptation of Behaviour Trees Through Human Input. *TU Delft Repository*, 2022.
- [3] Joost CF De Winter and Dimitra Dodou. *Human Subject Research for Engineers: A Practical Guide*. Springer, 2017.
- [4] Andy Field. *Discovering statistics using IBM SPSS statistics*. sage, 2013.
- [5] Jinke D Van Der Laan, Adriaan Heino, and Dick De Waard. A simple procedure for the assessment of acceptance of advanced transport telematics. *Transportation Research Part C: Emerging Technologies*, 5(1):1–10, 1997.

Bibliography

- Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009. ISSN 09218890. doi: 10.1016/j.robot.2008.10.024. <http://dx.doi.org/10.1016/j.robot.2008.10.024>.
- Nils Axelsson and Gabriel Skantze. Modelling adaptive presentations in human-robot interaction using behaviour trees. *SIGDIAL 2019 - 20th Annual Meeting of the Special Interest Group Discourse Dialogue - Proceedings of the Conference*, (September):345–352, 2019. doi: 10.18653/v1/w19-5940.
- J. Andrew Bagnell, Felipe Cavalcanti, Lei Cui, Thomas Galluzzo, Martial Hebert, Moslem Kazemi, Matthew Klingensmith, Jacqueline Libby, Tian Yu Liu, Nancy Pollard, Mihail Pivtoraiko, Jean-Sebastien Valois, and Ranqi Zhu. An Integrated System for Autonomous Robotics Manipulation. 2012.
- Andre G Barto and Sridhar Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003. ISSN 02126567. doi: 10.1157/13086316.
- Herman Batelsson. Behavior Trees in the Unreal Engine Function and Application. Bachelor’s thesis, Uppsala Universitet, 2016.
- Sylvain Calinon. Robot Learning with Task-Parameterized Generative Models. pages 111–126, 2018. doi: 10.1007/978-3-319-60916-4{_}7.
- Ngee Choon Chia and Shu Peng Loh. Using the stochastic health state function to forecast healthcare demand and healthcare financing: Evidence from Singapore. *Review of Development Economics*, 22(3):1081–1104, 2018. ISSN 14679361. doi: 10.1111/rode.12528.
- Michele Colledanchise and Petter Ögren. How Behavior Trees modularize robustness and safety in hybrid systems. *IEEE International Conference on Intelligent Robots and Systems*, pages 1482–1488, 2014. ISSN 21530866. doi: 10.1109/IROS.2014.6942752.
- Michele Colledanchise and Petter Ögren. How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. *IEEE Transactions on Robotics*, 33(2):372–389, 2017. ISSN 15523098. doi: 10.1109/TRO.2016.2633567.
- Michele Colledanchise and Petter Ögren. Behavior Trees in Robotics and AI: An Introduction. 2020. doi: 10.1201/9780429489105. <http://arxiv.org/abs/1709.00084>
- Michele Colledanchise, Diogo Almeida, and Petter Ögren. Towards blended reactive planning and acting using behavior trees. *Proceedings - IEEE International Conference on Robotics and Automation*, 2019-May:8839–8845, 2019a. ISSN 10504729. doi: 10.1109/ICRA.2019.8794128.
- Michele Colledanchise, Ramvijas Parasuraman, and Petter Ögren. Learning of behavior trees for autonomous agents. *IEEE Transactions on Games*, 11(2):183–189, 2019b. ISSN 24751510. doi: 10.1109/TG.2018.2816806.
- Joost CF De Winter and Dimitra Dodou. *Human Subject Research for Engineers: A Practical Guide*. Springer, 2017.
- Peter F Edemekong, Deb Bomgaars, and Shoshana B Levy. Activities of Daily Living. 2020.
- Deborah Ellison, Danielle White, and Francisca Cisneros Farrar. Aging population. *Nursing Clinics of North America*, 50(1):185–213, 2015. ISSN 00296465. doi: 10.1016/j.cnur.2014.10.014. <http://dx.doi.org/10.1016/j.cnur.2014.10.014>.
- Marco Ewerton, Oleg Arenz, Guilherme Maeda, Dorothea Koert, Zlatko Kolev, Masaki Takahashi, and Jan Peters. Learning Trajectory Distributions for Assisted Teleoperation and Path Planning. *Frontiers in Robotics and AI*, 6(September):1–13, 2019. ISSN 2296-9144. doi: 10.3389/frobt.2019.

00089.

- Andy Field. *Discovering statistics using IBM SPSS statistics*. sage, 2013.
- Kevin French, Shiyu Wu, Tianyang Pan, Zheming Zhou, and Odest Chadwicke Jenkins. Learning behavior trees from demonstration. *Proceedings - IEEE International Conference on Robotics and Automation*, 2019-May:7791–7797, 2019. ISSN 10504729. doi: 10.1109/ICRA.2019.8794104.
- Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Backward-forward search for manipulation planning. *IEEE International Conference on Intelligent Robots and Systems*, 2015-December:6366–6373, 2015. ISSN 21530866. doi: 10.1109/IROS.2015.7354287.
- Malik Ghallab, Dana Nau, and Paolo Traverso. The actor’s view of automated planning and acting: A position paper. *Artificial Intelligence*, 208(1):1–17, 2014. ISSN 00043702. doi: 10.1016/j.artint.2013.11.002. <http://dx.doi.org/10.1016/j.artint.2013.11.002>.
- Barbara J. Grosz. Natural language processing. *Annual Review of Information Science and Technology*, 37(1):51–89, 2005. ISSN 00043702. doi: 10.1016/0004-3702(82)90032-7.
- Peter F. Hokayem and Mark W. Spong. Bilateral teleoperation: An historical survey. *Automatica*, 42(12):2035–2057, 2006. ISSN 00051098. doi: 10.1016/j.automatica.2006.06.027.
- S H Hosseini and K M Goher. Personal Care Robots for Older Adults: An Overview. *Asian Social Science*, 13(1):11, 2017. ISSN 19112017. doi: 10.5539/ass.v13n1p11.
- Thomas M. Howard, Stefanie Tellex, and Nicholas Roy. A natural language planner interface for mobile manipulators. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 6652–6659, 2014. ISSN 10504729. doi: 10.1109/ICRA.2014.6907841.
- Yordan Hristov, Alex Lascarides, and Subramanian Ramamoorthy. Interpretable latent spaces for learning from demonstration. *arXiv*, (CoRL), 2018. ISSN 23318422.
- Kyung-Rae Hyun, Sungwook Kang, and Sunmi Lee. Population Aging and Healthcare Expenditure in Korea. *Health Econ.*, 2015. doi: 10.1002/hec.3209.
- Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Learning rhythmic movements by demonstration using nonlinear oscillators. *IEEE International Conference on Intelligent Robots and Systems*, 1(October):958–963, 2002. doi: 10.1109/irids.2002.1041514.
- Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical movement primitives: Learning attractor models formotor behaviors. *Neural Computation*, 25(2):328–373, 2013. ISSN 08997667. doi: 10.1162/NECO_a_00393.
- Matteo Iovino, Edvards Scukins, Jonathan Styurd, Petter Ogrena, and Christian Smith. A Survey of Behavior Trees in Robotics and AI. *arXiv*, pages 1–22, 2020. ISSN 23318422.
- Rolf Isermann. Fault diagnosis of machines via parameter estimation and knowledge processing—tutorial paper. *Automatica*, 29(4):815–835, 1993.
- B. Iske and U. Rückert. A methodology for behaviour design of autonomous systems. *IEEE International Conference on Intelligent Robots and Systems*, 1:539–544, 2001. doi: 10.1109/iros.2001.973412.
- Damian Isla. Handling Complexity in the Halo 2 AI. In *GDC 2005 Proceedings*, volume 21, 2005.
- Damian Isla. Building a better battle. In *GDC 2005 Proceedings*, volume 32, 2008.
- Jeong Jung Kim, So Youn Park, and Ju Jang Lee. Adaptability improvement of Learning from Demonstration with Sequential Quadratic Programming for motion planning. *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, 2015-Augus:1032–1037, 2015. doi: 10.1109/AIM.2015.7222675.
- Kamil Kukliński, Kerstin Fischer, Ilka Marhenke, Franziska Kirstein, Maria V. Aus Der Wieschen, Dorte Sølvason, Norbert Krüger, and Thiusius Rajeeth Savarimuthu. Teleoperation for learning by demonstration: Data glove versus object manipulation for intuitive robot control. *International Congress on Ultra Modern Telecommunications and Control Systems and Workshops*, 2015-Janua (January):346–351, 2014. ISSN 2157023X. doi: 10.1109/ICUMT.2014.7002126.
- Jean-francois Lafleche, Shane Saunderson, Student Member, and Goldie Nejat. Robot Cooperative Behavior Learning Using Single-Shot Learning From Demonstration and Parallel Hidden Markov Models. 4(2):193–200, 2019.
- Jean-Claude Latombe. *Robot motion planning*, volume 124. Springer Science & Business Media, 2012.
- Myoun-Jae Lee. A proposal on game engine behavior tree. *Journal of Digital Convergence*, 14(8):

- 415–421, 2016.
- Wei Yin Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011. ISSN 19424795. doi: 10.1002/widm.8.
- Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ogren. Towards a unified behavior Trees framework for robot control. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 5420–5427, 2014. ISSN 10504729. doi: 10.1109/ICRA.2014.6907656.
- Agus Cahyo Nugroho. Expert system development for course enrollment process using ripple down rules in a university in surabaya. *ComTech: Computer, Mathematics and Engineering Applications*, 10(1):1–7, 2019.
- Alexandros Paraschos, Christian Daniel, Jan Peters, and Gerhard Neumann. Probabilistic movement primitives. *Advances in Neural Information Processing Systems*, pages 1–9, 2013. ISSN 10495258.
- Chris Paxton, Yonatan Bisk, Jesse Thomason, Arunkumar Byravan, and Dieter Fox. Prospection: Interpretable plans from language by predicting the future. *arXiv*, pages 6942–6948, 2019. ISSN 23318422.
- Alan Perotti, Guido Boella, and Artur d’Avila Garcez. Runtime verification through forward chaining. In *Runtime Verification*, pages 185–200. Springer, 2015.
- Corrado Pezzato. *Planning & Decision-making (RO47005) - Guest Lecture on State Machines & Behaviour Trees*. TU Delft, 2021.
- Muhammad Asif Rana, Mustafa Mukadam, Seyed Reza Ahmadzadeh, Sonia Chernova, and Byron Boots. Towards Robust Skill Generalization: Unifying Learning from Demonstration and Motion Planning. *Conference on Robot Learning*, (October):109–118, 2017. <http://proceedings.mlr.press/v78/rana17a.html>.
- Nathan D. Ratliff, Jan Issac, Daniel Kappler, Stan Birchfield, and Dieter Fox. Riemannian motion policies. *arXiv*, 2018. ISSN 23318422.
- Harish Ravichandar, Athanasios S. Polydoros, Sonia Chernova, and Aude Billard. Recent Advances in Robot Learning from Demonstration. *Annual Review of Control, Robotics, and Autonomous Systems*, 3(1):297–330, 2020. ISSN 2573-5144. doi: 10.1146/annurev-control-100819-063206.
- Ingo Rechenberg. Computational intelligence imitating life - chapter: Evolution strategy. Technical report, New York, NY (United States); Institute of Electrical and Electronics . . . , 1994.
- Hannah Ritchie and Max Roser. Age structure. *Our World in Data*, 2019. <https://ourworldindata.org/age-structure>.
- Ashwini Rupnawar, Ashwini Jagdale, and Samiksha Navsupe. Study on forward chaining and reverse chaining in expert system. *International Journal of Advanced Engineering Research and Science*, 3(12):236945, 2016.
- Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. *Imitation in Animals and Artifacts - Chapter 7: Learning to Fly*, volume 32. 2003. ISBN 0262042037. doi: 10.1108/k.2003.06732gae.004.
- Cory Ann Smarr, Tracy L. Mitzner, Jenay M. Beer, Akanksha Prakash, Tiffany L. Chen, Charles C. Kemp, and Wendy A. Rogers. Domestic Robots for Older Adults: Attitudes, Preferences, and Potential. *International Journal of Social Robotics*, 6(2):229–247, 2014. ISSN 18754805. doi: 10.1007/s12369-013-0220-0.
- Young Chol Song and Henry Kautz. A testbed for learning by demonstration from natural language and RGB-depth video. *Proceedings of the National Conference on Artificial Intelligence*, 3(Winograd 1972):2457–2458, 2012.
- StatisticKorea. Population projections for Korea. 2017. <http://kostat.go.kr>.
- Halit Bener Suay, Russell Toris, and Sonia Chernova. A Practical Comparison of Three Robot Learning from Demonstration Algorithm. *International Journal of Social Robotics*, 4(4):319–330, 2012. ISSN 18754791. doi: 10.1007/s12369-012-0158-7.
- Keith Sullivan, Sean Luke, and VA Ziparo. Hierarchical learning from demonstration on humanoid robots. *Proceedings of Humanoid Robots Learning from Human Interaction Workshop*, 2010.
- Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2): 146–160, 1972.
- David Vogt, Simon Stepputtis, Steve Grehl, Bernhard Jung, and Heni Ben Amor. A system for learn-

- ing continuous human-robot interactions from human-human demonstrations. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 2882–2889, 2017. ISSN 10504729. doi: 10.1109/ICRA.2017.7989334.
- Naoki Wake, Iori Yanokura, Kazuhiro Sasabuchi, and Katsushi Ikeuchi. Verbal focus-of-attention system for learning-from-demonstration. *arXiv preprint arXiv:2007.08705*, 2020.
- Tim Welschhold, Christian Dornhege, and Wolfram Burgard. Learning mobile manipulation actions from human demonstrations. *IEEE International Conference on Intelligent Robots and Systems*, 2017-September:3196–3201, 2017. ISSN 21530866. doi: 10.1109/IROS.2017.8206152.
- World Health Organization. Ageing and Health. 2018. <https://www.who.int/news-room/fact-sheets/detail/ageing-and-health>.
- Xuesu Xiao, Bo Liu, Garrett Warnell, Jonathan Fink, and Peter Stone. APPLD: Adaptive Planner Parameter Learning from Demonstration. *IEEE Robotics and Automation Letters*, 5(3):4541–4547, 2020. ISSN 23773766. doi: 10.1109/LRA.2020.3002217.
- Hongxin Zhang, Xingyu Lv, Wancong Leng, and Xuefeng Ma. Recent Advances on Vision-Based Robot Learning by Demonstration. *Recent Patents on Mechanical Engineering*, 11(4):269–284, 2018. ISSN 22127976. doi: 10.2174/2212797611666180917115823.
- Petter Ögren. *What is a Behavior Tree and How do they work? (BT intro part 1)*. KTH Royal Institute of Technology, 2019a. <https://www.youtube.com/watch?v=DCZJUvTQV5Q>.
- Petter Ögren. *How to create Behavior Trees using Backward Chaining (BT intro part 2)*. KTH Royal Institute of Technology, 2019b. <https://www.youtube.com/watch?v=dB7ZSsz890cwt=5s>.
- Petter Ögren. *Creating Behavior Trees using Decision Tree design (BT intro part 3)*. KTH Royal Institute of Technology, 2020a. <https://www.youtube.com/watch?v=L9KTzZO3C8s>.
- Petter Ögren. *Behavior Trees vs Finite State Machines (BT intro part 4)*. KTH Royal Institute of Technology, 2020b. <https://www.youtube.com/watch?v=gXrKGTPwfO8>.
- Petter Ögren. *Behavior Trees that avoid checking All conditions All the time (BT intro part 5)*. KTH Royal Institute of Technology, 2021a. <https://www.youtube.com/watch?v=UHUBYFal0DM>.
- Petter Ögren. *4 Common Bugs in Backward Chained Behavior Trees (BT intro part 6)*. KTH Royal Institute of Technology, 2021b. <https://www.youtube.com/watch?v=UHUBYFal0DM>.

