Behavior Trees in Robotics

MICHELE COLLEDANCHISE

Doctoral Thesis
Stockholm, Sweden 2017
Abstract

Behavior Trees (BTs) are a Control Architecture (CA) that was invented in the video game industry, for controlling non-player characters. In this thesis we investigate the possibilities of using BTs for controlling autonomous robots, from a theoretical as well as practical standpoint.

The next generation of robots will need to work, not only in the structured assembly lines of factories, but also in the unpredictable and dynamic environments of homes, shops, and other places where the space is shared with humans, and with different and possibly conflicting objectives. The nature of these environments makes it impossible to first compute the long sequence of actions needed to complete a task, and then blindly execute these actions. One way of addressing this problem is to perform a complete re-planning once a deviation is detected. Another way is to include feedback in the plan, and invoke additional incremental planning only when outside the scope of the feedback built into the plan. However, the feasibility of the latter option depends on the choice of CA, which thereby impacts the way the robot deals with unpredictable environments.

In this thesis we address the problem of analyzing BTs as a novel CA for robots. The philosophy of BTs is to create control policies that are both modular and reactive. Modular in the sense that control policies can be separated and recombined, and reactive in the sense that they efficiently respond to events that were not predicted, either caused by external agents, or by unexpected outcomes of robot’s own actions.

Firstly, we propose a new functional formulation of BTs that allows us to mathematically analyze key system properties using standard tools from robot control theory. In particular we analyze whenever a BT is safe, in terms of avoiding particular parts of the state space; and robust, in terms of having a large domain of operation. This formulation also allows us to compare BTs with other commonly used CAs such as Finite State Machines (FSMs); the Subsumption Architecture; Sequential Behavior Compositions; Decision Trees; AND-OR Trees; and Teleo-Reactive Programs.

Then we propose a framework to systematically analyze the efficiency and reliability of a given BT, in terms of expected time to completion and success probability. By including these performance measures in a user defined objective function, we can optimize the order of different fallback options in a given BT for minimizing such function.

Finally we show the advantages of using BTs within an Automated Planning framework. In particular we show how to synthesize a policy that is reactive, modular, safe, and fault tolerant with two different approaches: model-based (using planning), and model-free (using learning).
Sammanfattning.

Beteendeträd för Robotar.

Beteendeträd (BT) är en reglerarkitektur (RA) som introducerades i dataspelsbranchen för att styra motståndare i dataspel. I denna avhandling undersöker vi möjligheterna att använda BT för att styra robotar, både från ett teoretisk och praktiskt perspektiv.


I denna avhandling analyserar vi BT som en RA för robotar. Syftet med BT är att möjliggöra skapandet av beteenden som är modulara och reaktiva. Modulara i meningen att olika beteenden lätt kan tas isär och sättas ihop, och reaktiva i meningen att det är lätt att reagera på oförutsedda händelser, orsakade av antingen externa entiteter, eller oväntade följd av egna aktiviteter.

Först föreslår vi en ny funktionell formulering av BT, som möjliggör matematisk analys av olika systemegenskaper med hjälp av klassisk reglertechnik. Vi kan t.ex. analysera om ett BT är säkert, i meningen att det undvikar vissa delar av tillståndssrummet, och robust, i meningen att de löser uppgiften för en stor mängd initialtillstånd. Denna formulering möjliggör också jämförelser med andra vanliga RA, så som Tillståndsmaskiner, Subsumption-arkitekturen, Sequential Behavior Compositions, Beslutsträd, OCH-ELLER-träd och Teles-Reactiva program.

Sedan försöker vi ett ramverk för att systematiskt analysera och optimera effektiviteten och pålitligheten för ett BT, med avseende på tid till mål och sannolikhet att lyckas.

Slutligen analyserar vi möjligheterna att kombinera BT med metoder för att automatiskt skapa en reglerpolicy. Vi visar att en reaktiv, modular, säker och feltolerant policy kan skapas med både modellbaserade planeringsmetoder, och icke-modellbaserade inlärningsmetoder.
Acknowledgments

This thesis is the result of four years of studies. Four years of happiness and despair, long nights in the lab and unforgettable travels. In these years, I had the support of many people to whom I am very grateful.

First and foremost, I would like to express my sincere gratitude to my supervisor Petter Ögren, for his persisting support, firm motivation, and immense knowledge. For being always available to guide me through the different stages of my studies.

My co-supervisor Dimos Dimarogonas, for introducing me into the research life during my master thesis project, for his guidance on so many aspects of life and research, and for being a formidable leader in the RECONFIG project.

My co-supervisor Yiannis Karayannidis for the stimulating discussions about the correlations between control engineering and computer science, and for introducing me to the SARAFun project.

Danica Kragic and Patric Jensfel, for their wisdom and career advice during the supervisory group’s meetings.

Richard Murray, for providing me the opportunity to visit the Control and Dynamical Systems department at Caltech. For his great knowledge, experience, and enthusiasm. I wish to thank also the people who made my visit unforgettable: Catharine McGhan, Benson Christalin, Yuh-Shyang Wang, Ioannis Filippidis, Daniel Naftalovich, and Sumanth Dathathri.

Alejandro Marzinotto, for being my friend and office mate since the very beginning. For all the fruitful discussions about life and research, for his companionship during all the different trips we had, and for sharing the joy and sorrow of the Ph.D. path.

Meng Guo, for his valuable work in the RECONFIG project and for being a great companion during the numerous conference and project trips around the globe.

All the people working at the Robotics Perception and Learning lab at KTH, for being great companions. A particular thanks to: Alessandro Pieropan for our coffee breaks during which we talked about research, life, and Swedish weather. Akshaya Thippur, for bringing his liveliness in the lab; Francisco Viña, for sharing his programming experience with me; Diogo Almeida for his valuable work in the SARAFun project. Fredrik Båberg, Yuquan Wang, and Ramvivas Parasuraman for the discussions and fun during our group’s meetings.

The KTH team for the Amazon Picking Challenge, for the great and valuable experience: Alessandro Pieropan, Francisco Viña, Karl Pauwels, Kaiyu Hang, and Johannes Stork.

Rocco Santomo for his valuable help during the development of the first edition of the Behavior Tree library.

Friné Portal for all her efforts to make things run smoothly in the lab, for being always available to help me through the Swedish bureaucracy.

My close friends in Italy: Carmine, Fabio, Ilenia, Matteo, and Rocco, for always supporting me behind a computer screen.
VI

My close friends in Sweden: Davide, Karin, Nicolò, and Onofrio for discovering the beauties of Stockholm together.

A humongous thanks to my parents, Rita and Pasquale, and my sister Lucia, who believed in me at every single stage of my life, even when I did not.

Finally, I owe my biggest gratitude to Paola, for her love, patience, and wisdom over these years. Without her, I would not have come this far.

Michele Colledanchise
List of Papers

The thesis is based on the following papers:


Partial results of the papers above have been presented in the following conference proceedings:


Other Publications

The following additional works have been published during the author’s studies but they are not included in the thesis.


## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>I Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Challenges of control architectures</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Behavior Trees</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Research Question</td>
<td>11</td>
</tr>
<tr>
<td>1.4 Focus and Scope</td>
<td>11</td>
</tr>
<tr>
<td>1.5 Thesis Contribution</td>
<td>12</td>
</tr>
<tr>
<td>1.6 Thesis Outline</td>
<td>13</td>
</tr>
<tr>
<td>2 Common Robot Control Architectures</td>
<td>15</td>
</tr>
<tr>
<td>2.1 Finite State Machine</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Hierarchical Finite State Machines</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Subsumption Architecture</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Sequential Behavior Composition</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Teleo-Reactive programs</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Decision Trees</td>
<td>22</td>
</tr>
<tr>
<td>2.7 Design Principles</td>
<td>23</td>
</tr>
<tr>
<td>3 Behavior Trees</td>
<td>25</td>
</tr>
<tr>
<td>3.1 Classical Formulation of BTs</td>
<td>26</td>
</tr>
<tr>
<td>3.2 BTs as a Robot CA</td>
<td>29</td>
</tr>
<tr>
<td>3.3 BTs vs other Robot CAs</td>
<td>31</td>
</tr>
<tr>
<td>3.4 Use of BTs in real world applications</td>
<td>35</td>
</tr>
<tr>
<td>3.5 Disadvantages of BTs</td>
<td>40</td>
</tr>
<tr>
<td>4 Open Source ROS BT library</td>
<td>41</td>
</tr>
<tr>
<td>4.1 Purpose and Description</td>
<td>41</td>
</tr>
<tr>
<td>4.2 Software Structure</td>
<td>41</td>
</tr>
</tbody>
</table>
5 Summary of Papers 45
A How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture and Decision Trees 47
B How Behavior Trees Generalize the Teleo-Reactive Paradigm and AND-OR Trees 49
C Towards Blended Planning and Acting using Behavior Trees. A Reactive, Safe and Fault Tolerant Approach 51
D Stochastic Behavior Trees for Estimating and Optimizing the Performance of Reactive Plan Executions 53
E Learning of Reactive Behavior Trees for Autonomous Agents 55

6 Conclusions and Future Work 57

II Included Publications 65
A How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture and Decision Trees A1
1 Introduction A3
2 Background: Classical formulation of BTs A6
3 A New Functional Formulation of BTs A11
4 How BTs Modularize Hybrid Dynamical Systems A13
5 How BTs Generalize Decision Trees, the Subsumption Architecture and Sequential Behavior Compositions A24
6 Examples A30
7 Conclusions A40

B How Behavior Trees Generalize the Teleo-Reactive Paradigm and And-Or-Trees B1
1 Introduction B3
2 Background: BTs and TR B4
3 Analogy between And-Or-Trees and BTs B8
4 Analogy between TRs and BTs B10
5 Universal TRs and FT-Successful BTs B12
6 Conclusions B13

C Towards Blended Planning and Acting using Behavior Trees. A Reactive, Safe and Fault Tolerant Approach. C1
1 Introduction C3
2 Related work C6
3 Background: BT, HPN and HBF C8
4 Problem Formulation C12
Part I

Introduction
Chapter 1

Introduction

In this chapter we introduce the motivation and scope of the thesis. We start with some historical remarks about robots in human culture, then we focus on robots in the real world. We show some examples of desired behaviors, highlighting the corresponding challenges. We conclude the chapter with the thesis contributions and outline.

![Illustrations of artificial companions in modern literature](#)

Artificial companions, conceived as human-looking animated entity that can serve, protect, or interact with people, is a concept deeply rooted in human culture. Although the term robot and its modern literary conception as a mobile machine

---

1The figures are in the public domain. Attribution not legally required.
equipped with tools and human-like intelligence is fairly recent, the concept of artificial life has a long history and has evolved over time. Early myths present animated objects as instruments of divine or human will (e.g. the Tsukumogami in the Japanese mythology, the Golem of Prague in the Jewish folklore, and Talos in the Greek mythology). The first appearances in literature of artificial humans are traced back to the 1800s (see Figure 1.1). Edward Ellis, in his novel *The Steam Man of the Prairies*, introduces a colossal steam-powered man. This steam-man was constructed by a boy, who uses the steam-man to carry him in a carriage on several adventures. Carlo Collodi’s novel *Le avventure di Pinocchio* narrates the story of a woodcarver who wishes to have his marionette, Pinocchio, come alive. A magic fairy grants his wish and the story continues with the constant desire of Pinocchio to become a real boy. The first-ever introduction of a human-looking mechanical man with its own intelligence occurred in 1907 with Tik-Tok, in Lyman Frank Baum’s children’s novel *Ozma of Oz*, over a decade before the term robot was coined. Nowadays, some hundreds years later, we still wish we had a helpful version of those artificial companions.

Although the abstract idea and purpose of artificial companions has changed over time, one key concept remains unchanged: they coexist with humans and share the same environment. The robotics industry has made great strides in realizing such robots, however the current technology still cannot be used to design and develop, say, a fully functional robot helper for homes or offices, and make it available in the open market. The largest use of robots in industry is still found in the assembly lines of factories, where the robot’s motion has to be repeated in a cyclic fashion in a fixed workspace, with high precision. Human operators make sure that nothing on the line changes. If they need a change in the line, the software is manually updated accordingly. This type of setup is defined by an environment that is *structured*: the robot and the surroundings have a behavior that is predictable in space and time. Hence, while designing the robot’s software, engineers know what to expect, and when. Unfortunately, if we want artificial companions to co-exist with humans, the tasks they have to face differ in several fundamental ways from the ones in the assembly lines. Humans live in a world of uncertainty; every day they deal with tasks that take place in an environment that is *unstructured*. Due to the unpredictable and changing nature of this type of environment, the sequence of actions that is needed to complete a task cannot be found using *once-and-for-all* computation and it is not feasible via manual iterative software modification either. To address this, the robotics research community is currently working towards developing intelligent machines that can operate in a reasonably unstructured environment, allowing us to automate a wide span of tasks.

Automating tasks in uncertain environments is complex, and in some cases considered unfeasible. An interesting anecdote is the cancellation of NASA’s Hubble Space Telescope repair mission in 2004. The mission aimed to repair and replace batteries, sensors, electronic boards, and some other instruments of the Hubble Space Telescope. Being fully man-made, the telescope has a convenient mechanical structure that makes the repair task designable almost fully offline and executable.
1.1. CHALLENGES OF CONTROL ARCHITECTURES

by an autonomous robot. However, even with a convenient mechanical structure, the execution may not run as planned, because the robot may encounter unexpected hardware configuration (e.g. unscrewed bolts, broken plates, or twisted cables). Unlike in assembly lines, ground operators cannot check the setup regularly. Despite the small degree of unstructuredness, the mission raised serious challenges spanning from low level sensing to high level planning. Remarkably, even this small unstructuredness led to the project’s cancellation [21].

In the robotic research community, as stressed in [55], the high pressure to develop proof-of-concept solutions to test new theories has led scientists to neglect important aspects of a software system, such as maintainability, interoperability, and reusability. As an inevitable consequence, the vast majority of software applications implementing the robot’s functionalities are not reusable even in slightly different application scenarios, and experiments are often not replicable.

The functionality of a robot is encoded in the Control Architecture (CA) that describes how a certain task is carried out. The choice of CA can thus have significant impact on the way the robot deals with unstructured environments. This choice involves a number of challenges, as described below.

1.1 Challenges of control architectures

The choice of CA is important for autonomous robots. However, this issue usually rises in late stages of a robot’s development when it is generally too late for taking remedial actions. In real robot applications, it is often advantageous to choose a CA that allows hierarchically organized deliberation as well as continual planning and deliberation [43]. On a more practical side, it is preferable if the CA allows code reusability, modular design, heterogeneity of code developers’ expertise, and (possibly) human readability [13, 30]. Moreover a CA for robots should be sufficiently expressive [2]; suitable for automatic synthesis (planning and learning); and suitable for analysis. In detail, a list of design principles include:

Hierarchical organization: An autonomous robot needs to plan and execute its tasks hierarchically. Some tasks may need to perform some other operations (e.g. data gathering, online planning, etc.) in a specified hierarchy.

Continual closed-loop execution: For the type of unstructured environments we consider, the robot cannot plan a sequence of actions offline and then carry out those actions in a open-loop fashion.

Reusable code: Reusability principles must be followed when designing the software for any complex and long-term project. The ability to reuse designs relies in an essential way on the ability to build larger things from smaller parts, and on the independence of the input and output of those parts from their use in the project. Each functionality must interface the CA in a rigorous, universal and well-defined fashion.
**Modular design:** Modular design is an approach that subdivides a system into smaller parts or modules, that can be independently created and then used in different systems. A modular system can be characterized by functional partitioning into discrete and scalable modules. A modular design is loosely connected with code reusability and it can allow an heterogeneity of code developers’ expertise.

**Human readable:** A readable structure is desirable for reducing the cost of developing and debugging, especially when the task is human designed. The structure should remain readable even for large systems. Human readability requires a coherent and compact structure.

**Expressive:** A CA must be sufficiently expressive to encode a large variety of tasks.

**Suitable for analysis:** Safety critical robot applications require a deep and detailed analysis of qualitative and quantitative properties. These properties include: safety, in the sense of avoiding irreversible undesired behaviors; robustness, in the sense of a large domain of operation; efficiency, in the sense of time to completion; reliability, in the sense of success probability; and composability, in the sense on analyzing whether properties are preserved over compositions of sub-tasks.

**Suitable for automatic synthesis:** In some problem instances, it is preferable that the action ordering of a task, or a policy, is automatically synthesized using task-planning or machine learning techniques. The CA can influence the efficiency of such synthesis techniques (e.g. a FSM with a large number of transitions can drastically deteriorate the speed of an algorithm that has to consider all the possible paths in the FSMs).

To concretely illustrate the design principles above, consider the following simple example.

**Example 1.1.** A robot is asked to find a ball, pick it up and place it into a bin. If the robot fails to complete the task, it should go to a safe position and wait for a human operator. After picking up the ball (Figure 1.2a), the robot moves towards the bin (Figure 1.2b). While moving towards the bin, an external entity takes the ball from the robot’s gripper (Figure 1.2c) and immediately throws it in front of the robot, where it can be seen (Figure 1.2d). The robot aborts the execution of moving and it starts to approach the ball again.

In this example, the robot does not simply execute a pick-and-place task. It continually monitors the progress of the actions, stops whenever needed, skips planned actions, decides the actions to execute, and responds to exogenous events. In order to execute some actions, the robot might need to inject new actions into the plan (e.g. the robot might need to empty the bin before placing the ball). Hence the
1.1. CHALLENGES OF CONTROL ARCHITECTURES

(a) The robot is picking up the ball.
(b) The robot moves toward the bin (far away from the robot) with the ball in the hand.
(c) An external entity (a human) takes the ball from the robot gripper.
(d) The robot approaches the ball in the new location.

Figure 1.2: Execution stages of Example 1.1.

Task requires a CA suitable for extensions. These extensions might be man made (e.g., the robot asks the operator to update the current action policy) requiring a CA to be human readable, or automated (e.g., using model-based reasoning) requiring a CA to be suitable for automatic synthesis. In either case, to be able to easily extend and modify the action policy, its representation must be modular. In addition, new actions may subsume existing ones whenever needed (e.g., empty the bin if it is full must be executed before place the ball). This requires a hierarchical representation of the policy. Moreover there might be multiple different ways of carrying out a task (e.g., picking the ball using the left hand or the right hand). The robot must be able to decide which option is the best, requiring the CA to be suitable for analysis. Finally, once the policy is designed, it is desirable that it can be reused in other contexts.

Most used CAs do not show characteristics suitable for the properties described above. Take as an example a Finite State Machine (FSM) modeling the behavior of the robot in Example 1.1, depicted in Figure 1.3. As it can be seen, even for this simple example the FSM gets fairly complex with many transitions. This serious is-
CHAPTER 1. INTRODUCTION

Figure 1.3: FSM modeling the robot’s behavior in Example 1.1. The initial state has a thicker border.
1.2. Behavior Trees

Behavior Trees (BTs) are one of the most common CAs used in the video game industry. BTs were first introduced in [25], and is now an established tool appearing in textbooks [36, 52] and in game-coding software such as Pygame, Craft AI, Unity, and the Unreal Engine. BTs are appreciated for being highly modular, flexible and reusable. Most of the top profile video games use BTs to encode the task execution of NPCs. The philosophy of BTs is to encode behaviors that are modular and reactive. By modularity we mean the degree in which behaviors may be separated and recombined. By reactivity we mean the ability to react to exogenous events. Looking back at Example 1.1, using BTs instead of FSMs to implement the actions execution, allows us to describe the desired behavior in modules as depicted in Figure 1.4a. Such a behavior is composed by a sequence of sub-behaviors that are task independent, meaning that while creating a sub-behavior the designer does not need to know which sub-behavior will be performed next. Sub-behaviors can be designed recursively, adding more details as in Figure 1.4b. BTs are executed in a particular way, which will be described in Chapter 3, that allows the behavior to be carried out reactively. For example, the BTs in Figure 1.4 execute the sub-behavior \textit{Place Ball}, but simultaneously also verify that the ball is still found and picked. If, due to an exogenous event, the ball slips out from the robot’s hand, then the robot will abort the sub-behavior \textit{Place Ball} and will re-execute the sub-behavior \textit{Pick Ball} or \textit{Find Ball} according to the current situation.

---

2In video games, NPCs are characters controlled by the computer through artificial intelligence.
3http://www.pygame.org/project-cwyl-1004-.html
4http://www.craft.ai/
5https://www.unity3d.com/
6https://docs.unrealengine.com/latest/INT/Engine/ki/BehaviorTrees/
CHAPTER 1. INTRODUCTION

Figure 1.4: Illustrations of a BT carrying out a pick and place task with different degrees of abstraction.

1.2.1 Quotes on BTs

A convenient aspect of the video game industry is that their members often share experiences and toughs in online discussion forums. Here we report opinions on BTs from some highly skilled professionals:

- “There are a lot of different ways to create AI’s, and I feel like I’ve tried pretty much all of them at one point or another, but ever since I started using behavior trees, I wouldn’t want to do it any other way. I wish I could go back in time with this information and do some things differently.” Mike Weldon, Lead Programmer Disney Pixar Cars.

- “Behavior trees offer a good balance of supporting goal-oriented behaviors and reactivity. [...] While state machines are reasonably intuitive for simple cases, as they become more complex they are hard to keep goal-oriented. As the number of states increases, the transitions between states become

\[\text{http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php}\]
1.3 Research Question

Due to the success of BTs in the video game industry and due to the fact that NPCs show the capabilities that we desire for robots, we investigate the following research question in this thesis:

What are the advantages and disadvantages of using BTs as a CA in robotics?

1.4 Focus and Scope

We now list the focus of the thesis work, highlighting the problem we address.

- We are interested in analyzing the advantages and disadvantages of BTs as a CA in robotics.
- We are interested in applications where a robot needs to continually deliberate while acting, monitor the state of the world, monitor the actions execution, and abort or skips actions whenever needed.
- We do not focus on robots that operate in structured environments, fully predictable in space and time.
- We do not focus on designing the sensing and actuation needed for the low-level execution of actions.
- We are not trying to find the best CA.

---

9http://aigamedev.com/open/article/fsm-age-is-over/
1.5 Thesis Contribution

In this section we list the contributions of this thesis to the research community. The contributions are based on papers authored by the thesis' author. The individual contributions of the author are described in Chapter 5.

**Functional formulation of BTs.** We proposed a formal functional formulation of BTs that allows us to mathematically analyze key properties using standard tools from robot control theory. This formulation was published in [A] and [G].

**Structural analysis of Stochastic BTs.** The key properties we investigate are: safety, in terms of avoiding particular parts of the state space; and robustness, in terms of a large domain of operation. We also show how such properties are preserved in BT compositions. Preliminary results of such analysis have been published in [G]. Mature results have been published in [A].

**Performance analysis of BTs.** The performances measures we investigate are: efficiency, in terms of time to successful completion; and reliability, in terms of success probability. Preliminary results of such analysis have been presented in [F]. Mature results have been submitted in [D].

**BTs vs other CAs.** We compared BTs with other successful CA such as: FSMs, Subsumption Architectures, Sequential Compositions, Decision Trees, AND-OR Trees, and Teleo-Reactive Programs, highlighting the advantages and disadvantages of BTs compared to them. The results have been published in [A] and [B].

**BTs in planning.** We proposed a formal automated mechanism to synthesize a BT that has a desired behavior. The construction of the tree is inspired by the Hybrid Backward-Forward algorithm (HBF), which was proposed as an action planner for dealing with infinite state spaces [16]. The HBF algorithm has been shown to efficiently solve problems with large state spaces. Using an HBF algorithm we can refine the acting process by mapping the descriptive model of actions, which describes what the actions do, onto an operational model, which defines how to perform an action in certain circumstances. The proposed framework combines the planning capability in an infinite state space from HBF with the advantages of BTs in terms of reactivity, modularity, safety, and fault tolerance. The results have been submitted in [C].

**BTs in learning.** We proposed a model-free learning algorithm through the combination of a greedy learning algorithm with Genetic Programming (GP), allowing: a fast convergent algorithm; the realization of safe learning, where the algorithm is guaranteed to avoid some undesired behaviors; a human readable output, where a user can understand and edit the resulting action planner; and a computationally tractable solution for complex problems. The
tree structure makes BTs conveniently modular, and provides a perfect structure for GP, with a straightforward chromosome mapping allowing arbitrary sub-trees to cross-over and mutate. The results have been submitted in [E].

**Open Source ROS BT library.** We developed an open-source BT library running in the Robot Operating System (ROS) [51]. The user can code desired behaviors in both C++ and Python. The library has received interest in the online coding community, with other code developers using it and suggesting additional features.

### 1.6 Thesis Outline

The rest of this thesis is structured as follows: Chapter 2 reviews the common CAs that have found successful applications in robotics, highlighting their advantages and disadvantages. Chapter 3 introduces the BTs, comparing them with the other CAs and presents some real world robotic applications where BTs have been used successfully. Chapter 4 presents the BT ROS package developed at KTH. Chapter 5 summarizes the papers included in the second part of the thesis. Chapter 6 concludes the discussion of BTs in robotics and outlines potential future work. Finally, the second part of the manuscript is a collection of the papers composing this thesis.
Chapter 2

Common Robot Control Architectures

In this chapter, we overview a set of CAs that have found successful applications in robotics and we analyze their advantages and disadvantages. To allow formal analysis, robotics (and more generally computer science) needs to have formal models of its subject matters. Indeed, various task execution models exist, each abstracting away some of the properties of the real computation and modeling others.

2.1 Finite State Machine

FSMs are the most common mathematical model of computation where the system can be in only one of a finite number of states at any given time. A FSM can move between states in response to some events. The change from one state to another is called a transition. The analysis here is valid for all CAs based on FSMs (e.g. Mealy [39] and Moore [35] machines). Figure 2.1 shows an example of a FSM designed to carry out a grab-and-throw task.

2.1.1 Advantages and disadvantages

FSMs are widely used due to their two main advantages:

- Intuitive structure: the structure of FSMs is fairly intuitive to a human operator.
- Ease of implementation: FSMs can be implemented in any sequential programming language.

However, FSMs show their weakness when the task to describe is complex and the number of states increases. In particular, FSMs show the following disadvantages:
CHAPTER 2. COMMON ROBOT CONTROL ARCHITECTURES

Figure 2.1: Graphical representation of a FSM to carry out a simple grab-and-throw task. The initial state has a ticker border.

- Maintainability: Adding or removing states requires to re-evaluate possibly all the transitions and internal states of the whole FSM. This makes FSMs highly susceptible to human design errors and makes them inefficient to be used and generated by computer programs.

- Scalability: FSMs with many states and many transitions between them are hard to modify, for both humans and computers.

- Reusability: The transitions between states may depend on internal variables (like in a Hybrid Automaton), making it unpractical to encode the same task for use in multiple projects.

2.2 Hierarchical Finite State Machines

Hierarchical Finite State Machines (HFSMs), also known as state charts [22], where developed to alleviate some of the disadvantages of FSMs. In a HFSM, a state can itself contain one or more states. A state containing two or more states is called a super state. In a HFSM, a generalized transition is a transition between super states. Generalized transitions reduce the presence of redundant transitions connecting super states rather than connecting each state individually. Figure 2.2 shows an example of a HFSM for a video game character.

2.2.1 Advantages and disadvantages

The main advantages of HFSMs are:

- Modularity: it is possible to separate the tasks in sub-tasks. However these sub-tasks often still depend on each other through state-dependent transitions.
2.2. HIERARCHICAL FINITE STATE MACHINES

![Diagram of HFSM controlling a NPC of a combat game]

**Figure 2.2:** Example of a HFSM controlling a NPC of a combat game. *Patrol, Use Rifle,* and *Use Handgun* are superstates.

- Behavior inheritance: The state nesting in HFSMs allows so-called *behavior inheritance.* Behavior inheritance allows sub-states to inherit behaviors from the superstate; for example, in the HFSM depicted in Figure 2.2, while in the sub-states inside *Use Handgun,* the character holds the weapon using one hand whereas while in the sub-states inside *Use Rifle,* the character holds the weapon using two hands. The behavior of how to hold the weapon is inherited from the superstate.

The main disadvantages of HFSMs are:

- Maintainability: Adding or removing states is still hard. A long sequence of actions, with the possibility of going back in the sequence and re-execute a task that was undone by external agents (e.g. the environment), still requires a fully connected sub-graph.

- Non intuitive hierarchy: Although HFSMs were conceived as a hierarchical version of FSMs, the hierarchy has to be user defined and editing such a hierarchy is sometimes not intuitive.
2.3 Subsumption Architecture

The Subsumption Architecture [6] is heavily associated with the behavior-based robotic architecture, which was very popular in the late 1980s and 90s. This architecture has been widely influential in autonomous robotics and elsewhere in real-time AI and it has found a number of successful applications [7]. The basic idea of the Subsumtion Architecture is to have several controllers, each one implementing a task, running in parallel. Each controller is allowed to output both its actuation commands and a binary value that signifies if it wants to control the robot or not. The controllers are ordered according to some priority (usually user defined), and the highest priority controller, out of the ones that want to control the robot, has the access to the actuators. Thus, a controller with a higher priority is able to subsume a lower level one. Figure 2.3 shows and example of a Subsumption Architecture.

![Subsumption Architecture Diagram](image)

**Figure 2.3:** Example of Subsumption Architecture composed by three controllers. The controller *Stop if Overheated* subsumes the controller *Recharge if Needed*, which subsumes the controller *Do Other Tasks*.

2.3.1 Advantages and disadvantages

The Subsumption Architecture has many practical advantages, in particular:

- Easy development: The Subsumption Architecture is naturally well suited for iterative development and testing.

- Modularity: The Subsumption Architecture connects limited, task-specific actions.

- Hierarchy: The controllers are hierarchically ordered, which makes it possible to define high priority behaviors (e.g. safety guarantees) that override others.

The main disadvantages of the Subsumption Architecture are:

- Scalability: Designing complex action selection through a distributed system of inhibition and suppression can be hard.

- Maintainability: Due to the lack of structure, the consequences of adding or removing condition-action rules can be hard to estimate.
2.4 Sequential Behavior Composition

The basic idea of the Sequential Behavior Composition [8] is to expand the domain of a controller by using several other controllers where the asymptotically stable equilibrium of each controller is either the goal state, or it is inside the region of attraction of the next controller. Since each controller moves the state to its own local goal, the state crosses the domains of the different controllers. This process is repeated until the state reaches the domain of the final controller, which is the one that drives the state to the final goal. Graphically, if we represent the Lyapunov function of each controller as a funnel, the bottom of a funnel must then be inside the funnel of the next controller. Figure 2.4 shows this method of combining controllers.

![Figure 2.4: Example of Sequential Behavior Composition of three controllers where each of them has a Lyapunov function represented as a funnel. Each controller is active if the state is outside the domains of lower controllers (solid lines). The lowest controller stabilizes the system so that it reaches the goal state.](image)
2.4.1 Advantages and disadvantages

The main advantages of a Sequential Behavior Composition are:

- **Execution analysis**: With the Sequential Behavior Composition it is easy to analyze if the execution of a sub-task impedes the execution of another sub-task.
- **Modularity**: sub-tasks can be developed separately as long the local goal of one controller is in the domain of the next one.
- **Robustness**: sub-tasks can be added to increase the domain of operation.

The main disadvantages of a Sequential Behavior Composition are:

- **Maintainability**: Adding or removing controllers can be demanding as their region of attraction and goal states are tightly related.
- **Reusability**: The local goal states depend on the low level controllers, making the code written in a Sequential Behavior Composition unpractical to reuse.
- **Human readability**: It is hard to visualize a Sequential Behavior Composition in a arbitrary N-dimensional state space.

2.5 Teleo-Reactive programs

Teleo-Reactive (TR) programs were introduced by Nils Nilsson [45] at Stanford University in 1994 to allow engineers to define the behavior of a robotics system that has to achieve specific goals while being responsive to changes in the environment. A TR program is composed of a set of prioritized condition-action rules that directs the agent towards a goal state (hence the term teleo) while monitoring the environmental changes (hence the term reactive). In its simplest form, a TR program is denoted by a list of condition-action rules as the following:

\[
\begin{align*}
    c_1 & \rightarrow a_1 \\
    c_2 & \rightarrow a_2 \\
    \ldots & \\
    c_m & \rightarrow a_m
\end{align*}
\]

where the \( c_i \) are conditions and \( a_i \) are actions. The condition-action rules list is scanned from the top until it finds a condition that holds, then the corresponding action is executed. In a TR program, actions are usually **durative** rather than discrete. A durative action is one that continues indefinitely in time (e.g. the action \textit{move forwards} is a durative action). In a TR program, a durative action is executed as long as its corresponding condition remains the condition with the highest priority among the ones that hold. When the highest priority condition that
holds changes, the action executed changes accordingly. Thus, the conditions must be evaluated continuously so that the action associated with the current highest priority condition that holds, is always the one being executed. A running action terminates when its corresponding condition ceases to hold or when another condition with higher priority takes precedence. Figure 2.5 shows an example of a TR program for navigating in an obstacle-free environment.

\[
\begin{align*}
\text{Equal}(\text{pos}, \text{goal}) & \rightarrow \text{Idle} \\
\text{Heading Towards}(\text{goal}) & \rightarrow \text{Go Forwards} \\
\text{(else)} & \rightarrow \text{Rotate}
\end{align*}
\]

**Figure 2.5:** Example of teleoreactive program carrying out a navigation task. If the robot is in the goal position, the action performed is \text{Idle} (no actions executed). Otherwise if it is heading towards the goal, the action performed is \text{Go Forwards}. Otherwise, the robot performs the action \text{Rotate}.

TR programs have been extended in several directions, including integrating TR programs with automatic planning and machine learning [4, 58], removing redundant parts of a TR program [41], and using TR programs to play robot soccer [19].

### 2.5.1 Advantages and disadvantages

The main advantages of a TR program are:

- **Reactive execution:** TR programs allow a reactive execution by continually monitoring the conditions and aborting actions when needed.
- **Intuitive structure:** The list of condition-action rules is intuitive to design for small problems.

The main disadvantages of a TR program are:

- **Maintainability:** Due to its structure (a long list of rules), adding or removing condition-action rules is prone to errors when a TR program has to encode a complex system. In those cases, a TR program takes the shape of a long list.
- **Failure handling:** To model failure handling, a TR program needs to have a condition that verifies if an action fails.
2.6 Decision Trees

A Decision Tree (DT) is a directed tree that represents a list of nested if-then clauses used to derive decisions [53]. Leaf nodes describe decisions, conclusions, or actions to be carried out, whereas non-leaf nodes describe predicates to be evaluated. Figure 2.6 shows a DT where according to some conditions, a robot will decide what to do.

![Decision Tree Diagram]

**Figure 2.6**: Example of DT executing a generic robotic task. The predicate are evaluated traversing the tree in a top-down fashion.

2.6.1 Advantages and disadvantages

The main advantages of a DT are:

- **Modularity**: DT’s structure is modular, in the sense that a sub-DT can be developed independently from the rest of the DT.

- **Hierarchy**: DT’s structure is hierarchical, in the sense that predicates are evaluated in a top-down fashion.

- **Intuitive structure**: The list of predicates to be evaluated is easy to design.

The main disadvantages of a DT are:

- **Repetitions**: To describe a reactive behavior, a given predicate must be reevaluated at different depths of the tree resulting in a DT with many repetitions.

- **Maintainability**: Due to repetitions, if the number of outgoing arcs from a predicate should change, this will affect the entire tree where such predicates appears.


2.7 Design Principles

As mentioned in Section 1.1, we may consider a set of different design principles when choosing a CA. Below we discuss each design principle for each CA reviewed in this chapter. Table 2.1 summarizes the discussion.

<table>
<thead>
<tr>
<th>Hierarchical organization</th>
<th>FSM</th>
<th>HFSM</th>
<th>Sub. Arc</th>
<th>Seq Com.</th>
<th>TR prog</th>
<th>DTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closed-loop execution</td>
<td>hard</td>
<td>easy</td>
<td>easy</td>
<td>hard</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>Reusable code</td>
<td>hard</td>
<td>hard</td>
<td>easy</td>
<td>hard</td>
<td>easy</td>
<td>hard</td>
</tr>
<tr>
<td>Modular design</td>
<td>hard</td>
<td>easy</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>Human readable</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>Sufficiently expressive</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
</tr>
<tr>
<td>Suitable for analysis</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
</tr>
<tr>
<td>Suitable for synthesis</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of CAs based on the properties outlined in Section 1.1.

Hierarchical organization: The Subsumption Architecture allows a hierarchical heterogeneous composition of different tasks. HFSMs and DTs show the most primitive hierarchical organization which has to be user defined and is difficult to modify. The other CAs reviewed here do not have this property.

Continual closed-loop execution: TR programs were conceived precisely for this purpose. In a FSM, the monitoring will have to be done in the current running state (or superstate in a HFSM), increasing the complexity. In the Subsumption Architecture, each controller keeps reading the sensory input in a closed-loop fashion. The other CAs reviewed here do not show this property.

Reusable code: In a Subsumption Architecture, the input and output commands of each controller are independent from the overall task. This makes the implementation of each controller reusable for different designs. A similar argument
applies to TR programs, where the implementation of each condition-action predicate is independent from the program itself. The other CAs reviewed here do not have this property.

**Modular design:** HFSMs have a quite primitive modular design. In a HFSM, the execution inside each super state can be independently designed with the hard constraint that implementation must be compliant with the outgoing and incoming generalized transitions. DTs allow a modular design due to their tree structure. The other CAs reviewed here do not have this property.

**Human readable:** DTs, due to their tree structure, remain readable independently from their size. The other CAs discussed do not have this property unless in fairly small systems.

**Sufficiently expressive:** To a certain degree, all the CAs above are sufficiently expressive to encode a large variety of tasks.

**Suitable for analysis:** The Sequential Behavior Composition was conceived for analysis purposes. Analytical tools are available for all the CAs above. Some tools are more used than others.

**Suitable for automatic synthesis:** Many planning and learning techniques can be found in the state of the art, some more popular than others.
Chapter 3

Behavior Trees

BTs were developed in the video game industry, as a tool to increase modularity in the control structures of NPCs [9, 25, 26, 36, 52]. In this billion-dollar industry, modularity is a key property to enable reusability of code, incremental design of functionality, and efficient testing.

In games, the control structures of NPCs were naturally formulated in terms of FSM. However, just as Petri Nets [42] provide an alternative view of FSMs that emphasize concurrency, BTs provide an alternative view of FSMs that emphasize modularity.

Following the development in the industry, BTs have now also started to receive attention in academia [3, 5, 10, 15, 20, 23, 29, 32, 34, 44, 46, 54]. At Carnegie Mellon University, BTs have been used extensively to do robotic manipulation [3, 15]. The fact that modularity is the key reason for using BTs is clear from the following quote from [3]: “The main advantage is that individual behaviors can easily be reused in the context of another higher-level behavior, without needing to specify how they relate to subsequent behaviors”.

BTs have also been used to enable non-experts to do robot programming of pick and place operations, due to their “modular, adaptable representation of a robotic task” [20] and allowed “end-users to visually create programs with the same amount of complexity and power as traditionally-written programs” [47]. Furthermore, BTs have been proposed as a key component in brain surgery robotics due to their “flexibility, reusability, and simple syntax” [23].

The advantages of BTs as compared to FSMs was the reason for extending the so-called JADE agent Behavior Model with BTs in [5], and the benefits of using BTs to control complex multi mission UAVs was described in [46].

The modularity of BTs was used to address the formal verification of mission plans in [29] while the execution time of stochastic BTs was analyzed in [10]. BTs have also been studied in machine learning applications [32, 44] and details regarding efficient parameter passing were investigated in [54]. Finally, a Modelica implementation of BTs was presented in [28].
3.1 Classical Formulation of BTs

Formally speaking, a BT is a directed rooted tree where the internal nodes also are called control flow nodes and leaf nodes are called execution nodes. For each connected node we use the common terminology of parent and child. The root is the node without parents; all other nodes have one parent. The control flow nodes have at least one child. Graphically, the children of a node are placed below it, as shown in Figures 3.1-3.3.

A BT starts its execution from the root node that generates ticks with a given frequency, which are sent to its children. A tick is a signal that allows the execution of a node. A node is executed if and only if it receives ticks. The child immediately returns to the parent running if its execution is ongoing, success if it has achieved its goal, or failure otherwise.

In the classical formulation, there exist four categories of control flow nodes (fallback, sequence, parallel, and decorator) and two categories of execution nodes (action and condition).

**Fallback** The fallback node routes the ticks to its children from the left until it finds a child that returns either success or running, then it returns success or running accordingly to its parent. It returns failure if and only if all its children return failure. Note that when a child returns running or success, the fallback node does not route the ticks to the next child (if any). The fallback node is depicted with a box containing the label “?”.

![Figure 3.1: Graphical representation of a fallback node with N children.](image)

**Sequence** The sequence node routes the ticks to its children from the left until it finds a child that returns either failure or running, then it returns failure or running accordingly to its parent. It returns success if and only if all its children return success. Note that when a child returns running or failure, the sequence node does not route the ticks to the next child (if any). The sequence node is depicted with a box containing the label “→”, shown in Figure 3.2.

---

1Fallback nodes are sometimes called selector nodes.
3.1. CLASSICAL FORMULATION OF BTS

Parallel The parallel node routes the ticks to all its children and it returns success if $M$ children return success, it returns failure if $N - M + 1$ children return failure, and it returns running otherwise where $N$ is the number of children and $M \leq N$ is a user defined threshold. The parallel node is graphically depicted with a box containing the label “$\Rightarrow$”, shown in Figure 3.3.

Action When it receives ticks, an action node executes a command. It returns success if the action is correctly completed or failure if the action has failed. While the action is ongoing it returns running. An action node is shown in Figure 3.4a.

Condition

Decorator node. The label describes the user defined policy.

Policy

Decorate node. The label describes the user defined policy.

Figure 3.2: Graphical representation of a sequence node with $N$ children.

Figure 3.3: Graphical representation of a parallel node with $N$ children.

Figure 3.4: Graphical representation of action (a), condition (b), and decorator (c) node.
**Condition** When it receives ticks, a condition node checks a proposition. It returns *success* or *failure* depending if the proposition holds or not. Note that a condition node never returns a status of *running*. A condition node is shown in Figure 3.4b.

**Decorator** The decorator node is a control flow node with a single child that manipulates the return status of its child according to a user-defined rule and also selectively ticks the child according to some predefined rules. For example, an *invert* decorator inverts the *success/failure* status of the child; a *max-N-tries* decorator only lets its child fail $N$ times, then always returns *failure* without ticking the child; a *max-T-sec* decorator lets the child run for $T$ seconds, then, if the child is still running, the decorator returns *failure* without ticking the child. The decorator is graphically represented with a rhombus, as in Figure 3.4c.

### 3.1.1 Control Flow Nodes with Memory

The control flow nodes sequence and fallback keep sending ticks to the children to the left of a running child, in order to verify whether a child has to be re-executed and the current one has to be preempted. However sometimes the user learns that a child, once executed, does not need to be re-executed.

Control flow nodes with memory [36] have been introduced to enable the designer to avoid the unwanted re-execution of some nodes. Control flow nodes with memory always remember whether a child has returned *success* or *failure*, avoiding the re-execution of the child until the whole sequence or fallback finishes in either *success* or *failure*. In this thesis, nodes with memory are graphically represented with the addition of the symbol “$*$” (e.g. a sequence node with memory is graphically represented by a box with a “$*$”). The memory is cleared when the parent node returns either *success* or *failure* so that the next activation all children are considered. Note however that every execution of a control flow node with memory can be obtained with a non-memory BT using some auxiliary conditions as shown in Figure 3.5. Hence nodes with memory can be considered to be syntactic sugar.

**Figure 3.5:** Relation between memory and memory-less BT nodes.
Remark 1. Some BT implementations do not include the running return status [36]. Instead, they let each action run until it returns failure or success. We denote these BTs as non-reactive, since they do not allow actions other than the currently active one to react to changes. This is a significant limitation on non-reactive BTs, which was also noted in [36]. A non-reactive BT can be seen as a BT with only memory nodes.

3.1.2 Execution Example

Consider the BT in Figure 3.6 encoding the behavior of Example 1.1 (in Chapter 1). When the execution starts, the ticks traverse the BT reaching the condition node Ball Found. The robot does not know the ball position hence the condition node returns failure and the ticks reach the action Find Ball, which returns running (see Figure 3.7a). While executing this action, the robot finally sees the ball with the camera. In this new situation the robot knows the ball position. Hence the condition node Ball Found now returns success resulting in the ticks no longer reaching the action node Find Ball and the action is preempted. The ticks continue exploring the tree, and reach the condition node Ball Close, which returns failure (the ball is far away) and then reach the action node Approach Ball, which returns running (see Figure 3.7b). Then the robot eventually reaches the ball, picks it up and goes towards the bin (see Figure 3.7c). When an external entity moves the ball from the robot hand to the floor (where the ball is visible), the condition node Ball Found returns success while the condition node Ball Close returns failure. In this situation the ticks no longer reach the action Approach Bin (which is preempted) and they instead reach the action Approach Ball (see Figure 3.7d).

3.2 BTs as a Robot CA

In this section we analyze how BTs are contextualized with respect to the CAs mentioned in Chapter 2. We show that BTs satisfy, to a certain degree, the properties stressed in [13, 30, 43] and described in Section 1.1.

Hierarchical organization: The ticks traverse the tree in a Depth First Search fashion. This results in executing the BTs nodes of the same depth from left...
to right. This intrinsic hierarchy, combined with the modular tree-structure design, allows plan extensions, plan refinement, and hierarchical deliberation.

**Continual closed loop execution:** The continual generation of ticks and their tree traversal result in a closed loop execution. Actions are executed and aborted according to the ticks’ traversal, which depends on the leaf nodes’ return statuses. Leaf nodes are tightly connected with the environment (e.g., condition nodes evaluate the overall system properties and action nodes re-
3.3. BTS VS OTHER ROBOT CAS

turn failure/success if the action failed/succeeded). Hence BTs are highly responsive to changes in the environment.

**Reusable code:** When writing the code of a leaf node, the developer needs to just take care of returning the correct return status which is universally predefined as either running, success, or failure. Unlike FSMs and HFSMs, where the outgoing transitions require knowledge about the next state, in BTs leaf nodes are developed disregarding which node is going to be executed next. Hence, the BT logic is independent from the leaf node executions and vice versa. Moreover, since BTs are also modular, the code of an entire sub-tree can be easily reused.

**Modular design:** Due to the standard interface between nodes via return statuses and due to the tree structure, BTs are truly modular.

**Human readable:** Due to the tree structure, like DTs, BTs have the advantages of human readability even for large and complex BTs.

**Sufficiently expressive:** BTs generalize FSMs [34, 46]; DTs [A]; the Subsumption Architecture [A]; the Sequential Behavior Composition [A]; AND-OR Trees [B]; TR programs [B]; which makes BTs at least as expressive as them.

**Suitable for analysis:** BTs have tools available to evaluate performance [D,F]

**Suitable for synthesis:** It has been shown that BTs are suitable for both planning [C] and learning [E].

3.3 BTs vs other Robot CAs

From a theoretical standpoint, every execution described by a BT can be described by a FSM and vice-versa [34, 46]. However, due to the number of transitions, using a FSM as a CA is unpractical for some robotics applications. Moreover, a FSM assumes that the propositions that trigger the outgoing transitions from the same state are mutually exclusive. When implemented, the propositions are checked regularly in discrete time, hence there exists a non-zero probability that two or more propositions hold simultaneously after one cycle. To solve this problem we need to redefine the meaning of some transitions as done in the FSM in Figure 1.3 making the propositions of the outgoing transitions mutually exclusive. A FSM of this format is impractical to design for both humans and computers. Adding and removing behaviors by a human is prone to errors. After adding a new state, each existing transition must be re-evaluated (possibly removed or replaced) and new transitions from/to the new state must be evaluated as well. A high number of transitions makes any automated process to analyze or synthesize FSMs computationally expensive.

HFSMs is the most similar CA to BTs in terms of purpose and expressiveness. To compare BTs with HFSMs we use a more complicated example. Consider the
BT shown in Figure 3.9 describing the behavior of a humanoid robot. We can describe the same functionality using the HFSM shown in Figure 3.8. Note that we have used the standard notation \cite{22} of HFSMs to denote two activities running in parallel with a dashed line as separation. One important difference is that, in HFSMs, each layer in the hierarchy needs to be added explicitly, whereas in BTs every sub-tree can be seen as a module of its own, with the same interface as an atomic action.

In the HFSM shown in Figure 3.8, a proposition needs to be given for each transition, and to improve readability we have numbered these propositions from $C_1$ to $C_{10}$. In the top layer of the HFSM we have the sub-HFSMs of Self Protection and Perform Activities. Inside the latter we have two parallel sub-HFSMs. One is handling the user interaction, while the larger one contains a complete directed graph handling the switching between the different activities. Finally, Play Ball Game is yet another sub-HFSM with the ball tracking running in parallel with another complete directed graph, handling the reactive switching between Approach Ball, Grasp Ball, and Throw Ball.

It is clear from the two figures how modularity is handled by the HFSM. The explicitly defined sub-HFSM encapsulates Self Protection, Perform Activities and Play Ball Game. However, inside these sub-HFSMs, the transition structure is a complete directed graph, with $n(n-1)$ transitions that need to be maintained ($n$ being the number of nodes).

Looking at the available software for designing and executing FSMs and BTs, we note that the tools on the FSM side, such as IBM Rhapsody\textsuperscript{2} and Stateflow\textsuperscript{3} are much more mature. Still, many computer game development platforms, such as Unity3d\textsuperscript{4} and Unreal Engine\textsuperscript{5} now have tools for working with BTs. For those who wish to implement their own framework, we note that standard FSM implementation is quite straightforward, whereas both HFSMs and BTs require more consideration. However open source implementations are available for both.\textsuperscript{6,7}

Finally, in [A] we compare BTs with DTs; the subsumption architecture; and the sequential composition. In [B] we compare BTs with AND-OR Trees and TR programs.

\textsuperscript{2}http://www-03.ibm.com/software/products/sv/ratirhapi
\textsuperscript{3}http://se.mathworks.com/products/stateflow/?refresh=true
\textsuperscript{4}http://forum.unity3d.com/threads/behavior-designer-behavior-trees-for-everyone.227497/
\textsuperscript{5}https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/QuickStart/
\textsuperscript{6}http://michelecolledanchise.com/github/ (BTs)
\textsuperscript{7}http://qfsms.sourceforge.net (FSMs)
Figure 3.8: A HFSM description of the BT in Figure 3.9. The transition conditions are shown at the end of each arrow to indicate the direction of the transition. Note how the complexity of the transitions within each layer of the HFSM grows with the number of nodes. The conditions labels are: $C_1 =$ Activity Sit, $C_2 =$ Not Know What to Do, $C_3 =$ Activity Stand Up, $C_4 =$ Activity Sleep, $C_5 =$ Activity Ball Game, $C_6 =$ Ball Close, $C_7 =$ Ball Grasped, $C_8 =$ New User Suggestion, $C_9 =$ Activity Sit, $C_{10} =$ Bumper Pressed.
Figure 3.9: A BT that combines some capabilities of the NAO robot in an interactive and modular way. Note how atomic actions can easily be replaced by more complex sub-BTs.
3.4 Use of BTs in real world applications.

In this section we show the use and the practical advantages of BTs in real robot applications, spanning from autonomous navigation to industrial robotics. We review some projects that have chosen BT as their CA.

3.4.1 BTs in autonomous vehicles

There is no standard CA for autonomous vehicles, however reviewing the CAs used to address the DARPA Grand Challenge, a competition for autonomous vehicles, we note that most teams employed FSMs accurately designed and developed exactly for that challenge [56, 57]. Some of them used a HFSM [38] decomposing the mission task in multiple sub-tasks in a well-defined hierarchy.

![Trucks running iQmatic's software](image)

**Figure 3.10:** Trucks running iQmatic’s software.

---

*Picture courtesy of Scania.com*
iQmatic
iQmatic is a Scania led project that aims at developing a fully autonomous heavy vehicle (truck, bus, etc.) for goods transport; mining; and other industrial applications. The vehicle’s software has to be reusable, maintainable and easy to develop. For these reasons, the iQmatic’s developers chose BTs as the CA for the project. BTs are appreciated in iQmatic for their human readability, that support the design and development of early prototypes; and their maintainability, that makes the editing task easier. Figure 3.10 shows two trucks used in the iQmatic testbed.

3.4.2 BTs in industrial robotics
Industrial robots usually operate in structured environments and, as mentioned in Chapter 1, their CA is designed for a single specific task. Hence classical CAs as FSMs or Petri Nets [42] have found successful applications in the last decades. However, future generations of industrial robots will operate in less structured environments and will collaborate with humans. Several projects lie in this research direction.

![Figure 3.11: Experimental platform of the CoSTAR project](http://cpaxton.github.io/)  

CoSTAR
CoSTAR [47] is a project that aims at developing a software framework that contains tools for composing task plans and allows training robots how to perform complex behaviors for industrial applications that involve human cooperation.

---

9Picture courtesy of [http://cpaxton.github.io/](http://cpaxton.github.io/)
3.4. USE OF BTS IN REAL WORLD APPLICATIONS.

BTs have found successful applications in this project as they simplify the composition of sub-tasks. The order in which the sub-tasks are executed is independent from the sub-task implementation; this allows an easy composition of trees and the composition of larger and larger trees iteratively. Figure 3.11 shows the robotic platform of the project.

Figure 3.11: The robotic platform of the project.

SARAFun

SARAFun is a project that aims at developing a robot-programming framework that enables a non-expert user to program an assembly task on a robot in less than a day. It takes advantages of state of the art techniques in sensory and cognitive abilities, robot control, and planning.

BTs are used to drive the generic actions learned or planned. For the purpose of this project, the CA must be human readable, to allow non-programmers to understand the execution; enable code reusability, to use the same functionality in different projects; and modular, to allow the decomposition of the task in smaller and simpler tasks. For these reasons BTs are a promising choice and it has shown advantages also at the project’s development stage, when the code written by different partners had to be integrated. Figure 3.12 shows an ABB Yumi robot used in the SARAFun testbed.

Figure 3.12: Experimental platform of the SARAFun project.\textsuperscript{10}

\textsuperscript{10}Setup located at CERTH, Thessaloniki, Greece. Picture courtesy of Angeliki Topalidou-Kyniazopoulou.
BTs at Rethink Robotics

BT were at the “heart of the design” when Rethink Robotics developed Intera, a robotics software platform that is the “first-of-its-kind software platform that connects everything from a single robot controller, extending the smart, flexible power of Rethink Robotics’ Sawyer to the entire work cell and simplifying automation with unparalleled ease of deployment.”\textsuperscript{12} Intera is designed with the goal of creating the world’s fastest-to-deploy robot. It fundamentally changes the concepts of integration, making it drastically easier and affordable.

Intera allows robots to respond quickly to changes in the workflow. This makes robots much more attractive for those businesses that still have little automation due to their less-structured assembly lines. An example can be taken from the German magnet manufacturer MS Schramberg\textsuperscript{13}. With less than a day of training, an MS Schramberg’s engineer is able to deploy and train the robots in just an hour. The robots now run 24 hours per day, 6 days per week, and can easily configure complex logic tasks, minimizing the need for human interaction and freeing up employees for more complex tasks.

Intera’s BT defines the sequence of tasks the robot will perform. The tree can be created manually or trained by demonstration. Users can inspect any portion of the BT and make adjustments. The Intera interface (see Figure 3.13) also includes a simulated robot, so a user can run simulations while the program executes the BT. BTs are appreciated in this context because the train-by-demonstration framework builds a BT that is inspectable and modifiable\textsuperscript{14}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{intera_bts.png}
\caption{Intera’s BT (left) and simulation environment (right).\textsuperscript{11}}
\end{figure}

\textsuperscript{11}Picture courtesy of http://www.rethinkrobotics.com/intera/
\textsuperscript{12}http://www.rethinkrobotics.com/news-item/rethink-robotics-releases-intera-5-new-approach-automation/
\textsuperscript{13}http://www.magnete.de/
\textsuperscript{14}http://twimage.net/rodney-brooks-743452002
3.4. USE OF BTS IN REAL WORLD APPLICATIONS.

3.4.3 BTs in the Amazon Picking Challenge

The Amazon Picking Challenge (APC) is an international robot competition. Robots are asked to autonomously retrieve a wide range of products from a shelf and put them into a container. The challenge was conceived with the purpose of strengthening the ties between the industrial and academic robotic research, promoting shared solutions to some open problems in unstructured automation. Over thirty companies and research laboratories from different continents competed in the APC’s preliminary phases. The best performing teams earned the right to compete at the finals. The source codes of the finalists are publicly available.\(^{15}\)

The KTH entry in the final challenge used BTs in both editions (2015 and 2016). BTs were appreciated for their modularity and code reusability, which allowed the integration of different functionalities developed by programmers with different background and coding styles. In the 2015, the KTH entry got the best result out of the four teams competing with PR2 robots.

\(^{15}\)https://github.com/amazon-picking-challenge
3.5 Disadvantages of BTs

In this section we describe some disadvantages of BTs experienced by different BT developers.

**The BT engine can be complex to implement.** The implementation of the BT engine can get complicated using single threaded sequential programming. To guarantee the full functionality of BTs, the tick’s generation and traversal should be executed in parallel with the action execution. However the complexity of the BT engine is paid back with an easier task implementation. The BT engine can be reused and several BT engines are available as off the shelf software libraries.

**Checking all the conditions can be expensive.** A BT needs to check several conditions to implement the closed-loop task execution. In some applications this checking is expensive or not feasible. In those cases a closed-loop execution (using any CA) presents more costs than advantages. However it is still possible to design an open-loop task execution using BTs with memory nodes.

**Sometimes a feed-forward execution is just fine.** In those applications where the robot operates in a structured environment, predictable in space and time, BTs do not have any advantages over simpler CAs.

**BTs are different from FSM.** BTs, despite being easy to understand, require a new mindset when design a task execution. The execution is not focused on states but on conditions and the execution is not event driven but tick driven.

**BTs are less mature.** There is little software available for developing and testing BTs.
Chapter 4

Open Source ROS BT library

In this chapter we present the structure and functionality of the open source ROS BT library developed at KTH by the author of this thesis. The library is publicly available at http://wiki.ros.org/behavior_tree and the developer repository is available at https://github.com/miccol/ROS-Behavior-Tree. A non-ROS version of the library in C++ is available at https://github.com/miccol/Behavior-Tree. The library is currently the most starred ROS BT implementation available in GitHub.¹

4.1 Purpose and Description

The purpose of this library is to offer a free open source ROS BT package. The user defines the tree structure and develops the code of the leaf nodes for their execution and preemption. The leaf nodes can be implemented in the same ROS node as the main tree or as external ROS nodes. External ROS nodes can be implemented using C++ or Python.

4.2 Software Structure

The tree, the non-external leaf nodes, and the tree’s visualization are executed within the same ROS node: the main node. The communication between nodes, with the exception for the external ones, is done using callbacks. Each external node is executed in a separate ROS node and communicates with the tree through actionlib². Both main and external nodes are registered with the same roscore.³

¹Starring a repository allows users to keep track of projects that they find interesting.
²http://wiki.ros.org/actionlib
³The roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system.
4.2.1 Parent-Child communication details

To allow the tick traversal, we need to set up the communication between a node and its parent. A parent node sends a tick to a non-action child calling the child’s method `Tick()` whereas it sends a tick to an action child using a semaphores-based synchronization mechanism. The reason behind these two different communication mechanisms is that action nodes need to run asynchronously with the tree traversal as their executions can take time and must not block the tree’s execution. For this reason, each action node runs in a separate detached thread where the node is in idle unless its execution is requested from the parent. To abort the execution of a child (i.e. whenever a parent node no longer sends ticks to a child) the parent node calls the child’s method `Halt()`. The `Halt()` contains the portion of code (if any) that allows a safe abortion of the action (e.g. the `Halt()` method of the action `dance` moves the robot to a safe and balanced position). For this method there is no difference between action or non-action nodes. This is because we prefer to hold the tree’s execution until all the `Halt()` routines have ended.

4.2.2 Graphical Editor

To relieve the user from coding the structure of a complex BT, we developed a graphical user interface. The editor is available at [http://michelecolledanchise.com/editor](http://michelecolledanchise.com/editor). A Python script is also available to parse, build and execute the BT.

The graphic user interface is based on the Behavior3Editor\(^4\), it is possible to create custom-made leaf nodes and write the code directly into the editor. Figure 4.1 depicts the editor and Figure 4.2 shows the dialog box for creating custom made leaf nodes.

![Figure 4.1: Example of a BT constructed using the graphical editor.](https://github.com/behavior3/behavior3editor)

---

\(^4\)https://github.com/behavior3/behavior3editor
4.2. SOFTWARE STRUCTURE

(a) Dialog box for creating a custom made leaf node in python.

(b) Dialog box for creating a custom made leaf node in C++.

Figure 4.2: Dialog box for creating a custom made leaf nodes.
Chapter 5

Summary of Papers

In this chapter we summarize the publications composing the thesis. Paper A proposes a new functional formulation of BTs and provides a structural analysis of BTs. Paper B shows that BTs generalize both AND-OR Trees and TR programs. Paper C proposes an integrated planning and acting approach that is reactive, safe, and fault tolerant. Paper D presents a tool to measure the performance of a BT and how to optimize the structure of a BT based on an objective function. Finally, paper E shows a model-free approach to automatically synthesize a BT, given a set of actions and sensing capabilities.

Author’s contribution

The thesis’ author contributed to the publications by developing the main part of the theories, writings, implementations and analyses.
A How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture and Decision Trees

In this work we propose a new functional formulation of BTs. The new formulation allows us to investigate a key property of BTs, modularity, using standard tools from robot control theory. The benefits of modularity becomes even clearer when key system properties can be shown to be preserved across compositions of smaller modules into bigger systems. The key properties preserved across composition investigated are efficiency, in terms of time to successful completion, safety, in terms of avoiding particular parts of the state space; and robustness, in terms of large regions of attraction.

In this analysis, we also show that BTs can be seen as generalizations of three classical concepts from the robot control literature: the Subsumption Architectures (see Figure 5.1), Sequential Behavior Compositions, and DTs (see Figure 5.2).

![Diagram](image)

**Figure 5.1:** Example of how BTs generalize Subsumption Architectures.

![Diagram](image)

**Figure 5.2:** Example of how BTs generalize DTs.
B How Behavior Trees Generalize the Teleo-Reactive Paradigm and AND-OR Trees.

In this paper we show how BTs generalize TR programs, and how the result on Universal TR programs in [45] is analogous to the richer continuous state space result on compositions of Finite Time Successful BTs in [A]. We also show that the result on Finite Time Successful BTs is stronger, as it covers a case that can be used as a counter example to the Universal TRs. The analogy result in turn implies that much of the work on TR programs can be translated into BT designs, as in Figure 5.4. We also show how AND-OR Trees can be directly translated into BT feedback executions, as in Figure 5.3. This fact, together with the observation that TR programs and AND-OR Trees are quite different, shows that BTs are not equivalent to the other two structures, but in fact richer, as it generalizes both.

![Figure 5.3: Example of how a BTs generalize AND-OR Trees.](image)

Equal(pos,loc) → Idle
Heading Towards (loc) → Go Forwards
(else) → Rotate

(a) TR program of example.

(b) BT generalizing the TR program in (a).

Figure 5.4: Example of how a BTs generalize TR programs.

In this paper, we propose an approach that produces reactive execution plans in the form of BTs. The BT includes reactivity, in the sense that the execution plan is responsive to changes in the environment. The BT also supports iterative plan refinement where the original BT can be extended to include new sub-plans.

Within the AI community, there has been an increased interest in the combination of planning and acting, [17, 18, 27]. In particular, [17] describes two key open challenges:

• “Hierarchically organized deliberation. This principle goes beyond existing hierarchical planning techniques; its requirements and scope are significantly different. The actor performs its deliberation online”

• “Continual planning and deliberation. The actor monitors, refines, extends, updates, changes and repairs its plans throughout the acting process, using both descriptive and operational models of actions.”

Similarly, the recent book [18] describes the need for an agent that “reacts to events and extends, updates, and repairs its plan on the basis of its perception” (see Figure 5.5). Finally, [18] also notes that most of the current work in action planning yields a static plan, i.e., a sequence of actions that brings the system from the initial state to the goal state. Its execution is usually represented as a classical FSM. However, due to the environmental changes, the effect of an action can be unexpected. This may lead to situations where the agent replans from scratch on a regular basis, which can be expensive in terms of both time and computational load. In this paper we show how to iteratively extend the BT to achieve a goal.

Figure 5.5: A simple example scenario where a moving obstacle requires the replanning. The goal is to place the green cube $C$ onto the goal region $G$. The fact that the sphere $S$ might be blocking the path must be handled. The nominal plan (left) is $MoveTo(C)\rightarrow Pick(C)\rightarrow MoveTo(G)\rightarrow Drop()$ then the sphere suddenly blocks the path (right). After replanning, the plan is $MoveTo(S)\rightarrow Push(S)\rightarrow MoveTo(C) \rightarrow Pick(C)\rightarrow MoveTo(G)\rightarrow Drop()$ then the sphere moves away before being pushed, requiring a new plan.
In this work we model the reliability and efficiency of reactive robot plan executions, in terms of execution times and success/failure probabilities. We estimate these performance measures for plans that are encoded and executed using BTs and we show how to derive an equivalent BT that maximizes an objective function based on the aforementioned performances (see Figure 5.6 for an example). The analysis is done by defining Stochastic Behavior Trees (SBTs) in a way that is somewhat analogous to how Stochastic Petri Nets (SPNs) relate to Petri Nets (PNs). Having defined the SBTs, we describe the interaction of a BT node with its children in terms of a Discrete Time Markov Chain (DTMC). This enables us to propagate performance estimates from one level in the tree to the next in a bottom up approach. Applying the scheme in a recursive fashion then makes it possible to compute the properties of an arbitrarily complex BT, and reuse estimates in a modular way when new BTs are formed by combining existing ones. The proposed approach builds upon assumptions regarding the performance of the atomic action controllers that are at the leaves of the BTs. We propose a mapping from BTs to DTMC to aggregate these performance measures from one level of the tree to the next. The fact that BTs have a recursive tree structure thus allows us to compute the performance of the complete overall task in a modular way. Finally, we verify our analytical results using massive Monte Carlo simulations, and provide an illustrative example of the results for a complex robotic task.

Figure 5.6: A given BT and its performance.
E Learning of Behavior Trees for Autonomous Agents

In this paper, we study the problem of automatically synthesizing a BT, given a set of low level actions (such as move, jump, shoot) and high level sensing in terms of a number of binary conditions (such as obstacle at position \( x \), enemy at position \( y \)). This is a particular instance of the Automated Planning problem [37] with the goal of generating a BT. We focus our study on a model-free version, where there is no prior information about the environment and the effects of actions.

Exploiting the fact that BTs generalize AND-OR Trees and also provide very natural chromosome mappings for Genetic Programming (GP), we combine the long-term performance of GP with a greedy element and a clear separation between conditions and actions. The connection to AND-OR Trees makes sure that the learning outcome is not black box, but perfectly human readable. Finally, earlier results on BTs enable us to provide certain safety guarantees for the resulting system.

Using a testing environment, we compare our approach to alternative methods for learning BTs and one for learning FSMs. The evaluation shows that the clear separation between conditions and actions, together with the greedy element, has a significant positive impact on performance and solution complexity.

The testing environment is the Mario AI benchmark, shown in Figure 5.7. In the benchmark the agent has input in terms of receptive fields, with two binary conditions (enemy/no-enemy and obstacle/no-obstacle) for each field. The actions available are: go left/right, jump, crouch, and shoot. The problem is now to learn an action switching policy, mapping conditions to actions. There is no prior knowledge of what the conditions mean, or what the actions do and the only measure of progress is the game score.

![Simple Scenario](image1.png) ![Complex Scenario](image2.png)

Figure 5.7: Mario AI benchmark. The agent can observe only a small subset of the environment, namely the receptive field.
In this thesis we investigated the following research question: *What are the advantages and disadvantages of using BTs as a CA in robotics?* In order to do that, we proposed a formal functional description of BTs that allowed us to analyze properties of BTs such as safety, robustness, efficiency and reliability. We then compared BTs with other common CAs used in robotics. We finally investigated the use of BTs for planning and learning. We made the following observations:

**BTs generalize many successful CAs:** BTs generalize many successful CAs: Sequential Behavior Compositions, TR programs, Subsumption Architectures, AND-OR Trees, and DTs. In Papers A and B we showed these generalizations and compared BTs with the other CAs mentioned.

**BTs are good for analysis:** The BT representation allows system properties to be analyzed and designed in a modular way. In Paper A we showed how some properties are preserved across compositions. The propositions investigated are safety and robustness. In Paper D we showed how to compute the properties of an arbitrarily complex BT in a modular way. The properties investigated were reliability and efficiency.

**BTs are good for planning:** A planning algorithm can benefit from the BT representation. In Paper C we showed that BTs allow us to blend planning and acting in a way that is reactive, hierarchical, safe, and fault tolerant.

**BTs are good for learning:** In Paper E we showed that the BT representation allows us to combine the long-term performance of genetic programming (with a greedy element) with the convenient structure of AND-OR trees.

**BTs are not magic wands:** BTs present some disadvantages. BTs are still less mature than other common CAs used in robotics. BTs require a new mindset when designing a task execution policy, and for structured environments, the classical choice of FSMs work just as well.
Moreover, the following additional observations were made:

**Closed-loop task execution has advantages:** In most real world applications, it is not possible to fully model or predict the environmental changes. Then, the task execution must be done in a closed-loop fashion, monitoring the environment and action outcomes.

**Architectural design principles are often disregarded:** From a purely theoretical standpoint, FSMs can describe (almost) any conceivable behavior. However when it comes to the implementation stages, an unfortunate choice of architecture can significantly impede the system development.

**Modularity:** Modularity is the key property for a complex, maintainable and reusable system. Modularity in this sense goes far beyond placing a subsystem into a bigger one. It is beneficial to have a uniform interface between task ordering and action execution and to have a convenient modular structure. It is preferable if the implementation of actions is independent from the action ordering choices.

**Algorithms can benefit from good data structures:** A tree structure often improves the performance of algorithms. For example, automatic synthesis in terms of learning or planning techniques, benefits from the structure of BTs.

**Planning frameworks gravitate around the CA:** A good CA can improve planning performance. A change of direction in how we model the action ordering could lead to important results in automated planning.

As discussed in Chapter 3 BTs are still somewhat young, and further investigations are needed to make BTs a well-established CA in robotics. An interesting direction lies in the integration of the result of blended planning and acting (Paper C) with the learning of action templates. In particular we are interested in learning the preconditions and effects for each action to enable subsequent planning. Paper E shows how BTs are suitable for learning. Note however that the challenges we will face are slightly different that the one addressed in Paper E. We are not able to define an objective function that is maximized when the preconditions are found. Roughly speaking Paper E, finds a sufficient set of preconditions of an action for a given task. We will be interested in finding necessary and sufficient set of preconditions of an action for any task. We believe that the modularity of BTs will allow us to learn individual pieces for preconditions and effects to characterize the action templates. The learning process could be supervised by a domain expert, taking inspirations from [1, 24] or carried out by interacting with the environment, taking inspirations from [59]. The learning process can be integrated in a sequential fashion (i.e. the approach learns first the action templates and then blends planning and acting,) or in a interleaved fashion (the learning of preconditions and effects gets blended with planning and acting). We believe that the particular structure and execution of BTs effectively allows blending learning, acting, and planning.
Bibliography


