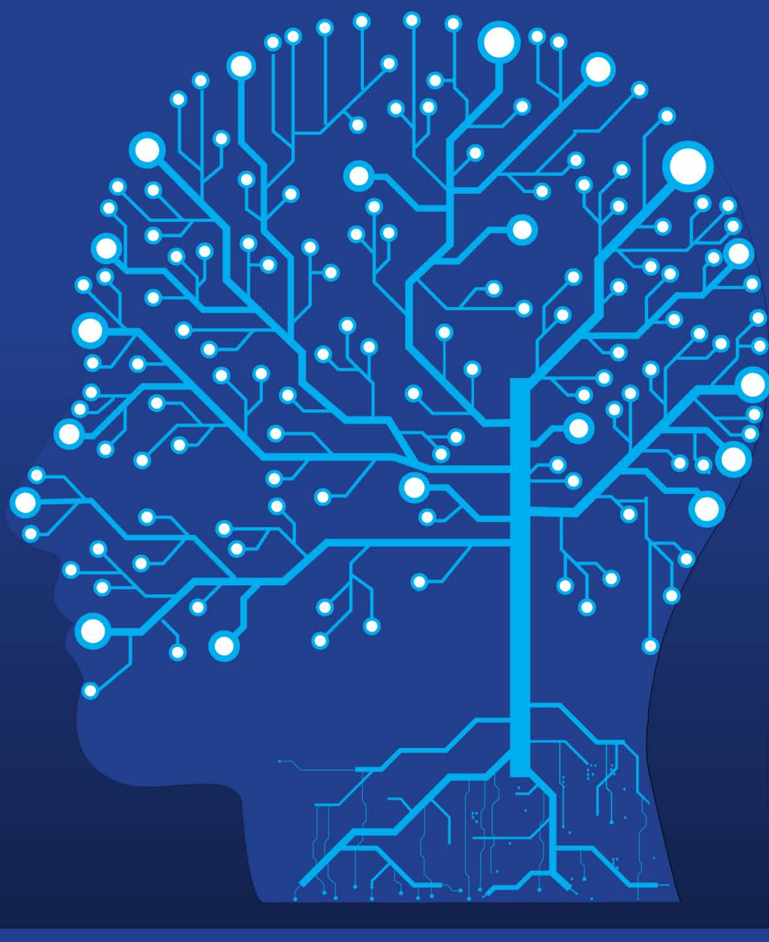
Chapman & Hall/CRC Artificial Intelligence and Robotics Series

Behavior Trees in Robotics and AI

AN INTRODUCTION



CPD CERTIFIED
The CPD Certification Service

Michele Colledanchise
Petter Ögren

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Behavior Trees in Robotics and AI

Chapman & Hall/CRC
Artificial Intelligence and Robotics Series
Series Editor: Roman Yampolskiy

Contemporary Artificial Intelligence
Richard E. Neapolitan

The Virtual Mind
Designing the Logic to Approximate Human Thinking
Niklas Hageback

Intelligent Autonomy of UAVs
Advanced Missions and Future Use
Yasmina Bestaoui Sebbane

Artificial Intelligence
With an Introduction to Machine Learning, Second Edition
Richard E. Neapolitan, Xia Jiang

Artificial Intelligence and the Two Singularities
Calum Chace

Behavior Trees in Robotics and Al
An Introduction
Michele Colledanchise, Petter Ögren

For more information about this series please visit:
https://www.crcpress.com/Chapman--HallCRC-Artificial-Intelligence-and-Robotics-Series/book-series/ARTILRO

Behavior Trees in Robotics and AI

An Introduction

Michele Colledanchise
Petter Ögren

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20180410

International Standard Book Number-13: 978-1-138-59373-2 (Hardback)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To Paola
MC

To Gustav, Julia and Viktoria
PÖ

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

Preface

This book is about behavior trees (BTs), a way to structure the behavior, or more precisely the task switching, of an artificial agent such as a robot or a non-player character in a computer game.

The problems regarding task switching are very similar in robotics and virtual agent design. However, these problems have received far more attention from game developers than from the robotics community. One reason for this is that designing individual tasks, such as grasping, localization, and mapping are research topics of their own in robotics, while being trivial in a virtual world. A game character does not need to worry about real-world physics and mechanics, and is free to directly access the positions of itself and other objects in some world coordinates, or setting the position of an object to be the same as its hand. Thus it is not a coincidence that BTs were invented in the game development community, while robotics researchers were busy making robots able to execute individual tasks.

Modularity and reactivity, the two key features of BTs compared to other task switching structures, are also very important in game development. A computer game is an example of a very large software development project, and the importance of modularity in software development is well known. Reactivity, or the ability of agents to react to external events in general, and the actions of the human players in particular, is also of key importance. But although switching structures have not been a focus area in robotics, the need for a modular and reactive switching structure will grow as robots become more capable in terms of individual task.

This book will guide you to the subject of BTs from simple topics, such as semantics and design principles, to complex topics, such as learning and task planning. For each topic we provide a set of examples, ranging from simple illustrations to realistic complex behaviors, to enable the reader to successfully combine theory with practice.

TARGET AUDIENCE

The target audience of this book is very broad and includes both students and professionals interested in modeling complex behaviors for robots, game characters, or other AI agents. The readers can choose at which depth and pace they want to learn the subject, depending on their needs and background.

WEB MATERIAL

The website[1] accompanying this book contains the source code of several examples, as well as a Behavior Tree library with a Graphical User Interface to let the readers create their own behaviors and test them in a game or robotics setting.

COMMENTS AND SUGGESTIONS

Please use the webpage to send us your feedback in terms of comments or suggestions.

ABOUT THE AUTHORS

Michele Colledanchise is currently a postdoctoral researcher in the iCub Facility at the Italian Institute of Technology, Genoa, Italy. He received his Ph.D. degree in computer science from the Royal Institute of Technology (KTH), Stockholm, Sweden, in 2017. In the spring of 2016, he visited the Control and Dynamical Systems, Californa Institute of Technology (Caltech), Pasadena, CA. His research interests include control systems, system architectures, and automated planning, with a strong focus on robotic applications.

Petter Ögren was born in Stockholm, Sweden, in 1974. He received his M.S. degree in engineering physics and Ph.D. degree in applied mathematics from the Royal Institute of Technology (KTH), Stockholm, Sweden, in 1998 and 2003, respectively. In the fall of 2001, he visited the Mechanical Engineering Department, Princeton University, Princeton, NJ. From 2003 to 2012, he worked as a senior scientist and deputy research director in Autonomous Systems at the Swedish Defence Research Agency (FOI). He is currently an Associate Professor at the Robotics, Perception and Learning lab (RPL) at KTH. His research interests include robot control architectures and multi-agent coordination.

[1] https://btirai.github.io/

QUOTES ON BEHAVIOR TREES

I'm often asked why I chose to build the SDK with behavior trees instead of finite state machines. The answer is that behavior trees are a far more expressive tool to model behavior and control flow of autonomous agents. [2]

Jonathan Ross
Head of Jibo SDK

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 [3]

Mike Weldon
Disney, Pixar

[...]. Sure you could build the very same behaviors with a finite state machine (FSM). But anyone who has worked with this kind of technology in industry knows how fragile such logic gets as it grows. A finely tuned hierarchical FSM before a game ships is often a temperamental work of art not to be messed with! [4]

Alex J. Champandard
Editor in Chief & Founder AiGameDev.com,
Senior AI Programmer Rockstar Games

Behavior trees offer a good balance of supporting goal-oriented behaviors and reactivity. [5]

Daniel Broder
Unreal Engine developer

The main advantage [of Behavior Trees] 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, [2].

Andrew Bagnell et al.
Carnegie Mellon University.

[2] https://developers.jibo.com/blog/the-jibo-sdk-reaching-out-beyond-the-screen

[3] http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

[4] http://aigamedev.com/open/article/fsm-age-is-over/

[5] https://forums.unrealengine.com/showthread.php?6016-Behavior-Trees-What-and-Why

CHAPTER 1

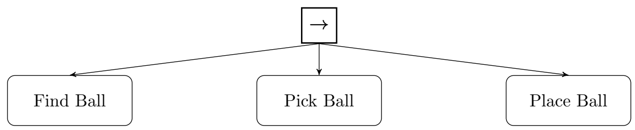
What Are Behavior Trees?

A behavior tree (BT) is a way to structure the switching between different tasks[1] in an autonomous agent, such as a robot or a virtual entity in a computer game. An example of a BT performing a pick and place task can be seen in Figure 1.1(a). As will be explained, BTs are a very efficient way of creating complex systems that are both *modular* and *reactive*. These properties are crucial in many applications, which has led to the spread of BT from computer game programming to many branches of AI and robotics.

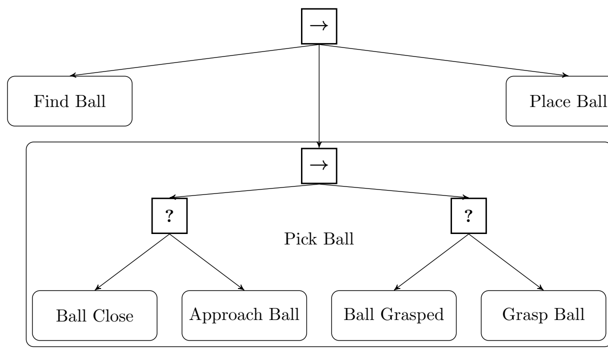
In this book, we will first give an introduction to BTs, in the present chapter. Then, in Chapter 2 we describe how BTs relate to, and in many cases generalize, earlier switching structures, or control architectures as they are often called. These ideas are then used as a foundation for a set of efficient and easy-to-use design principles described in Chapter 3. Then, in Chapter 4 we describe a set of important extensions to BTs. Properties such as safety, robustness, and efficiency are important for an autonomous system, and in Chapter 5 we describe a set of tools for formally analyzing these using a state space formulation of BTs. With the new analysis tools, we can formalize the descriptions of how BTs generalize earlier approaches in Chapter 6. Then, we see how BTs can be automatically generated using planning, in Chapter 7 and learning, in Chapter 8. Finally, we describe an extended set of tools to capture the behavior of Stochastic BTs, where the outcomes of actions are described by probabilities, in Chapter 9. These tools enable the computation of both success probabilities and time to completion.

In this chapter, we will first tell a brief history of BTs in Section 1.1, and explain the core benefits of BTs, in Section 1.2, then in Section 1.3 we will describe how a BT works. Then, we will create a simple BT for the computer game Pac-Man in Section 1.4 and a more sophisticated BT for a mobile manipulator in Section 1.5. We finally describe the usage of BT in a number of applications in Section 1.6.

[1]Assuming that an activity can somehow be broken down into reusable sub-activities called *tasks* sometimes also denoted as *actions* or *control modes*.



(a) A high level BT carrying out a task consisting of first finding, then picking and finally placing a ball.



(b) The Action Pick Ball from the BT in Figure 1.1(a) is expanded into a sub-BT. The Ball is approached until it is considered close, and then the Action Grasp is executed until the Ball is securely grasped.

Figure 1.1: Illustrations of a BT carrying out a pick and place task with different degrees of detail. The execution of a BT will be described in Section 1.3.

1.1 A SHORT HISTORY AND MOTIVATION OF BTs

BTs were developed in the computer game industry, as a tool to increase modularity in the control structures of non-player character (NPCs) [9, 31, 32, 39, 43, 60]. In this billion-dollar industry, modularity is a key property that enables reuse of code, incremental design of functionality, and efficient testing.

In games, the control structures of NPCs were often formulated in terms of finite state machines (FSMs). However, just as Petri nets [48] provide an alternative to FSMs that supports design of *concurrent* systems, BTs provide an alternative view of FSMs that supports design of *modular* systems.

Following the development in the industry, BTs have now also started to receive attention in academia [2, 5, 11, 20, 27, 30, 35, 37, 38, 50, 55, 67].

At Carnegie Mellon University, BTs have been used extensively to do robotic manipulation [2, 20]. The fact that modularity is the key reason for using BTs is clear from the following quote from [2]: “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" [27] and allowed "end-users to visually create programs with the same amount of complexity and power as traditionally written programs" [56]. Furthermore, BTs have been proposed as a key component in brain surgery robotics due to their "flexibility, reusability, and simple syntax" [30].

1.2 WHAT IS WRONG WITH FSMs? THE NEED FOR REACTIVENESS AND MODULARITY

Many autonomous agents need to be both reactive and modular. By reactive we mean the ability to quickly and efficiently react to changes. We want a robot to slow down and avoid a collision if a human enters into its planned trajectory and we want a virtual game character to hide, flee, or fight, if made aware of an approaching enemy. By modular, we mean the degree to which a system's components may be separated into building blocks, and recombined [23]. We want the agent to be modular, to enable components to be developed, tested, and reused independently of one another. Since complexity grows with size, it is beneficial to be able to work with components one at a time, rather than the combined system.

FSMs have long been the standard choice when designing a task switching structure [45, 59], and will be discussed in detail in Section 2.1, but here we make a short description of the unfortunate tradeoff between reactivity and modularity that is inherent in FSMs. This tradeoff can be understood in terms of the classical Goto statement that was used in early programming languages. The Goto statement is an example of a so-called *one-way control transfer*, where the execution of a program jumps to another part of the code and continues executing from there. Instead of *one-way control transfers*, modern programming languages tend to rely on *two-way control transfers* embodied in, e.g., function calls. Here, execution jumps to a particular part of the code, executes it, and then returns to where the function call was made. The drawbacks of *one-way control transfers* were made explicit by Edsger Dijkstra in his paper *Goto statement considered harmful* [15], where he states that "The Goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program." Looking back at the state transitions in FSMs, we note that they are indeed *one-way control transfers*. This is where the tradeoff between reactivity and modularity is created. For the system to be reactive, there needs to be many transitions between components, and many transitions means many *one-way control transfers* which, just as Dijkstra noted, harms modularity by being an "invitation to make a mess of one's program." If, for example, one component is removed, every transition to that component needs to be revised. As will be seen, BTs use *two-way control transfers*, governed by the internal nodes of the trees.

Using BTs instead of FSMs to implement the task switching, allows us to describe the desired behavior in modules as depicted in Figure 1.1(a). Note that in the next section we will describe how BTs work in detail, so these figures are just meant to give a first glimpse of BTs, rather than the whole picture.

A behavior is often composed of a sequence of sub-behaviors that are task independent, meaning that while creating one 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 1.1(b). BTs are executed in a particular way, which will be described in the following section, that allows the behavior to be carried out reactively. For example, the BT in Figure 1.1 executes the sub-behavior *Place Ball*, but also verifies that the ball is still at a known location and securely grasped. If, due to an external event, the ball slips out out of the grasp, then the robot will abort the sub-behavior *Place Ball* and will re-execute the sub-behavior *Pick Ball* or *Find Ball* according to the current situation.

1.3 CLASSICAL FORMULATION OF BTs

At the core, BTs are built from a small set of simple components, just as many other powerful concepts, but throughout this book, we will see how this simple formalism can be used to create very rich structures, in terms of both applications and theory.

Formally speaking, a BT is a directed rooted tree where the internal nodes 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 1.2-1.4.

A BT starts its execution from the root node that generates signals called *Ticks* with a given frequency. These signals allow the execution of a node and are propagated to one or several of the children of the ticked node. A node is executed if and only if it receives Ticks. The child immediately returns *Running* to the parent, if its execution is under way, *Success* if it has achieved its goal, or *Failure* otherwise.

In the classical formulation, there exist four categories of control flow nodes (Sequence, Fallback, Parallel, and Decorator) and two categories of execution nodes (Action and Condition). They are all explained below and summarized in Table 1.1.

The Sequence node executes Algorithm 1.1, which corresponds to routing 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 own 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 symbol of the Sequence node is a box containing the label "→", shown in Figure 1.2.

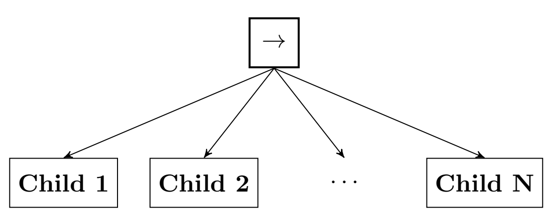


Figure 1.2: Graphical representation of a Sequence node with *N* children.

Algorithm 1.1: Pseudocode of a Sequence node with N children

```
Function Tick()
    for i ← 1 to N do
        childStatus ← child(i).Tick()
        if childStatus = Running then
            return Running
        else if childStatus = Failure then
            return Failure
    return Success
```

The Fallback node[2] executes Algorithm 1.2, which corresponds to routing 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 own 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 symbol of the Fallback node is a box containing the label “?”, shown in Figure 1.3.

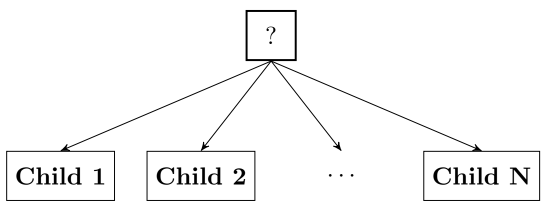


Figure 1.3: Graphical representation of a Fallback node with N children.

Algorithm 1.2: Pseudocode of a Fallback node with N children

```
Function Tick()
    for i ← 1 to N do
        childStatus ← child(i).Tick()
        if childStatus = Running then
            return Running
        else if childStatus = Success then
            return Success
    return Failure
```

[2]Fallback nodes are sometimes also called *selector* or *priority selector* nodes.

The Parallel node executes Algorithm 1.3, which corresponds to routing 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 symbol of the Parallel node is a box containing the label "$\rightrightarrows$", shown in Figure 1.4.

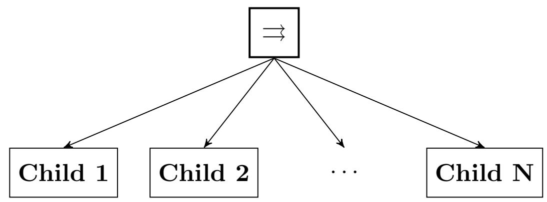
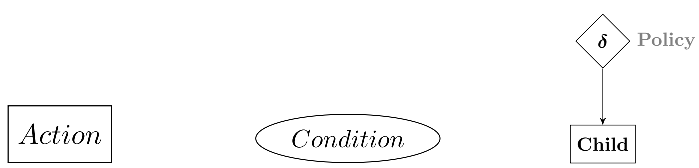


Figure 1.4: Graphical representation of a Parallel node with N children.

Algorithm 1.3: Pseudocode of a Parallel node with N children and success threshold M

Function *Tick()*
  forall $i \leftarrow 1$ **to** N **do**
    childStatus[i] $\leftarrow$ *child(i)*.`Tick()`
  if $\Sigma_{i:childStatus[i]=Success} 1 = N$ **then**
    return *Success*
  else if $\Sigma_{i:childStatus[i]=Failure} 1 > 0$ **then**
    return *Failure*
  else
    return *Running*



(a) Action node. The label describes the action performed.

(b) Condition node. The label describes the condition verified.

(c) Decorator node. The label describes the user defined policy.

Figure 1.5: Graphical representation of Action (a), Condition (b), and Decorator (c) nodes.

When an Action node receives Ticks, it executes a command, as in Algorithm 1.4. It returns *Success* if the action is successfully completed or *Failure* if the action has failed. While the action is ongoing it returns *Running*. An Action node is shown in Figure 1.5(a).

Algorithm 1.4: Pseudocode of an Action node

```
Function Tick()
    ExecuteCommand()
    if action-succeeded then
        return Success
    else if action-failed then
        return Failure
    else
        return Running
```

Algorithm 1.5: Pseudocode of a Condition node

```
Function Tick()
    if condition-true then
        return Success
    else
        return Failure
```

Note that in a robotic system, an action execution might need to run at a *higher frequency* than the BT itself. For instance, a force controlled manipulator opening a drawer might need a frequency of 100-1000 Hz, while the BT deciding what to do after the drawer is opened might be fine with 10 Hz. This is achieved by letting actions run until they either succeed/fail or explicitly receive an abort command as a result of no more Ticks being received, instead of just letting actions wait for the next Tick, as described in Algorithm 1.4. Thus, robotics BT libraries, such as YARP-BT[3] and ROS-BT[4], use an implementation specific routine for safely aborting actions, while computer games and simulators, such as Unreal Engine[5] and Pygame[6], usually do not have this problem and execute a command at each Tick as in Algorithm 1.4.

When a Condition node receives Ticks, it checks a proposition, as in Algorithm 1.5. It returns *Success* or *Failure* depending on if the proposition holds or not. Note that a Condition node never returns a status of *Running*. A Condition node is shown in Figure 1.5(b).

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 rule. 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

[3] https://github.com/miccol/YARP-Behavior-Trees
[4] http://wiki.ros.org/behavior_tree
[5] https://docs.unrealengine.com/en-us/Engine/AI/BehaviorTrees/
[6] http://www.pygame.org/project-owyl-1004-.html

lets the child run for T seconds then, if the child is still Running, the Decorator returns *Failure* without ticking the child. The symbol of the Decorator is a rhombus, as in Figure 1.5(c).

Node type	Symbol	Succeeds	Fails	Running
Fallback	$\boxed{?}$	If one child succeeds	If all children fail	If one child returns Running
Sequence	$\boxed{\rightarrow}$	If all children succeed	If one child fails	If one child returns Running
Parallel	$\boxed{\rightrightarrows}$	If $\geq M$ children succeed	If $> N-M$ children fail	else
Action	$\boxed{\text{text}}$	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	$\Diamond$	Custom	Custom	Custom

Table 1.1: The node types of a BT.

1.3.1 Execution example of a BT

Consider the BT in Figure 1.6 designed to make an agent look for a ball, approach it, grasp it, proceed to a bin, and place the ball in the bin. This example will illustrate the execution of the BT, including the reactivity when another (external) agent takes the ball from the first agent, making it switch to looking for the ball and approaching it again. When the execution starts, the Ticks traverse the BT reaching the condition node *Ball Found*. The agent 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 1.7(a)). While executing this action, the agent sees the ball with the camera. In this new situation the agent 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 1.7(b)). Then the agent eventually reaches the ball, picks it up and goes towards the bin (see Figure 1.7(c)). When an external agent moves the ball from the hand of the first agent 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 1.7(d)).

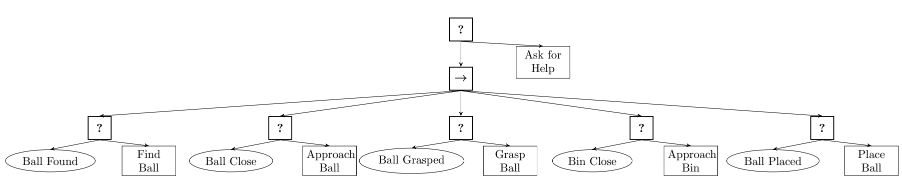
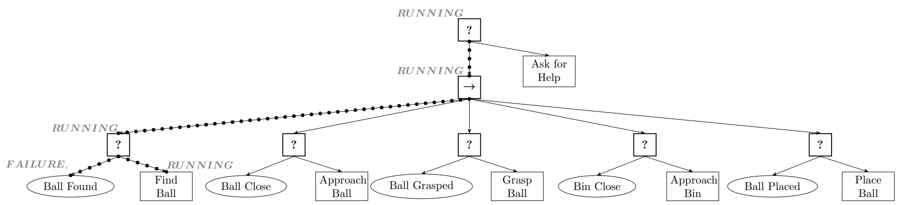
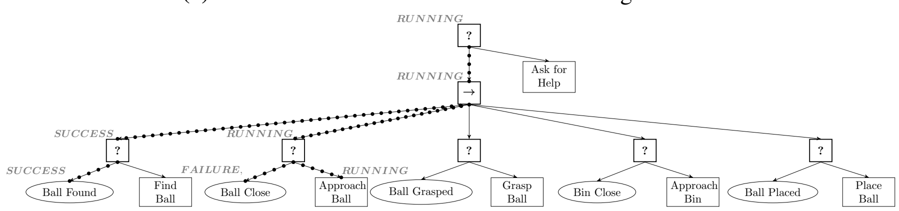


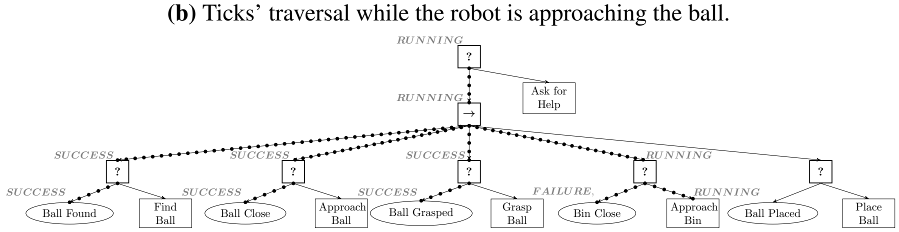
Figure 1.6: BT encoding the behavior of Example 2.1.



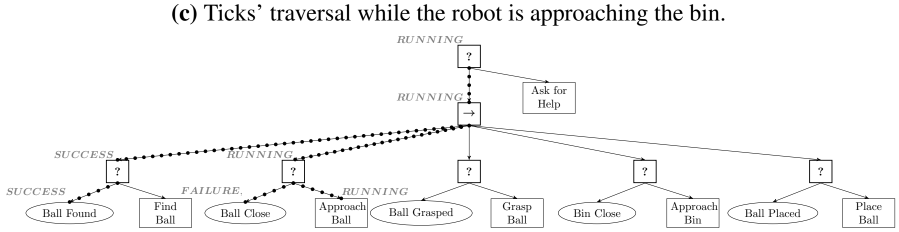
(a) Ticks' traversal when the robot is searching the ball.



(b) Ticks' traversal while the robot is approaching the ball.



(c) Ticks' traversal while the robot is approaching the bin.



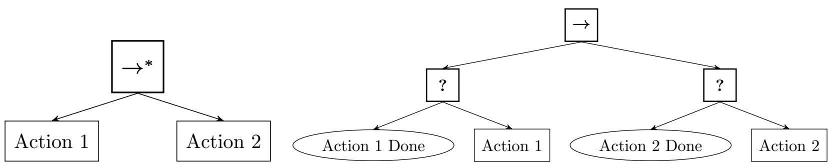
(d) Ticks' traversal while the robot is approaching the ball again (because it was removed from the hand).

Figure 1.7: Visualization of the Ticks' traversal in different situations, as explained in Section 1.3.1.

1.3.2 Control flow nodes with memory

As seen in the example above, to provide reactivity 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 knows that a child, once executed, does not need to be re-executed.

Memory nodes [43] 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 book, 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 "$\rightarrow^*$"). The memory is cleared when the parent node returns either *Success* or *Failure*, so that at 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 1.8. Hence nodes with memory can be considered to be syntactic sugar.



(a) Sequence composition with memory.

(b) BT that emulates the execution of the Sequence composition with memory using nodes without memory.

Figure 1.8: Relation between memory and memory-less BT nodes.

Some BT implementations, such as the one described in [43], do not include the *Running* return status. 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 [43]. A non-reactive BT can be seen as a BT with only memory nodes.

As reactivity is one of the key strengths of BTs, the non-reactive BTs are of limited use.

1.4 CREATING A BT FOR PAC-MAN FROM SCRATCH

In this section, we create a set of BTs of increasing complexity for playing the game Pac-Man. The source code of all the examples is publicly available and editable[7]. We use a clone of the Namco Pac-Man computer game depicted in Figure 1.9[8].

In the testbed, a BT controls the agent, Pac-Man, through a maze containing two ghosts, a large number of pills, including two so-called power pills. The goal of the game is to consume all the pills, without being eaten by the ghosts. The power pills are such that, if eaten, Pac-Man receives temporary super powers, and is able to eat the ghosts. After a given time the effect of the power pill wears off, and the ghosts can again eat Pac-Man. When a ghost is eaten, it returns to the center box where it is regenerated and becomes dangerous again. Edible ghosts change color, and then flash to signal when they are about to become dangerous again.

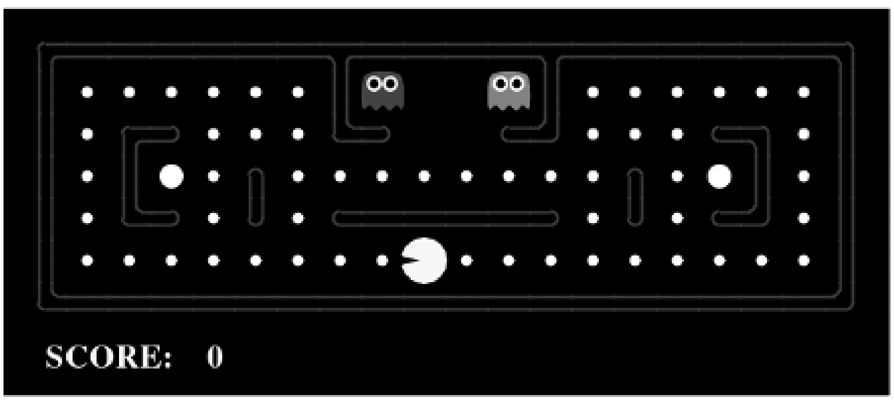


Figure 1.9: The game Pac-Man for which we will design a BT. There exists maps of different complexity.

The simplest behavior is to let Pac-Man ignore the ghosts and just focus on eating pills. This is done using a greedy action *Eat Pills* as in Figure 1.10.

Figure 1.10: BT for the simplest non-random behavior, *Eat Pills*, which maximizes the number of pills eaten in the next time step.

The simple behavior described above ignores the ghosts. To take them into account, we can extend the previous behavior by adding an *Avoid Ghosts* Action to be executed whenever the condition *Ghost Close* is true. This Action will greedily

[7] https://btirai.github.io/

[8] The software was developed at UC Berkeley for educational purposes. More information available at: http://ai.berkeley.edu/project_overview.html

maximize the distance to all ghosts. The new Action and condition can be added to the BT as depicted in Figure 1.11. The resulting BT will switch between Eat Pills and Avoid Ghost depending on whether Ghost Close returns Success or Failure.

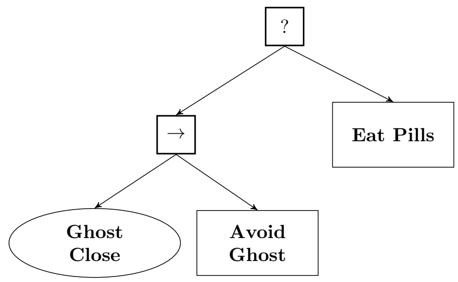


Figure 1.11: If a Ghost is Close, the BT will execute the Action Avoid Ghost, else it will run Eat Pills.

The next extension we make is to take the power pills into account. When Pac-Man eats a power pill, the ghosts are edible, and we would like to chase them, instead of avoiding them. To do this, we add the condition *Ghost Scared* and the Action *Chase Ghost* to the BT, as shown in Figure 1.12. *Chase Ghost* greedily minimizes the distance to the closest edible ghost. Note that we only start chasing the ghost if it is close, otherwise we continue eating pills. Note also that all extensions are modular, without the need to rewire the previous BT.

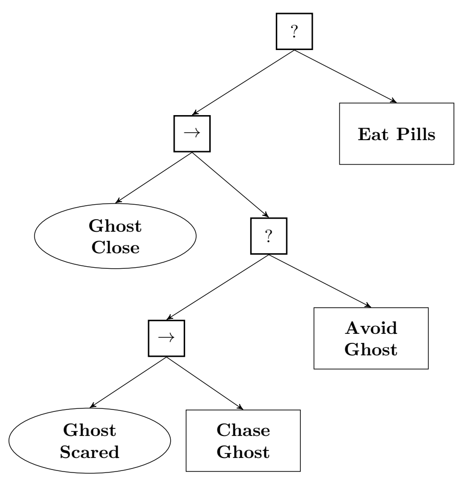


Figure 1.12: BT for the Combative Behavior.

With this incremental design, we have created a basic artificial intelligence (AI) for playing Pac-Man, but what if we want to make a world class Pac-Man AI? You could add additional nodes to the BT, such as moving towards the power pills when being chased, and stop chasing ghosts when they are blinking and soon will transform into normal ghosts. However, much of the fine details of Pac-Man lies in considerations of the Maze geometry, choosing paths that avoid dead ends and possible capture by multiple ghosts. Such spatial analysis is probably best done inside the actions, e.g., making Avoid Ghosts take dead ends and ghost positions into account. The question of what functionality to address in the BT structure, and what to take care of inside the actions is open, and must be decided on a case-by-case basis, as discussed in Section 3.7.

1.5 CREATING A BT FOR A MOBILE MANIPULATOR ROBOT

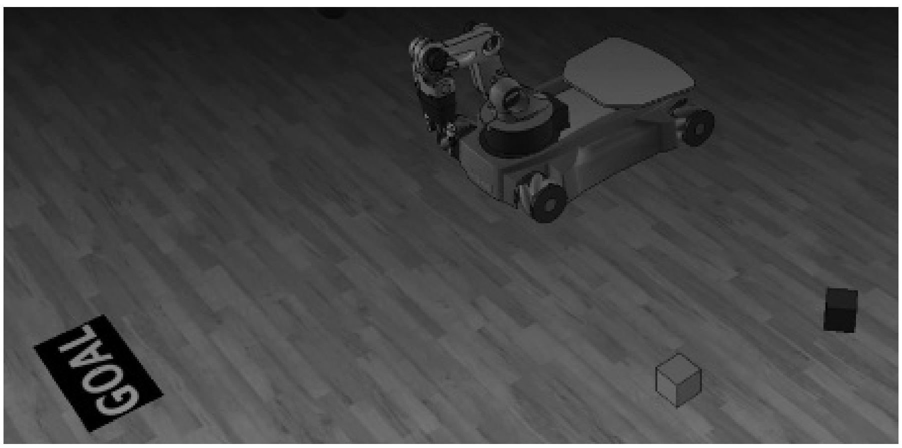


Figure 1.13: The Mobile Manipulator for which we will design a BT.

In this section, we create a set of BTs of increasing complexity for controlling a mobile manipulator. The source code of all the examples is publicly available and editable.[9] We use a custom-made testbed created in the V-REP robot simulator depicted in Figure 1.13.

In the testbed, a BT controls a mobile manipulator robot, a youBot, on a flat surface. In the scenario, several colored cubes are lying on a flat surface. The goal is to move the green cube to the goal area while avoiding the other cubes. The youBot's grippers are such that the robot is able to pick and place the cubes if the robot is close enough.

The simplest possible BT is to check the goal condition *Green Cube on Goal*. If this condition is satisfied (i.e., the cube is on the goal) the task is done, if it is not satisfied the robot needs to *place the cube* onto the goal area. To correctly execute the Action *Place Cube*, two conditions need to hold: the robot *is holding the green cube* and the robot *is close to the goal area*. The behavior described so far can be encoded in the BT in Figure 1.14. This BT is able to place the green cube on the goal area if and only if the robot is close to the goal area with the green cube grasped.

[9] https://btirai.github.io/

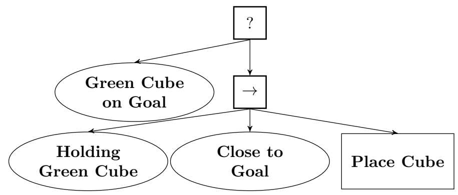
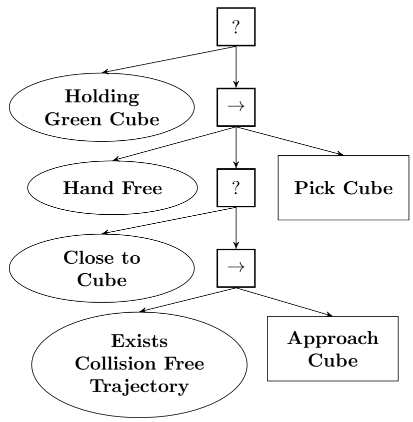


Figure 1.14: BT for the simple Scenario.

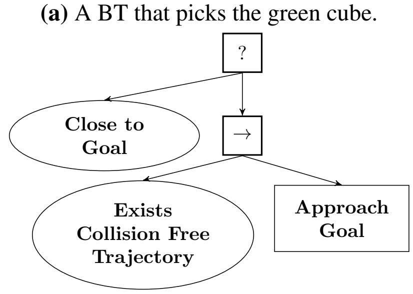
Now, thanks to the modularity of BTs, we can separately design the BTs needed to satisfy the two lower conditions in Figure 1.14, i.e., the BT needed to grasp the green cube and the BT needed to reach the goal area. To grasp the green cube, the robot needs to have the *hand free* and be *close to the cube*. If it is not close, it approaches as long as a collision free trajectory exists. This behavior is encoded in the BT in Figure 1.15(a). To reach the goal area, we let the robot simply *Move To the Goal* as long as a *collision free trajectory exists*. This behavior is encoded in the BT in Figure 1.15(b).

Now we can extend the simple BT in Figure 1.14 above by replacing the two lower conditions in Figure 1.14 with the two BTs in Figure 1.15. The result can be seen in Figure 1.16. Using this design, the robot is able to place the green cube in the goal area as long as there exists a collision free trajectory to the green cube and to the goal area.

We can continue to incrementally build the BT in this way to handle more situations, for instance removing obstructing objects to ensure that a *collision free trajectory exists*, and dropping things in the hand to be able to pick the green cube up.



(a) A BT that picks the green cube.



(b) A BT that reaches the goal region.

Figure 1.15: Illustrations of a BT carrying out the sub-tasks of picking the green cube and reaching the goal area.

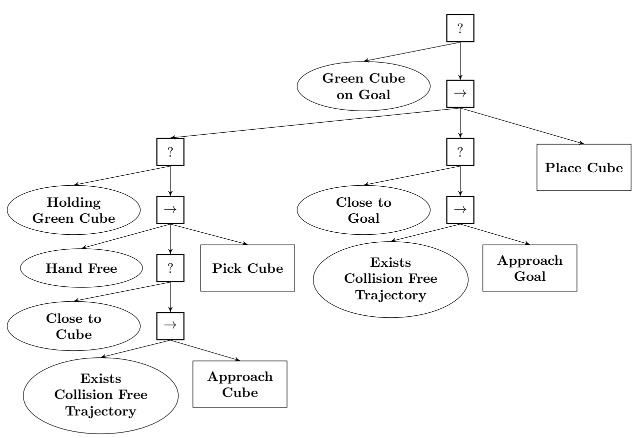


Figure 1.16: Final BT resulting from the aggregation of the BTs in Figures. 1.14-1.15.

1.6 USE OF BTs IN ROBOTICS AND AI

In this section we describe the use of BTs in a set of real robot applications and projects, spanning from autonomous driving to industrial robotics.

1.6.1 BTs in autonomous vehicles

There is no standard control architecture for autonomous vehicles; however, reviewing the architectures used to address the DARPA Grand Challenge, a competition for autonomous vehicles, we note that most teams employed FSMs designed and developed exactly for that challenge [71,72]. Some of them used a hierarchical finite state machine (HFSM) [45] decomposing the mission task in multiple sub-tasks in a hierarchy. As discussed in Section 1.2, there is reason to believe that using BTs instead of FSMs would be beneficial for autonomous driving applications.

Figure 1.17: Trucks running the Scania iQMatic's software.

iQMatic is a Scania-led project that aims at developing a fully autonomous heavy truck 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 control architecture for the project. BTs are appreciated in iQMatic for their human readability, supporting the design and development of early prototypes; and their maintainability, making the editing task easier. Figure 1.17 shows two trucks used in the iQMatic project.

1.6.2 BTs in industrial robotics

Industrial robots usually operate in structured environments and their control architecture is designed for a single specific task. Hence classical architectures such as FSMs or Petri nets [48] have found successful applications in the last decades. However, future generations of collaborative industrial robots, so-called cobots, will operate in less structured environments and collaborate closely with humans. Several research projects explore this research direction.

Figure 1.18: Experimental platform of the CoSTAR project.[10]

CoSTAR [56] is a project that aims at developing a software framework that contains tools for industrial applications that involve human cooperation. The use cases include non-trained operators composing task plans, and training robots to perform complex behaviors. 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 enables easy composition of trees and the iterative composition of larger and larger trees. Figure 1.18 shows one of the robotic platforms of the project.

SARAFun[11] is a project that aims at developing a robot-programming framework that enables a non-expert user to program an assembly task from scratch on a robot in less than a day. It takes advantage of state-of-the-art techniques in sensory and cognitive abilities, robot control, and planning.

BTs are used to execute the generic actions learned or planned. For the purpose of this project, the control architecture must be human readable, enable code reuse, and

[10]Picture courtesy of http://cpaxton.github.io/

[11]H2020 project id: 644938

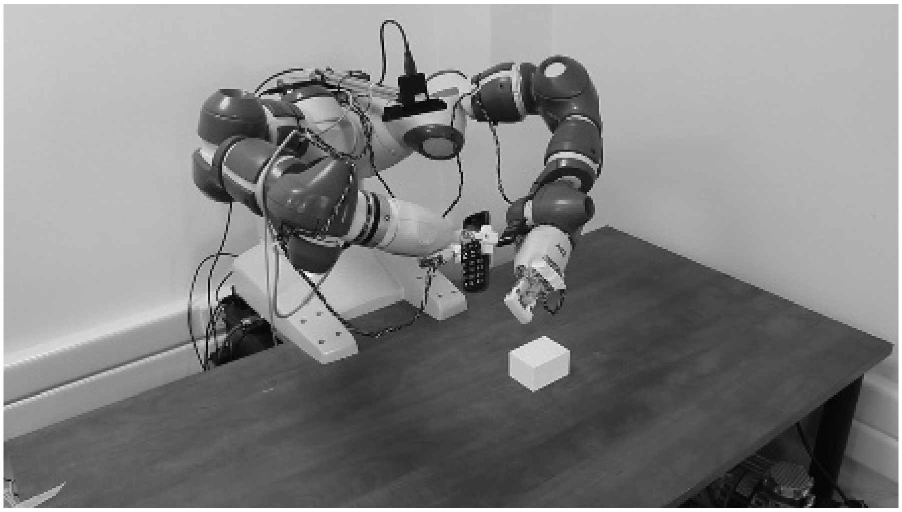


Figure 1.19: Experimental platform of the SARAFun project.[12]

modular. BTs have created advantages also during the development stage, when the code written by different partners had to be integrated. Figure 1.19 shows an ABB Yumi robot used in the SARAFun testbed.

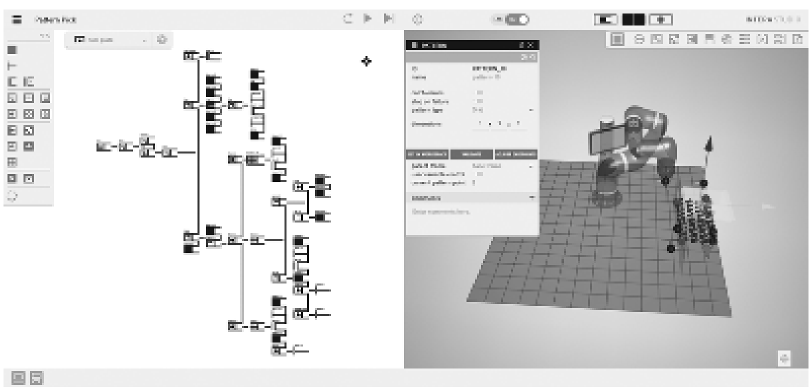


Figure 1.20: Intera's BT (left) and simulation environment (right).[13]

Rethink Robotics released its software platform Intera in 2017, with BTs at the "heart of the design." Intera claims to be a "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 automa-

[12]Setup located at CERTH/ITI. Picture courtesy of Angeliki Topalidou-Kyniazopoulou.
[13]Picture courtesy of Rethink Robotics.

tion with unparalleled ease of deployment."[14] It is designed with the goal of creating the world's fastest-to-deploy robot and fundamentally changing the concepts of integration, making it drastically easier and more affordable.

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 1.20) 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 easily inspectable and modifiable.[15]

1.6.3 BTs in the Amazon picking challenge

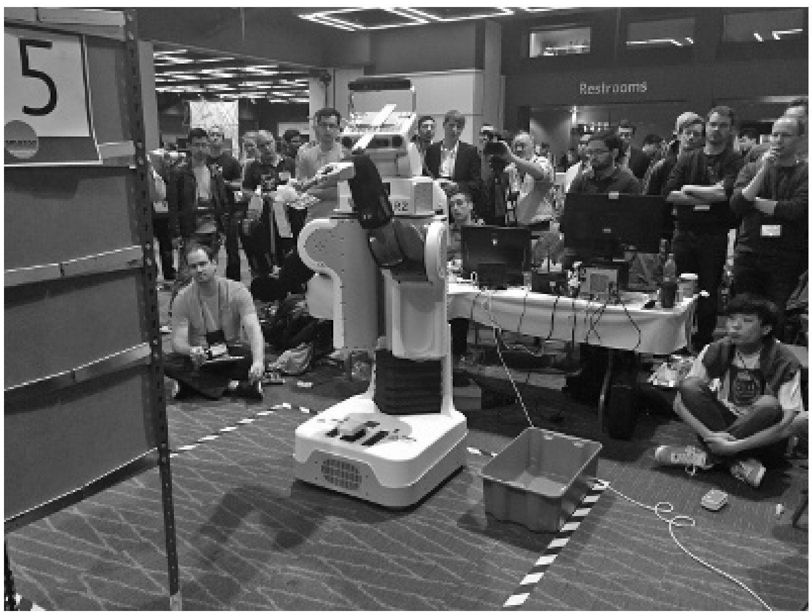


Figure 1.21: The KTH entry in the Amazon picking challenge at ICRA 2015.

The Amazon picking challenge (APC) is an international robot competition. Robots need 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 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 APC's preliminary

[14] http://www.rethinkrobotics.com/news-item/rethink-robotics-releases-intera-5-new-approach-automation/

[15] http://twimage.net/rodney-brooks-743452002

phases. The best performing teams earned the right to compete at the finals and the source codes of the finalists were made publicly available.

The KTH entry in the final challenge used BTs in both 2015 and 2016, see Figure 1.21. 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 2015, the KTH entry got the best result out of the four teams competing with PR2 robots.

1.6.4 BTs inside the social robot JIBO

JIBO is a social robot that can recognize faces and voices, tell jokes, play games, and share information. It is intended to be used in homes, providing the functionality of a tablet, but with an interface relying on speech and video instead of a touch screen. JIBO has been featured in *Time* Magazine's Best Inventions of 2017.[16] BTs are a fundamental part of the software architecture of JIBO[17], including an open software development kit (SDK) inviting external contributors to develop new skills for the robot.

[16] http://time.com/5023212/best-inventions-of-2017/
[17] https://developers.jibo.com/docs/behavior-trees.html

Bibliography

[1] David Aha, Matthew Molineaux, and Marc Ponsen. Learning to Win: Case-based Plan Selection in a Teal-time Strategy Game. *Case-based Reasoning Research and Development*, pages 5–20, 2005.

[2] J. Andrew (Drew) Bagnell, Felipe Cavalcanti, Lei Cui, Thomas Galluzzo, Martial Hebert, Moslem Kazemi, Matthew Klingensmith, Jacqueline Libby, Tian Yu Liu, Nancy Pollard, Mikhail Pivtoraiko, Jean-Sebastien Valois, and Ranqi Zhu. An Integrated System for Autonomous Robotics Manipulation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2955–2962, October 2012.

[3] Andrew G. Barto and Sridhar Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[4] Scott Benson and Nils J. Nilsson. Reacting, Planning, and Learning in an Autonomous Agent. In *Machine Intelligence 14*, pages 29–64. Citeseer, 1995.

[5] Iva Bojic, Tomislav Lipic, Mario Kusek, and Gordan Jezic. Extending the JADE Agent Behaviour Model with JBehaviourtrees Framework. In *Agent and Multi-Agent Systems: Technologies and Applications*, pages 159–168. Springer, 2011.

[6] R. Brooks. A Robust Layered Control System for a Mobile Robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, 1986.

[7] R.A. Brooks. Elephants Don't Play Chess. *Robotics and Autonomous Systems*, 6(1-2):3–15, 1990.

[8] Robert R. Burridge, Alfred A. Rizzi, and Daniel E. Koditschek. Sequential Composition of Dynamically Dexterous Robot Behaviors. *The International Journal of Robotics Research*, 18(6):534–555, 1999.

[9] A.J. Champandard. Understanding Behavior Trees. *AIGameDev. com*, Volume 6, 2007.

[10] Michele Colledanchise, Diogo Almeida, and Petter Ögren. Towards Blended Reactive Planning and Acting using Behavior Trees. *arXiv Preprint arXiv:1611.00230*, 2016.

[11] Michele Colledanchise, Alejandro Marzinotto, and Petter Ögren. Performance Analysis of Stochastic Behavior Trees. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, June 2014.

[12] Michele Colledanchise and Petter Ögren. How Behavior Trees Generalize the Teleo-Reactive Paradigm and And-Or-Trees. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 424–429. IEEE, 2016.

[13] 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.

[14] Michele Colledanchise, Ramviyas Parasuraman, and Petter Ögren. Learning of Behavior Trees for Autonomous Agents. *IEEE Transactions on Games, DOI 10.1109/TG.2018.2816806*, 2018.

[15] Edsger W. Dijkstra. Letters to the Editor: Go To Statement Considered Harmful. *Commun. ACM*, 11:147–148, March 1968.

[16] A.F. Filippov and F.M. Arscott. *Differential Equations with Discontinuous Righthand Sides: Control Systems*. Mathematics and its Applications. Kluwer Academic Publishers, 1988.

[17] Gonzalo Flórez-Puga, Marco Gomez-Martin, Belen Diaz-Agudo, and Pedro Gonzalez-Calero. Dynamic Expansion of Behaviour Trees. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference. AAAI Press*, pages 36–41, 2008.

[18] Marc Freese, Surya Singh, Fumio Ozaki, and Nobuto Matsuhira. Virtual Robot Experimentation Platform V-REP: A Versatile 3D Robot Simulator. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 51–62, 2010.

[19] Zhiwei Fu, Bruce L Golden, Shreevardhan Lele, S Raghavan, and Edward A Wasil. A Genetic Algorithm-based Approach for Building Accurate Decision Trees. *INFORMS Journal on Computing*, 15(1):3–22, 2003.

[20] Thomas Galluzzo, Moslem Kazemi, and Jean-Sebastien Valois. BART - Behavior Architecture for Robotic Tasks. Technical report, 2013.

[21] Ramon Garcia-Martinez and Daniel Borrajo. An Integrated Approach of Learning, Planning, and Execution. *Journal of Intelligent and Robotic Systems*, 29(1):47–78, 2000.

[22] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Backward-forward Search for Manipulation Planning. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 6366–6373. IEEE, 2015.

[23] J.K. Gershenson, G.J. Prasad, and Y. Zhang. Product Modularity: Definitions and Benefits. *Journal of Engineering design*, 14(3):295–313, 2003.

[24] Malik Ghallab, Dana Nau, and Paolo Traverso. The Actor's View of Automated Planning and Acting: A Position Paper. *Artif. Intell.*, 208:1–17, March 2014.

[25] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[26] Gerhard Gubisch, Gerald Steinbauer, Martin Weiglhofer, and Franz Wotawa. A Teleo-reactive Architecture for Fast, Reactive and Robust Control of Mobile Robots. In *New Frontiers in Applied Artificial Intelligence*, pages 541–550. Springer, 2008.

[27] Kelleher R. Guerin, Colin Lea, Chris Paxton, and Gregory D. Hager. A Framework for End-User Instruction of a Robot Assistant for Manufacturing. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[28] Blake Hannaford, Danying Hu, Dianmu Zhang, and Yangming Li. Simulation Results on Selector Adaptation in Behavior Trees. *arXiv Preprint arXiv:1606.09219*, 2016.

[29] David Harel. Statecharts: A Visual Formalism For Complex Systems. *Science of computer programming*, 8(3):231–274, 1987.

[30] Danying Hu, Yuanzheng Gong, Blake Hannaford, and Eric J. Seibel. Semi-autonomous Simulated Brain Tumor Ablation with Raven II Surgical Robot using Behavior Tree. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[31] Damian Isla. Handling Complexity in the Halo 2 AI. In *Game Developers Conference*, 2005.

[32] Damian Isla. Halo 3 - Building a Better Battle. In *Game Developers Conference*, 2008.

[33] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical Task and Motion Planning in the Now. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1470–1477. IEEE, 2011.

[34] Sergey Karakovskiy and Julian Togelius. The Mario AI Benchmark and Competitions. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):55–67, 2012.

[35] Andreas Klökner. Interfacing Behavior Trees with the World Using Description Logic. In *AIAA conference on Guidance, Navigation and Control, Boston*, 2013.

[36] Martin Levihn, Leslie Pack Kaelbling, Tomas Lozano-Perez, and Mike Stilman. Foresight and Reconsideration in Hierarchical Planning and Execution. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 224–231. IEEE, 2013.

[37] C.U. Lim, R. Baumgarten, and S. Colton. Evolving Behaviour Trees for the Commercial Game DEFCON. *Applications of Evolutionary Computation*, pages 100–110, 2010.

[38] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. Towards a Unified Behavior Trees Framework for Robot Control. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, June 2014.

[39] M. Mateas and A. Stern. A Behavior Language for Story-based Believable Agents. *IEEE Intelligent Systems*, 17(4):39–47, Jul. 2002.

[40] Joshua McCoy and Michael Mateas. An Integrated Agent for Playing Real-Time Strategy Games. In *AAAI*, volume 8, pages 1313–1318, 2008.

[41] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *The Bell System Technical Journal*, 34(5):1045–1079, Sept. 1955.

[42] Bill Merrill. Ch 10, Building Utility Decisions into Your Existing Behavior Tree. *Game AI Pro. A Collected Wisdom of Game AI Professionals*, 2014.

[43] Ian Millington and John Funge. *Artificial Intelligence for Games*. CRC Press, 2009.

[44] Tom M. Mitchell. *Machine Learning. WCB*, volume 8. McGraw-Hill Boston, MA:, 1997.

[45] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. Junior: The Stanford Entry in the Urban Challenge. *Journal of field Robotics*, 25(9):569–597, 2008.

[46] Edward F. Moore. Gedanken-experiments on Sequential Machines. *Automata studies*, 34:129–153, 1956.

[47] Seyed R. Mousavi and Krysia Broda. *Simplification Of Teleo-Reactive sequences*. Imperial College of Science, Technology and Medicine, Department of Computing, 2003.

[48] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[49] Dana S. Nau, Malik Ghallab, and Paolo Traverso. Blended Planning and Acting: Preliminary Approach, Research Challenges. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 4047–4051. AAAI Press, 2015.

[50] M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon. Evolutionary Behavior Tree Approaches for Navigating Platform Games. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2016.

[51] Nils J. Nilsson. Teleo-reactive Programs for Agent Control. *JAIR*, 1:139–158, 1994.

[52] J.R. Norris. *Markov Chains*. Number no. 2008 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998.

[53] Sergio Ocio. *A Dynamic Decision-making Model for Videogame AI Systems, Adapted to Players*. PhD thesis, Ph. D. diss., Department of Computer Science, University of Oviedo, Spain, 2010.

[54] Sergio Ocio. Adapting AI Behaviors To Players in Driver San Francisco: Hinted-Execution Behavior Trees. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[55] Petter Ögren. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. In *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.

[56] Chris Paxton, Andrew Hundt, Felix Jonathan, Kelleher Guerin, and Gregory D Hager. CoSTAR: Instructing Collaborative Robots with Behavior Trees and Vision. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 564–571. IEEE, 2017.

[57] Renato de Pontes Pereira and Paulo Martins Engel. A Framework for Constrained and Adaptive Behavior-based Agents. *arXiv Preprint arXiv:1506.02312*, 2015.

[58] Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution. In *Proceedings of the 2011 International Conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplications'11, Berlin, Heidelberg, 2011. Springer.

[59] Matthew Powers, Dave Wooden, Magnus Egerstedt, Henrik Christensen, and Tucker Balch. The Sting Racing Team's Entry to the Urban Challenge. In *Experience from the DARPA Urban Challenge*, pages 43–65. Springer, 2012.

[60] Steve Rabin. *Game AI Pro*, chapter 6. The Behavior Tree Starter Kit. CRC Press, 2014.

[61] Ingo Rechenberg. Evolution Strategy. *Computational Intelligence: Imitating Life*, 1, 1994.

[62] Glen Robertson and Ian Watson. Building Behavior Trees from Observations in Real-time Strategy Games. In *Innovations in Intelligent SysTems and Applications (INISTA), 2015 International Symposium on*, pages 1–7. IEEE, 2015.

[63] Günter Rudolph. Convergence Analysis of Canonical Genetic Algorithms. *Neural Networks, IEEE Transactions on*, pages 96–101, 1994.

[64] I. Sagredo-Olivenza, P. P. Gomez-Martin, M. A. Gomez-Martin, and P. A. Gonzalez-Calero. Trained Behavior Trees: Programming by Demonstration to Support AI Game Designers. *IEEE Transactions on Games*, PP(99):1–1, 2017.

[65] Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. *Imitation in Animals and Artifacts*, chapter Learning to Fly, page 171. MIT Press, 2002.

[66] Kirk Y. W. Scheper, Sjoerd Tijmons, Coen C. de Visser, and Guido C. H. E. de Croon. Behaviour Trees for Evolutionary Robotics. *CoRR*, abs/1411.7267, 2014.

[67] Alexander Shoulson, Francisco M Garcia, Matthew Jones, Robert Mead, and Norman I Badler. Parameterizing Behavior Trees. In *Motion in Games*. Springer, 2011.

[68] William J Stewart. *Probability, Markov Chains, Queues, and Simulation: the Mathematical Basis of Performance Modeling*. Princeton University Press, 2009.

[69] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*, volume 1. MIT press Cambridge, 1998.

[70] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[71] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. Autonomous Driving in Urban Environments: Boss and the Urban Challenge. In *Journal of Field Robotics*, volume 25, pages 425–466. Wiley Online Library, 2008.

[72] Chris Urmson, J Andrew Bagnell, Christopher R Baker, Martial Hebert, Alonzo Kelly, Raj Rajkumar, Paul E Rybski, Sebastian Scherer, Reid Simmons, Sanjiv Singh, et al. *Tartan Racing: A Multi-modal Approach to the Darpa Urban Challenge*. 2007.

[73] Blanca Vargas and E. Morales. Solving Navigation Tasks with Learned Teleo-Reactive Programs. In *Proceedings of IEEE International Conference on Robots and Systems (IROS)*, 2008.

[74] Ben G. Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala. Reactive Planning Idioms for Multi-scale Game AI. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 115–122. IEEE, 2010.

[75] Ben George Weber, Michael Mateas, and Arnav Jhala. Building Human-Level AI for Real-Time Strategy Games. In *AAAI Fall Symposium: Advances in Cognitive Systems*, volume 11, page 01, 2011.