

Michele Colledanchise and Petter Ögren

Behavior Trees in Robotics and AI

An Introduction

Contents

1	What are Behavior Trees?	3
1.1	A Short History and Motivation of BTs	4
1.2	What is wrong with FSMs? The Need for Reactiveness and Modularity	5
1.3	Classical Formulation of BTs	6
1.3.1	Execution Example of a BT	9
1.3.2	Control Flow Nodes with Memory	11
1.4	Creating a BT for Pac-Man from Scratch	12
1.5	Creating a BT for a Mobile Manipulator Robot	14
1.6	Use of BTs in Robotics and AI	15
1.6.1	BTs in autonomous vehicles	16
1.6.2	BTs in industrial robotics	18
1.6.3	BTs in the Amazon Picking Challenge	20
1.6.4	BTs inside the social robot JIBO	21
2	How Behavior Trees Generalize and Relate to Earlier Ideas	23
2.1	Finite State Machines	23
2.1.1	Advantages and disadvantages	24
2.2	Hierarchical Finite State Machines	24
2.2.1	Advantages and disadvantages	24
2.2.2	Creating a FSM that works like a BTs	29
2.2.3	Creating a BT that works like a FSM	32
2.3	Subsumption Architecture	32
2.3.1	Advantages and disadvantages	33
2.3.2	How BTs Generalize the Subsumption Architecture	33
2.4	Teleo-Reactive programs	33
2.4.1	Advantages and disadvantages	34
2.4.2	How BTs Generalize Teleo-Reactive Programs	35
2.5	Decision Trees	35
2.5.1	Advantages and disadvantages	36
2.5.2	How BTs Generalize Decision Trees	36

2.6	Advantages and Disadvantages of Behavior Trees	37
2.6.1	Advantages	37
2.6.2	Disadvantages	42
3	Design principles	45
3.1	Improving Readability using Explicit Success Conditions	45
3.2	Improving Reactivity using Implicit Sequences	46
3.3	Handling Different Cases using a Decision Tree Structure	47
3.4	Improving Safety using Sequences	47
3.5	Creating Deliberative BTs using Backchaining	49
3.6	Creating Un-Reactive BTs using Memory Nodes	51
3.7	Choosing the Proper Granularity of a BT	52
3.8	Putting it all together	53
4	Extensions of Behavior Trees	57
4.1	Utility BTs	58
4.2	Stochastic BTs	58
4.3	Temporary Modification of BTs	59
4.4	Other extensions of BTs	61
4.4.1	Dynamic Expansion of BTs	61
5	Analysis of Efficiency, Safety, and Robustness	63
5.1	Statespace Formulation of BTs	63
5.2	Efficiency and Robustness	66
5.3	Safety	70
5.4	Examples	73
5.4.1	Robustness and Efficiency	74
5.4.2	Safety	77
5.4.3	A More Complex BT	81
6	Formal Analysis of How Behavior Trees Generalize Earlier Ideas	83
6.1	How BTs Generalize Decision Trees	83
6.2	How BTs Generalize the Subsumption Architecture	86
6.3	How BTs Generalize Sequential Behavior Compositions	88
6.4	How BTs Generalize the Teleo-Reactive approach	89
6.4.1	Universal Teleo-Reactive programs and FTS BTs	91
7	Behavior Trees and Automated Planning	93
7.1	The Planning and Acting (PA-BT) approach	94
7.1.1	Algorithm Overview	98
7.1.2	The Algorithm Steps in Detail	98
7.1.3	Comments on the Algorithm	101
7.1.4	Algorithm Execution on Graphs	103
7.1.5	Algorithm Execution on an existing Example	104
7.1.6	Reactivity	111
7.1.7	Safety	112

Contents	III
7.1.8 Fault Tolerance	113
7.1.9 Complex Execution on Realistic Robots	114
7.2 Planning using A Behavior Language (ABL)	127
7.2.1 An ABL Agent	127
7.2.2 The ABL Planning Approach	130
7.2.3 Brief Results of a Complex Execution in StarCraft	134
7.3 Comparison between PA-BT and ABL	135
8 Behavior Trees and Machine Learning	137
8.1 Genetic Programming Applied to BTs	137
8.2 The GP-BT Approach	139
8.2.1 Algorithm Overview	140
8.2.2 The Algorithm Steps in Detail	143
8.2.3 Pruning of Ineffective Subtrees	144
8.2.4 Experimental Results	144
8.2.5 Other Approaches using GP applied to BTs	149
8.3 Reinforcement Learning applied to BTs	149
8.3.1 Summary of Q-Learning	150
8.3.2 The RL-BT Approach	150
8.3.3 Experimental Results	151
8.4 Comparison between GP-BT and RL-BT	153
8.5 Learning from Demonstration applied to BTs	153
9 Stochastic Behavior Trees	155
9.1 Stochastic BTs	155
9.1.1 Markov Chains and Markov Processes	156
9.1.2 Formulation	159
9.2 Transforming a SBT into a DTMC	164
9.2.1 Computing Transition Properties of the DTMC	166
9.3 Reliability of a SBT	169
9.3.1 Average sojourn time	169
9.3.2 Mean Time To Fail and Mean Time To Succeed	170
9.3.3 Probabilities Over Time	172
9.3.4 Stochastic Execution Times	172
9.3.5 Deterministic Execution Times	173
9.4 Examples	175
10 Concluding Remarks	187
References	188

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.”¹

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.”²

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!”³

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.”⁴

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.

¹ <https://developers.jibo.com/blog/the-jibo-sdk-reaching-out-beyond-the-screen>

² http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

³ <http://aigamedev.com/open/article/fsm-age-is-over/>

⁴ <https://forums.unrealengine.com/showthread.php?6016-Behavior-Trees-What-and-Why>

Chapter 1

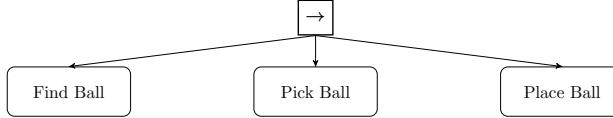
What are Behavior Trees?

A Behavior Tree (BT) is a way to structure the switching between different tasks¹ 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 Fig. 1.1a. 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.

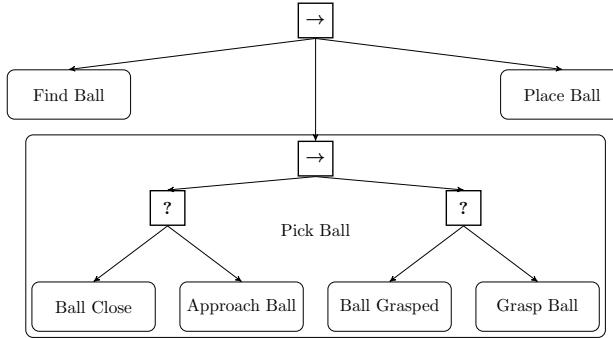
In this book, we will first give an introduction to BTs, in the present chapter. Then, in Chapter 2 we describe how BTs relate to, and in many cases generalize, earlier switching structures, or control architectures as they are often called. These ideas are then used as a foundation for a set of efficient 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 *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 Fig. 1.1a 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.

Fig. 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 Characters (NPCs) [31, 9, 39, 32, 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 [37, 50, 67, 5, 55, 38, 2, 35, 11, 20, 30, 27].

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 [59, 45], and will be discussed in detail in Chapter 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 continue 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 1.1a. 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 1.1b. 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 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.

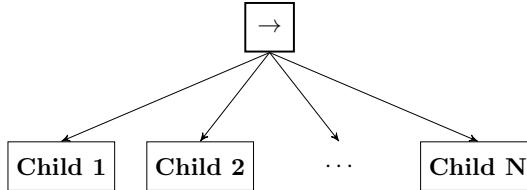
A BT starts its execution from the root node that generates signals that allow the execution of a node called *ticks* with a given frequency, which are sent to its children. A node is executed if and only if it receives ticks. The child immediately returns *Running* to the parent, if its execution is 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, 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.

The Fallback node² executes Algorithm 2, which corresponds to routing the ticks to its children from the left until it finds a child that returns either *Success* or *Run-*

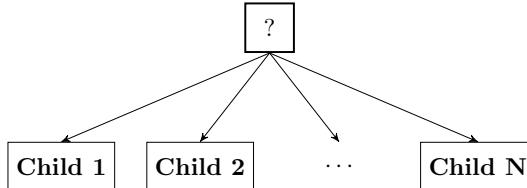
² Fallback nodes are sometimes also called *selector* or *priority selector* nodes.

**Fig. 1.2:** Graphical representation of a Sequence node with N children.**Algorithm 1:** Pseudocode of a Sequence node with N children

```

1 for  $i \leftarrow 1$  to  $N$  do
2   |   childStatus  $\leftarrow$  Tick (child( $i$ ))
3   |   if childStatus = Running then
4   |   |   return Running
5   |   else if childStatus = Failure then
6   |   |   return Failure
7 return Success
  
```

ning, 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 the Fallback node is a box containing the label “?”, shown in Figure 1.3.

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

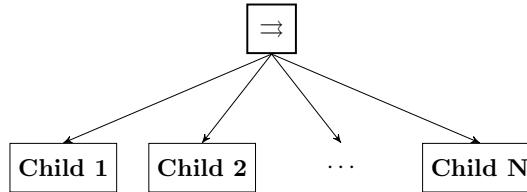
The Parallel node executes Algorithm 3, which corresponds to routing the ticks to all its children and it returns *Success* if M children return *Success*, it returns *Failure* if $N - M + 1$ children return *Failure*, and it returns *Running* otherwise, where N is the number of children and $M \leq N$ is a user defined threshold. The symbol of the Parallel node is a box containing the label “ \Rightarrow ”, shown in Figure 1.4.

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 1.5a.

Algorithm 2: Pseudocode of a Fallback node with N children

```

1 for  $i \leftarrow 1$  to  $N$  do
2   |  $childStatus \leftarrow \text{Tick}(child(i))$ 
3   | if  $childStatus = \text{Running}$  then
4     |   | return  $\text{Running}$ 
5   | else if  $childStatus = \text{Success}$  then
6     |   | return  $\text{Success}$ 
7 return  $\text{Failure}$ 
```

**Fig. 1.4:** Graphical representation of a Parallel node with N children.**Algorithm 3:** Pseudocode of a Parallel node with N children and success threshold M

```

1 for  $i \leftarrow 1$  to  $N$  do
2   |  $childStatus(i) \leftarrow \text{Tick}(child(i))$ 
3   | if  $\sum_{i:childStatus(i)=\text{Success}} 1 \geq M$  then
4     |   | return  $\text{Success}$ 
5   | else if  $\sum_{i:childStatus(i)=\text{Failure}} 1 > N - M$  then
6     |   | return  $\text{Failure}$ 
7 return  $\text{Running}$ 
```



(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.

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

When it receives ticks, a Condition node checks a proposition. 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 1.5b.

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

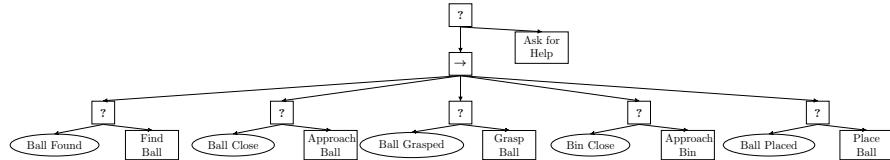
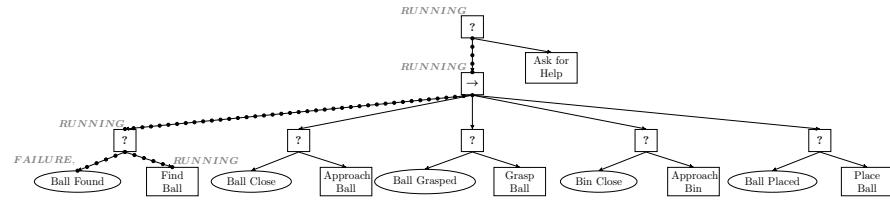
ticks the child according to some predefined rule. For example, an *invert* decorator inverts the *Success/Failure* status of the child; a *max-N-tries* decorator only lets its child fail N times, then always returns *Failure* without ticking the child; a *max-T-sec* decorator lets the child run for 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 1.5c.

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

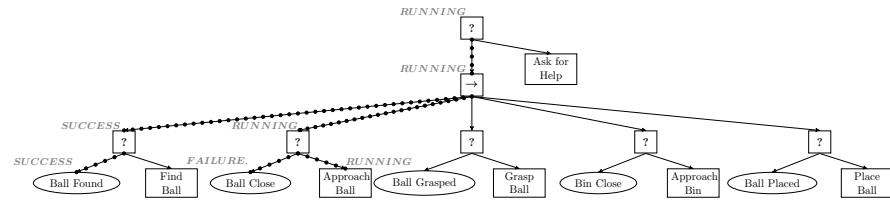
Table 1.1: The node types of a BT.

1.3.1 Execution Example of a BT

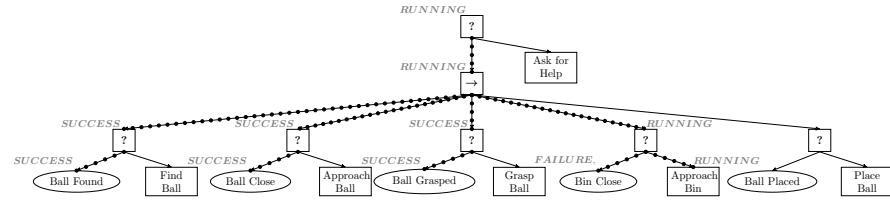
Consider the BT in Figure 1.6 designed to make an agent look for a ball, approach it, grasp it, proceed to a bin, and place the ball in the bin. This example will illustrate the execution of the BT, including the reactivity when another (external) agent takes the ball from the first agent, making it 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 1.7a). 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 1.7b). Then the agent eventually reaches the ball, picks it up and goes towards the bin (see Figure 1.7c). 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 1.7d).

**Fig. 1.6:** BT encoding the behavior of Example 2.1.

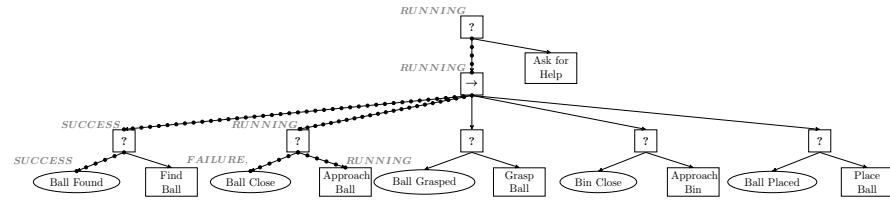
(a) Ticks' traversal when the robot is searching the ball.



(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).

Fig. 1.7: Visualization of the ticks' traversal in the 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.

Nodes with memory [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.

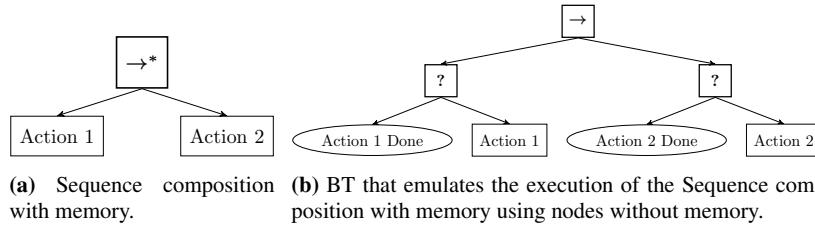


Fig. 1.8: Relation between memory and memory-less BT nodes.

Remark 1.1. 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.³ We use a clone of the Namco's Pac-Man computer game depicted in Figure 1.9⁴.

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.

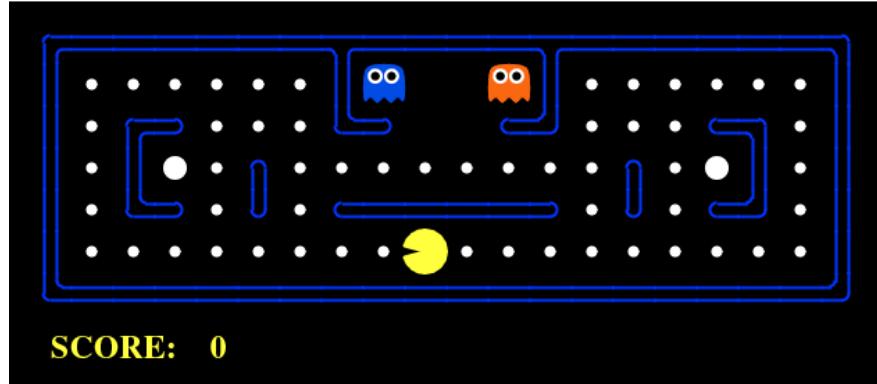


Fig. 1.9: The game Pac-Man for which we will design a BT. There exists maps of different complexity.

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.



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

³ <https://btirai.github.io/>

⁴ The software was developed at UC Berkeley for educational purposes. More information available at: http://ai.berkeley.edu/project_overview.html

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

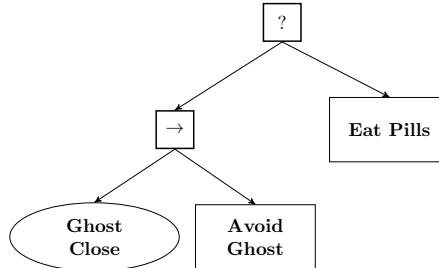


Fig. 1.11: If a Ghost is Close, the BT will execute the Action Avoid Ghost, else it will run Eat Pills.

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 Fig. 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.

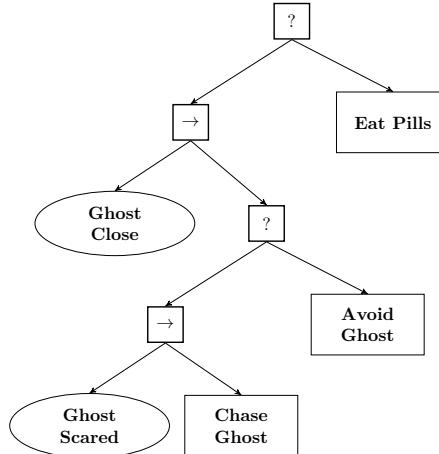


Fig. 1.12: BT for the Combative Behavior

With this incremental design, we have created a basic 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 to case basis, as discussed in Section 3.7.

1.5 Creating a BT for a Mobile Manipulator Robot



Fig. 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.⁵ 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.

⁵ <https://btirai.github.io/>

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.

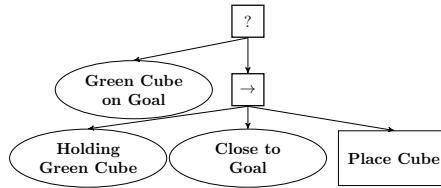


Fig. 1.14: BT for the simple Scenario.

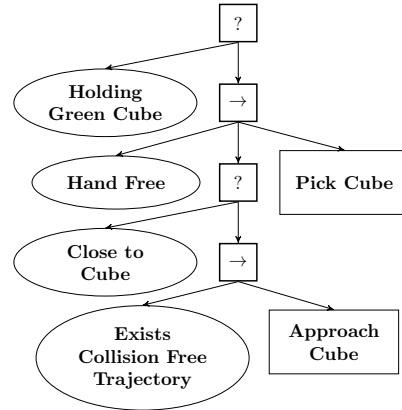
Now, thanks to the modularity of BTs, we can separately design the BTs needed to satisfy the two lower conditions in Fig. 1.14, i.e., the BT needed to grasp the green cube and the BT needed to reach the goal area. To grasp the green cube, the robot needs to have the *hand free* and be *close to the cube*.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

Now we can extend the simple BT in Fig. 1.14 above by replacing the two lower conditions in Fig. 1.14 with the two BTs in Fig. 1.15. The result can be seen in Fig. 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.

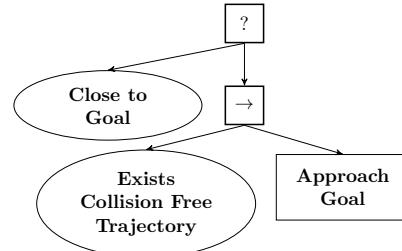
We can continue to incrementally build the BT in this way to handle more situations, for instance removing obstructing objects to ensure that a *collision free trajectory exists*, and dropping things in the hand to be able to pick the green cube up.

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.



(a) A BT that picks the green cube.



(b) A BT that reaches the goal region.

Fig. 1.15: Illustrations of a BT carrying out the subtasks of picking the green cube and reaching the goal area

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 HFSM[45] decomposing the mission task in multiple subtasks 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.

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

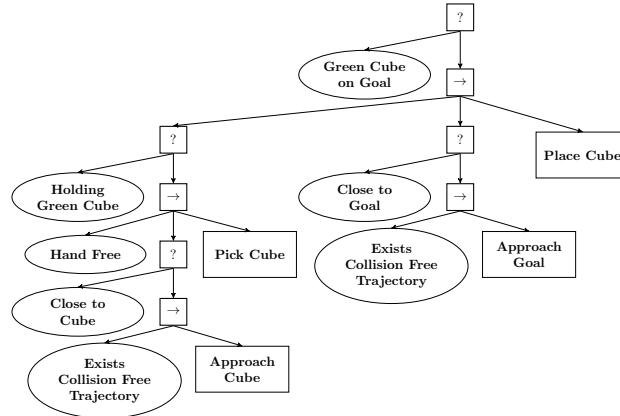


Fig. 1.16: Final BT resulting from the aggregation of the BTs in Figs. 1.14-1.15



Fig. 1.17: Trucks running the Scania iQmatic's software.⁶

development of early prototypes; and their maintainability, making the editing task easier. Figure 1.17 shows two trucks used in the iQmatic project.

⁶ Picture courtesy of Scania.com

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.



Fig. 1.18: Experimental platform of the CoSTAR project.⁷

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 subtasks. The order in which the subtasks are executed is independent from the subtask 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.

SARAFun⁸ 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 advantages of state of the art techniques in sensory and cognitive abilities, robot control, and planning.

⁷ Picture courtesy of <http://cpaxton.github.io/>

⁸ <http://h2020sarafun.eu>



Fig. 1.19: Experimental platform of the SARAFun project.⁹

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 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.

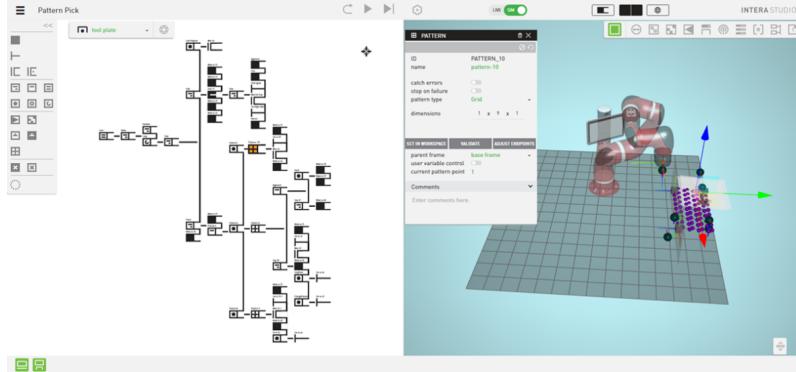


Fig. 1.20: Intera's BT (left) and simulation environment (right).¹⁰

⁹ Setup located at CERTH, Thessaloniki, Greece. Picture courtesy of Angeliki Topalidou-Kyniazopoulou.

¹⁰ Picture courtesy of <http://www.rethinkrobotics.com/intera/>

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 automation with unparalleled ease of deployment.”¹¹ 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 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.¹²

1.6.3 BTs in the Amazon Picking Challenge



Fig. 1.21: The KTH entry in the Amazon Picking Challenge at ICRA 2015.

¹¹

<http://www.rethinkrobotics.com/news-item/rethink-robotics-releases-intera-5-new-approach-automation/>

¹² <http://twimage.net/rodney-brooks-743452002>

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 the APC's preliminary phases. The best performing teams earned the right to compete at the finals and the source codes of the finalists were made publicly available.¹³

The KTH entry in the final challenge used BTs in both 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 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 2017.¹⁴ BTs are a fundamental part of the software architecture of JIBO¹⁵, including an open SDK inviting external contributors to develop new skills for the robot.

¹³ <https://github.com/amazon-picking-challenge>

¹⁴ <http://time.com/5023212/best-inventions-of-2017/>

¹⁵ <https://developers.jibo.com/docs/behavior-trees.html>



Fig. 1.22: The JIBO social robot has an SDK based on BTs.

Chapter 2

How Behavior Trees Generalize and Relate to Earlier Ideas

In this chapter, we describe how BTs relate to, and often generalize, a number of well known control architectures including FSMs (Section 2.1), the Subsumption Architecture (Section 2.3), the Teleo-Reactive Approach (Section 2.4) and Decision Trees (Section 2.5). We also present advantages and disadvantages of each approach. Finally, we list a set of advantages and disadvantages of BTs (2.6). Some of the results of this chapter were previously published in the journal paper [13].

2.1 Finite State Machines

A FSM is one of the most basic mathematical models of computation. The FSM consists of a set of states, transitions and events, as illustrated in Fig. 2.1 showing an example of a FSM designed to carry out a grab-and-throw task. Note that the discussion here is valid for all control architectures based on FSMs, including Mealy [46] and Moore [41] machines.

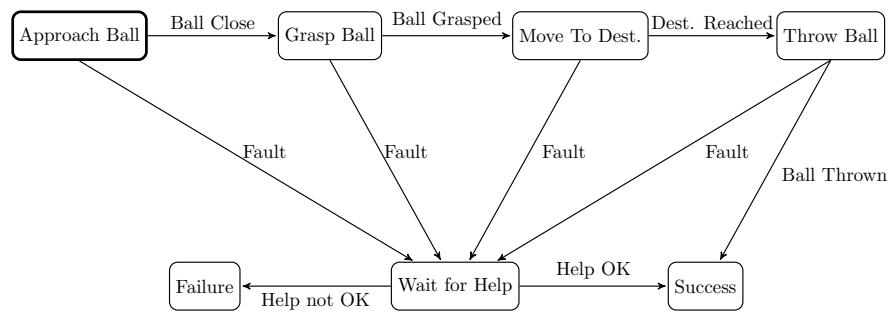


Fig. 2.1: Graphical representation of a FSM designed to carry out a simple grab-and-throw task. The initial state has a thicker border, and events names are given next to the corresponding transition arrows.

2.1.1 Advantages and disadvantages

FSMs are widely used due to their three main advantages:

- Very common structure, used in many different parts of computer science.
- Intuitive and easy to understand.
- Easy to implement.

However, the drawbacks of FSMs give rise to problems when the system modelled grows in complexity and number of states, as described briefly in Section 1.2. In particular we have the following drawbacks

- Reactivity/Modularity tradeoff. A reactive system needs many transitions, and every transition corresponds to a Goto statement, see Section 1.2. In particular, the transitions give rise to the problems below:
 - Maintainability: Adding or removing states requires the re-evaluation of potentially large number of transitions and internal states of the FSM. This makes FSMs highly susceptible to human design errors and impractical from an automated design perspective.
 - 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, making it unpractical to reuse the same sub-FSM in multiple projects.

2.2 Hierarchical Finite State Machines

Hierarchical FSMs (HFSMs), also known as State Charts [29], were developed to alleviate some of the disadvantages of FSMs. In a HFSM, a state can in turn contain one or more substates. A state containing two or more states is called a *superstate*. In a HFSM, a *generalized transition* is a transition between superstates. Generalized transitions can reduce the number of transitions by connecting two superstates rather than connecting a larger number of substates individually. Each superstate has one substate identified as the starting state, executing whenever a transition to the superstate occurs. Figure 2.2 shows an example of a HFSM for a computer game character.

2.2.1 Advantages and disadvantages

The main advantages of HFSMs are:

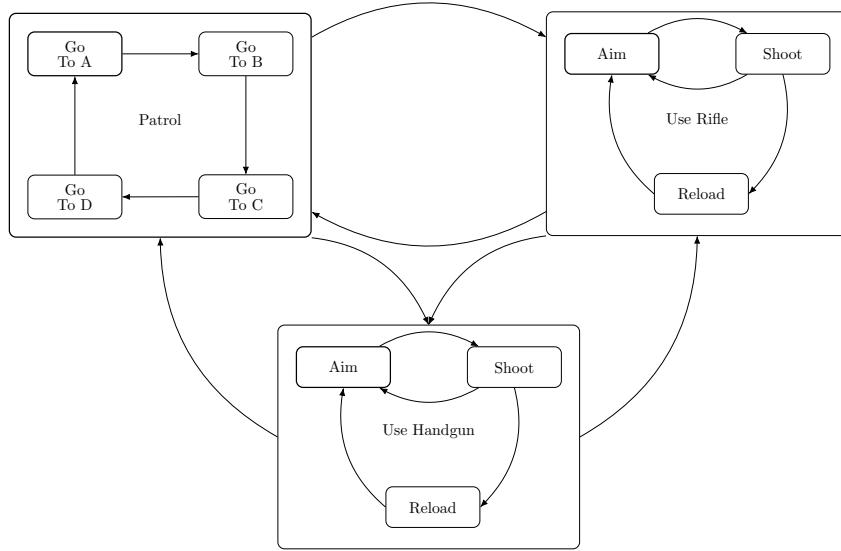


Fig. 2.2: Example of a HFSM controlling a NPC of a combat game. *Patrol*, *Use Rifle*, and *Use Handgun* are superstates.

- Increased Modularity: it is possible to separate the tasks in subtasks. However these subtasks often still depend on each other through state-dependent transitions.
- Behavior inheritance: The state nesting in HFSMs allows so-called *behavior inheritance*. Behavior inheritance allows substates to inherit behaviors from the superstate; for example, in the HFSM depicted in Figure 2.2, while in the substates inside *Use Handgun*, the character holds the weapon using one hand whereas while in the substates inside *Use Rifle*, the character holds the weapon using two hands. Thus, there is no need for the sub states to specify this property, instead, it 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 subgraph.
- Manually created hierarchy: Although HFSMs were conceived as a hierarchical version of FSMs, the hierarchy has to be user defined and editing such a hierarchy can be difficult. The hierarchy resolves some problems, but a reactive HFSM still results in some sub graphs being fully connected with many possible transitions, see Fig. 2.3.

From a theoretical standpoint, every execution described by a BT can be described by a FSM and vice-versa [55, 38]. However, due to the number of transitions, using a FSM as a control architecture is unpractical for some applications as shown in Chapter 1. Moreover, a potential problem is that a FSM does not assume that the conditions triggering the outgoing transitions from the same state are mutually exclusive. When implemented, the conditions are checked regularly in discrete time, hence there exists a non-zero probability that two or more conditions hold simultaneously after one cycle. To solve this problem we need to redefine some transitions, as done in the FSM in Figure 2.22, making the propositions of the outgoing transitions mutually exclusive. A FSM of this format is impractical to design for both humans and computers. Manually adding and removing behaviors 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 make any automated process to analyze or synthesize FSMs computationally expensive.

HFSMs is the most similar control architecture to BTs in terms of purpose and use. To compare BTs with HFSMs we use the following complex example. Consider the HFSM shown in Figure 2.3 describing the behavior of a humanoid robot. We can describe the same functionality using the BT shown in Figure 2.4. Note that we have used the standard notation [29] 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 subtree can be seen as a module of its own, with the same interface as an atomic action.

In the HFSM shown in Figure 2.3, a proposition needs to be given for each transition, and to improve readability we have numbered these propositions from $C1$ to $C10$. 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).

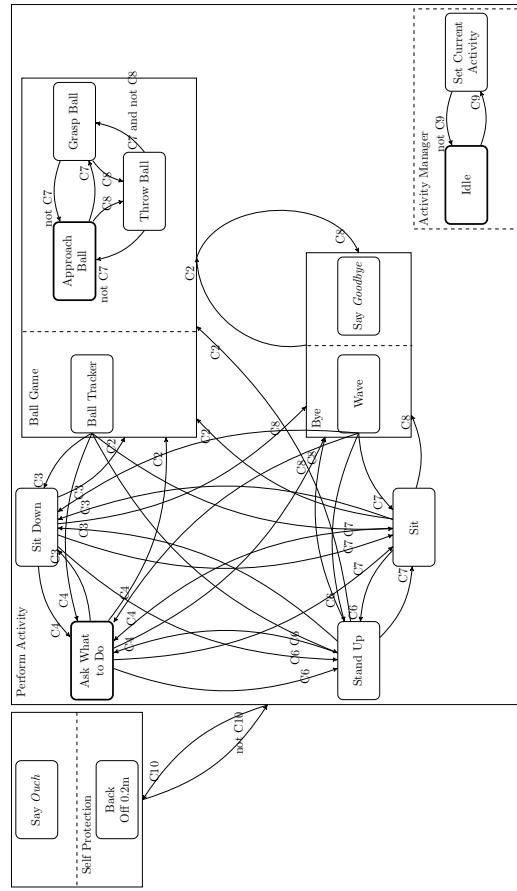


Fig. 2.3: A HFSM description of the BT in Figure 2.4. The transition conditions are shown at the end of each arrow to indicate the direction of the transition. Note how the complexity of the transitions *within* each layer of the HFSM grows with the number of nodes. The conditions labels are: C1 = Activity Sit, C2 = Not Know What to Do, C3 = Activity Sleep, C4 = Activity Stand Up, C5 = Activity Ball Game, C6 = Ball Close, C7 = Ball Grasped, C8 = New User Suggestion, C9 = Activity Sit, C10 = Bumper Pressed.

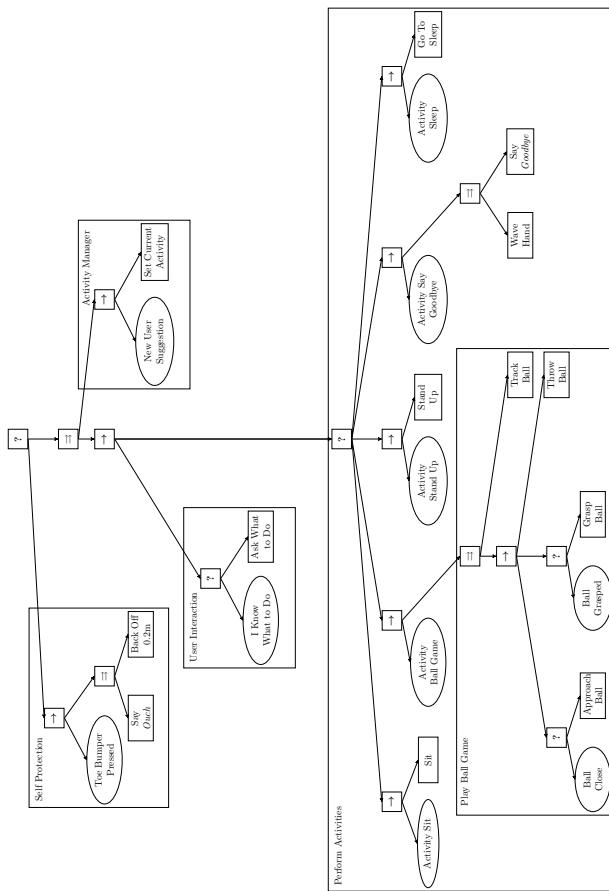


Fig. 2.4: A BT that combines some capabilities of a humanoid robot in an interactive and modular way. Note how atomic actions can easily be replaced by more complex sub-BTs.

2.2.2 Creating a FSM that works like a BTs

As described in Chapter 1, each BT returns *Success*, *Running* or *Failure*. Imagine we have a state in a FSM that has 3 transitions, corresponding to these 3 return statements. Adding a Tick source that collect the return transitions and transfer the execution back into the state, as depicted in Figure 2.5, we have a structure that resembles a BT.

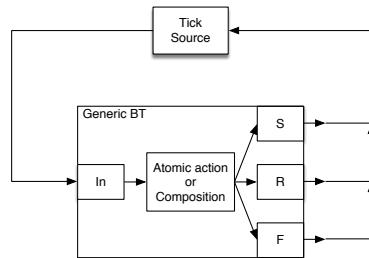


Fig. 2.5: An FSM behaving like a BT, made up of a single normal state, three out transitions Success (S), Running (R) and Failure (F), and a Tick source.

We can now compose such FSM states using both Fallback and Sequence constructs. The FSM corresponding to the Fallback example in Figure 2.6 would then look like the one shown in Figure 2.7.

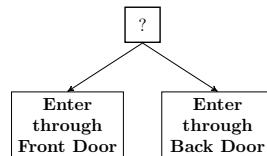


Fig. 2.6: A Fallback is used to create an *Enter Building* BT. The back door option is only tried if the front door option fails.

Similarly, the FSM corresponding to the sequence example in Figure 2.8 would then look like the one shown in Figure 2.9, and a two level BT, such as the one in Figure 2.10 would look like Figure 2.11.

A few observations can be made from the above examples. First, it is perfectly possible to design FSMs with a structure taken from BTs. Second, considering that a BT with 2 levels corresponds to the FSM in Figure 2.11, a BT with 5 levels, such as the one in Figure 2.12 would correspond to a somewhat complex FSM.

Third, and more importantly, the *modularity* of the BT construct is illustrated in Figures 2.5-2.11. Figure 2.11 might be complex, but that complexity is encapsu-

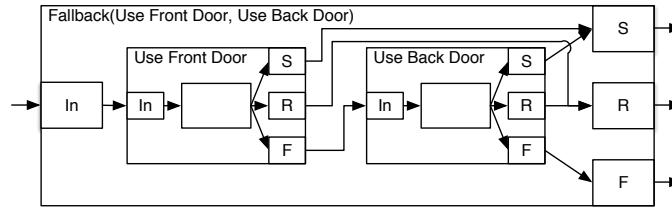


Fig. 2.7: A FSM corresponding to the Fallback BT in Figure 2.6. Note how the second state is only executed if the first fails.

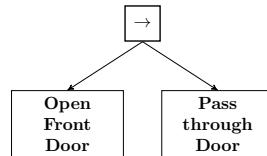


Fig. 2.8: A Sequence is used to create an *Enter Through Front Door* BT. Passing the door is only tried if the opening action succeeds.

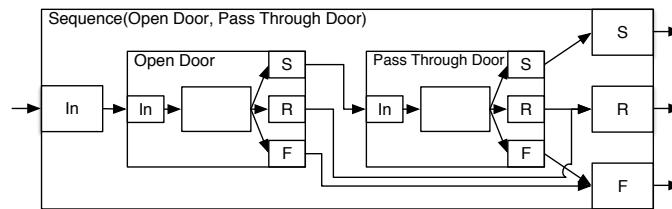


Fig. 2.9: An FSM corresponding to the Sequence BT in Figure 2.8. Note how the second state is only executed if the first succeeds.

lated in a box with a single in-transition and three out-transitions, just as the box in Figure 2.5.

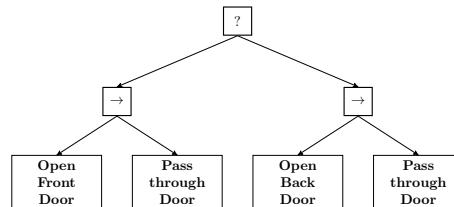


Fig. 2.10: The two BTs in Figures 2.6 and 2.8 are combined to larger BT. If e.g. the robot opens the front door, but does not manage to pass through it, it will try the back door.

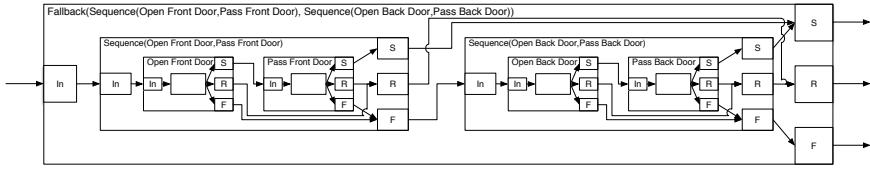


Fig. 2.11: An FSM corresponding to the BT in Figure 2.10.

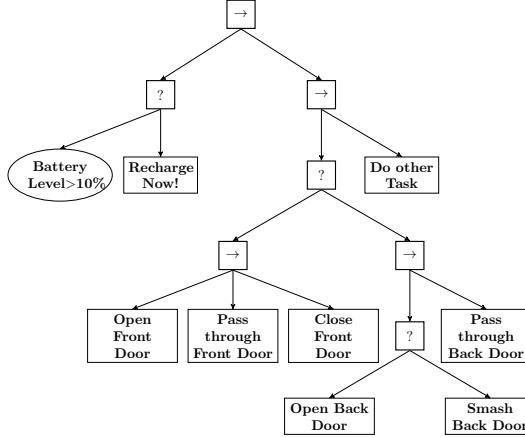


Fig. 2.12: Combining the BTs above and some additional Actions, we get a flexible BT for entering a building and performing some task.

Fourth, as was mentioned in Section 1.2, the decision of what to do after a given sub-BT returns is always decided on the parent level of that BT. The sub-BT is ticked, and returns *Success*, *Running* or *Failure* and the parent level decides whether to tick the next child, or return something to its own parent. Thus, the BT ticking and returning of a sub-BT is similar to a *function call* in a piece of source code, just as described in Section 1.2. A function call in Java, C++, or Python moves execution to another piece of the source code, but then returns the execution to the line right below the function call. What to do next is decided by the piece of code that made the function call, not the function itself. As discussed, this is quite different from standard FSMs where the decision of what to do next is decided by the state being transitioned to, in a way that resembles the Goto statement.

2.2.3 Creating a BT that works like a FSM

If you have a FSM design and want to convert it to a BT, the most straight forward way is to create a *State Variable* available to all parts of the BT and then list all the states of the FSM and their corresponding transitions and actions as shown in Figure 2.13.

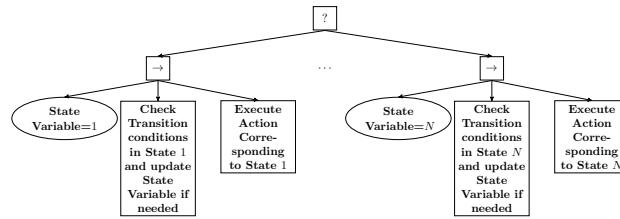


Fig. 2.13: Example of a straightforward translation of a FSM to a BT using a global *State Variable*.

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 has found a number of successful applications [7]. The basic idea of the Subsumption 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, is given access to the actuators. Thus, a controller with a higher priority is able to subsume a lower level one. Figure 2.14 shows an example of a Subsumption Architecture.

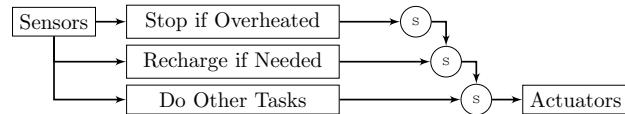


Fig. 2.14: Example of Subsumption Architecture composed by three controllers. The controller *Stop if Overheated* subsumes the controller *Recharge if Needed*, which subsumes the controller *Do Other Tasks*.

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 controllers can be hard to estimate.

2.3.2 How BTs Generalize the Subsumption Architecture

There is a straightforward mapping from a Subsumption Architecture design to a BT using a Fallback node. If each controller in the Subsumption Architecture is turned into a BT Action, returning running if the binary output indicates that it wants to run and Failure the rest of the time, a standard Fallback composition will create an equivalent BT. As an example we see that the structure in Fig. 2.14 is represented by the BT in Fig. 2.15. A more formal argument using a state space representation of BTs will be given in Section 6.2.

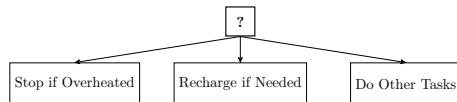


Fig. 2.15: A BT version of the subsumption example in Figure 2.14.

2.4 Teleo-Reactive programs

Teleo-Reactive (TR) programs were introduced by Nils Nilsson [51] at Stanford University in 1994 to allow engineers to define the behavior of a robotics system

that had 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 described by a list of condition-action rules as the following:

$$\begin{aligned} c_1 &\rightarrow a_1 \\ c_2 &\rightarrow a_2 \\ &\dots \\ c_m &\rightarrow a_m \end{aligned}$$

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 *move forwards* is a durative action, whereas the action *take one step* is discrete. In a TR program, a durative action is executed as long as its corresponding condition remains the one 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 2.16 shows an example of a TR program for navigating in a obstacle free environment.

$$\begin{aligned} \text{Equal(pos,goal)} &\rightarrow \text{Idle} \\ \text{Heading Towards (goal)} &\rightarrow \text{Go Forwards} \\ (\text{else}) &\rightarrow \text{Rotate} \end{aligned}$$

Fig. 2.16: Example of teleoreactive program carrying out a navigation task. If the robot is in the goal position, the action performed is *Idle* (no actions executed). Otherwise if it is heading towards the goal, the action performed is *Go Forwards*. Otherwise, the robot performs the action *Rotate*.

TR programs have been extended in several directions, including integrating TR programs with automatic planning and machine learning [4, 73], removing redundant parts of a TR program [47], and using TR programs to play robot soccer [26].

2.4.1 Advantages and disadvantages

The main advantages of a TR program are:

- Reactive execution: TR programs enable reactive executions 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 cause 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 enable failure handling, a TR program needs to have a condition that checks if an action fails.

2.4.2 How BTs Generalize Teleo-Reactive Programs

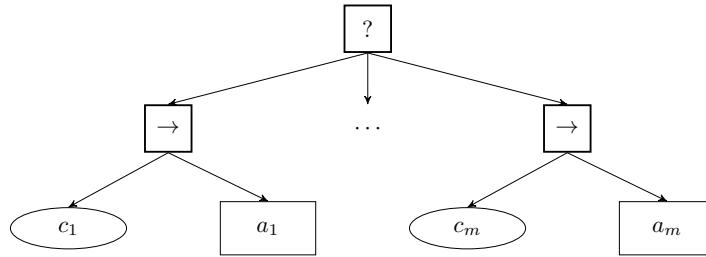


Fig. 2.17: The BT that is analogous to a given TR.

The core idea of continuously checking conditions and applying the corresponding rules can be captured using a Fallback node and pairs of conditions and actions. Thus, a general TR program can be represented in the BT of Fig. 2.17. A more formal argument using a state space representation of BTs will be given in Section 6.4.

2.5 Decision Trees

A Decision Tree is a directed tree that represents a list of nested if-then clauses used to derive decisions [65]. Leaf nodes describe decisions, conclusions, or actions to be carried out, whereas non-leaf nodes describe predicates to be evaluated. Figure 2.18 shows a Decision Tree where according to some conditions, a robot will decide what to do.

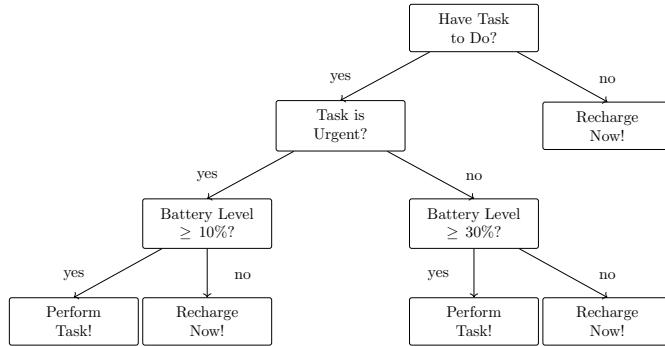


Fig. 2.18: Example of a Decision Tree executing a generic robotic task. The predicates are evaluated traversing the tree in a top-down fashion.

2.5.1 Advantages and disadvantages

The main advantages of a Decision Tree are:

- **Modularity:** The Decision Tree structure is modular, in the sense that a subtree can be developed independently from the rest of the Decision Tree, and added where suitable.
- **Hierarchy:** Decision Tree's structure is hierarchical, in the sense that predicates are evaluated in a top-down fashion.
- **Intuitive structure:** It is straightforward to design and understand Decision Trees.

The main disadvantages of a Decision Tree are:

- No information flow out from the nodes, making failure handling very difficult

2.5.2 How BTs Generalize Decision Trees

A general Decision Tree can be converted into a BT using the mapping shown in Fig. 2.19. By converting the predicate to a condition, letting the leaves be Action nodes always returning Running, we can map each decision node of the Decision Tree to a small BT. Applying the mapping to the Decision Tree of Fig. 2.18 we get the BT of Fig. 2.20. A more formal argument using a state space representation of BTs will be given in Section 6.1. Note that this structure requires actions always returning Running, reflecting the drawback of Decision Trees that no information flows out of the actions.

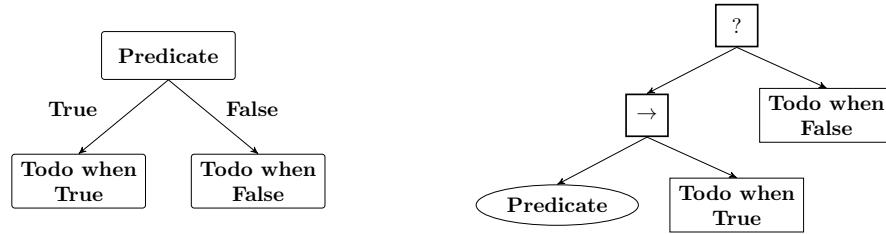


Fig. 2.19: The basic building blocks of Decision Trees are ‘If ... then ... else ...’ statements (left), and those can be created in BTs as illustrated above (right).

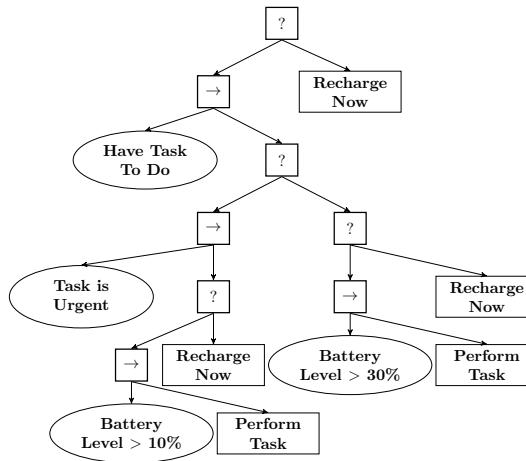


Fig. 2.20: A BT that is equivalent to the Decision Tree in Figure 2.18.

2.6 Advantages and Disadvantages of Behavior Trees

Having looked at how BTs relate to a set of existing control architectures we will now take a step back and list a number of advantages and disadvantages of BTs.

2.6.1 Advantages

As described in Section 1.2 many advantages stem from BTs being both modular and reactive. Below we list a set of advantages of BTs.

Modular: By modular, we mean the degree to which a system’s components may be separated into building blocks, and recombined [23]. A modular system can

be designed, implemented, tested and reused one module at a time. The benefits of modularity thus increases, the more complex a system is, by enabling a divide and conquer approach when designing, implementing and testing.

BTs are modular, since each subtree of a BT can be seen as a module in the above sense, with a standard interface given by the return statuses. Thus, BTs are modular on all scales ranging from the topmost subtrees to all the leaves of the tree.

Hierarchical organization: If a control architecture contains several levels of decision making it is hierarchical. The possibility of designing and analyzing structures on different hierarchical levels is important for both humans and computers, as it enables e.g., iterative refinement and extensions of a plan, see Section 3.5. BTs are hierarchical, since each level of a BT automatically defines a level in the hierarchy.

Reusable code: Having reusable code is very important in any large, complex, long-term project. The ability to reuse designs relies 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. To enable reuse of code, each module must interface the control architecture in a clear and well-defined fashion.

BTs enable reusable code, since given the proper implementation, any subtree can be reused in multiple places of a BT. Furthermore, 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 viceversa.

Reactivity: By reactive we mean the ability to quickly and efficiently react to changes. For unstructured environments, where outcomes of actions are not certain and the state of the world is constantly changed by external actors, plans that were created offline and then executed in an open loop fashion are often likely to fail.

BTs are reactive, since 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 return *Failure/Success* if the action failed/succeeded). Thus, BTs are highly responsive to changes in the environment.

Human readable: A readable structure is desirable for reducing the cost of development 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.

BTs are human readable due to their tree structure and modularity.

Expressive: A control architecture must be sufficiently expressive to encode a large variety of behaviors.

BTs are at least as expressive as FSMs, see Section 2.1, the Subsumption Architecture, see Section 2.3, Teleo-Reactive programs, see Section 2.4, and Decision Trees, see Section 2.5.

Suitable for analysis: Safety critical robot applications often require an analysis of qualitative and quantitative system 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 of analyzing whether properties are preserved over compositions of subtasks.

BTs have tools available to evaluate such system properties, see Chapters 5 and 9.

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 control architecture 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).

BTs are suitable for automatic synthesis in terms of both planning, see Section 3.5 and in more detail Chapter 7 and learning, see Chapter 8.

To illustrate the advantages listed above, we consider the following simple example.

Example 2.1. A robot is tasked 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 2.21a), the robot moves towards the bin (Figure 2.21b). While moving towards the bin, an external entity takes the ball from the robot's gripper (Figure 2.21c) and immediately throws it in front of the robot, where it can be seen (Figure 2.21d). The robot aborts the execution of moving and it starts to approach the ball again.

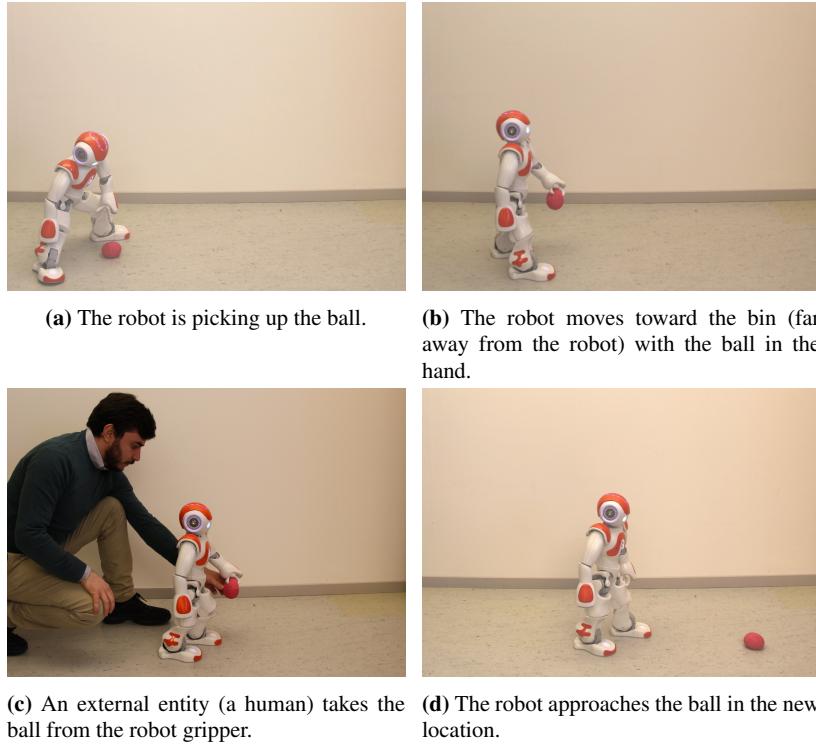


Fig. 2.21: Execution stages of Example 2.1.



Fig. 2.22: FSM modeling the robot's behavior in Example 2.1. The initial state has a thicker border.

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 task requires a control architecture suitable for extensions. These extensions might be human made (e.g. the robot asks the operator to update the current action policy) requiring an architecture to be *human readable*, or automated (e.g. using model-based reasoning) requiring an architecture 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 architecture to be *suitable for analysis*. Finally, once the policy is designed, it is desirable that it can be *reused* in other contexts.

Most control architectures lack one or more of the properties described above. Take as an example a FSM modeling the behavior of the robot in Example 2.1, depicted in Figure 2.22. As can be seen, even for this simple example the FSM gets fairly complex with many transitions.

2.6.2 Disadvantages

In this section we describe some disadvantages of BTs.

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 BT engine only needs to be implemented once, it can be reused, and several BT engines are available as off the shelf software libraries.¹

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 even infeasible. In those cases a closed-loop execution (using any architecture) presents more costs than advantages. However, it is still possible to design an open-loop task execution using BTs with memory nodes, see Section 1.3.2.

¹ C++ library: <https://github.com/miccol/Behavior-Tree>
ROS library: http://wiki.ros.org/behavior_tree
python library: <https://github.com/futureneer/beetree>

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

BTs are different from FSMs. BTs, despite being easy to understand, require a new mindset when designing a solution. The execution of BTs is not focused on states but on conditions and the switching is not event driven but tick driven. The ideas presented in this book, and in particular the design principles of Chapter 3, are intended to support the design of efficient BTs.

BT tools are less mature. Although there is software for developing BTs, it is still far behind the amount and maturity of the software available for e.g. FSMs.

Chapter 3

Design principles

BTs are fairly easy to understand and use, but to make full use of their potential it can be good to be aware of a set of design principles that can be used in different situations. In this chapter, we will describe these principles using a number of examples. First, in Section 3.1, we will describe the benefit of using explicit success conditions in sequences, then, in Section 3.2, we describe how the reactivity of a BT can be increased by creating implicit sequences, using Fallback nodes. In Section 3.3, we show how BTs can be designed in a way that is similar to Decision Trees. Then, in Section 3.4, we show how safety can be improved using sequences. Backchaining is an idea used in automated planning, and in Section 3.5 we show how it can be used to create deliberative, goal directed, BTs. Memory nodes and granularity of BTs is discussed in Sections 3.6 and 3.7. Finally, we show how easily all these principles can be combined at different levels of a BT in Section 3.8.

3.1 Improving Readability using Explicit Success Conditions

One advantage of BTs is that the switching structure is clearly shown in the graphical representation of the tree. However, one thing that is not shown is the details regarding when the individual actions return Success and Failure.

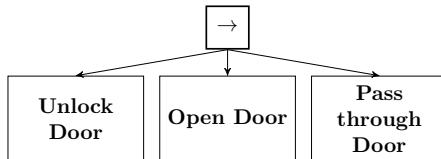


Fig. 3.1: Simple Sequence

Consider the sequence in Figure 3.1. One can assume that Unlock Door returns Success when it has unlocked the door, but what if it is called when the door is already unlocked? Depending on the implementation it might either return Success immediately, or actually try to unlock the door again, with the possibility of returning Failure if the key cannot be turned further. A similar uncertainty holds regarding the implementation of Open Door (what if the door is already open?) and Pass through Door. To address this problem, and remove uncertainties regarding the implementation, explicit Success conditions can be included in the BT.

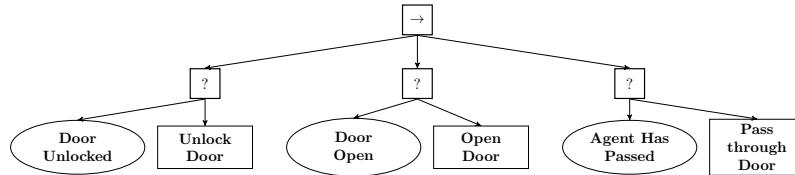


Fig. 3.2: Sequence with explicit success conditions. Note how each action is paired with a condition through a Fallback node, making the success condition of the pair explicit.

In Figure 3.2, the BT from Figure 3.1 has been extended to include explicit success conditions. These conditions are added in a pair with the corresponding action using a Fallback node. Now, if the door is already unlocked and open, the two first conditions of Figure 3.2 will return Success, the third will return Failure, and the agent will proceed to execute the action Pass through Door.

3.2 Improving Reactivity using Implicit Sequences

It turns out that we can improve the reactivity of the BT in Figure 3.2 even further, using the fact that BTs generalize the Teleo-Reactive approach, see Section 2.4.2. Consider the case when the agent has already passed the door, but the door is closed behind it. The BT in Figure 3.2 would then proceed to unlock the door, open it, and then notice that it had already passed it and return Success.

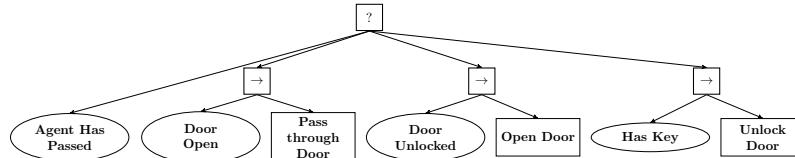


Fig. 3.3: An Implicit Sequence is constructed using a Fallback node, reversing the order of the actions and pairing them with appropriate preconditions.

The key observation needed to improve reactivity is to realize that the goal is to get through the door, and that the other actions are just means to get to that goal. In the BT in Figure 3.3 we have reversed the order of the actions in order to check the goal state first. We then changed fallbacks to sequences and vice versa, and finally changed the conditions. Now, instead of checking outcomes, or success conditions as we did in Figure 3.2, we check preconditions, conditions needed to execute the corresponding actions, in Figure 3.3. First the BT checks if the agent has passed the door, if so it returns Success. If not, it proceeds to check if the door is open, and if so passes through it. If neither of the previous conditions are satisfied, it checks if the door is unlocked, and if so starts to open it. As a final check, if nothing else returns Success, it checks if it has the key to the door. If it does, it tries to open it, if not it returns Failure.

The use of implicit sequences is particularly important in cases where the agent needs to undo some of its own actions, such as closing a door after passing it. A systematic way of creating implicit sequences is to use back chaining, as described in Section 3.5.

3.3 Handling Different Cases using a Decision Tree Structure

Sometimes, a reactive switching policy can be easily described in terms of a set of cases, much like a Decision Tree. Then, the fact that BTs generalize Decision Trees can be exploited, see Section 2.5.2.

A simple Pac-Man example can be found in Figure 3.4. The cases are separated by the two conditions *Ghost Close* and *Ghost Scared*. If no ghost is close, Pac-Man continues to eat pills. If a ghost is close, the BT checks the second condition, *Ghost Scared*, which turns true if Pac-Man eats a Power Pill. If the ghost is scared, Pac-Man chases it, if not, Pac-Man avoids the Ghost.

3.4 Improving Safety using Sequences

In some agents, in particular robots capable of performing irreversible actions such as falling down stairs or damaging equipment, it is very important to be able to guarantee that some situations will never occur. These unwanted situations might be as simple as failing to reach the recharging station before running out of battery, or as serious as falling down a staircase and hurting someone.

A Sequence node can be used to guarantee safety, as shown in Figure 3.5. Looking closer at the BT in Figure 3.5 we see that it will probably lead to an unwanted chattering behavior. It will recharge until it reaches just over 20% and then start doing Main Task, but the stop as soon as the battery is back at 20%, and possibly end

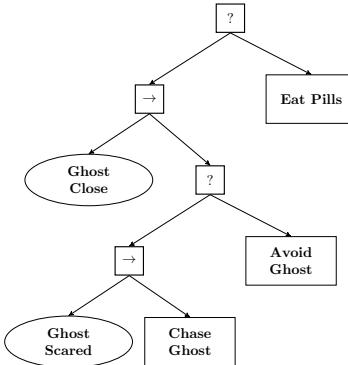


Fig. 3.4: Simple Pac-Man example using a Decision Tree structure.

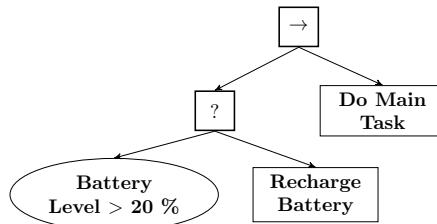


Fig. 3.5: A BT that is guaranteed not to run out of batteries, as long as Main Task keeps the robot close enough to the recharging station so that 20% of battery will be enough to travel back.

up chattering i.e. quickly switching between the two tasks. The solution is to make sure that once recharging, the robot waits until the battery is back at 100%. This can be achieved by the BT in Fig 3.6.

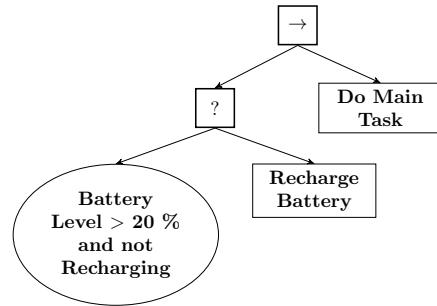


Fig. 3.6: By changing the condition in Fig. 3.5 the robot now keeps recharging until the Battery level reaches 100%.

3.5 Creating Deliberative BTs using Backchaining

BTs can also be used to create deliberative agents, where the actions are carried out in order to reach a specific goal. We will use an example to see how this is done. Imagine we want the agent to end up inside a house. To make that goal explicit, we create the trivial BT in Figure 3.7, with just a single condition checking if the goal is achieved or not.

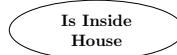


Fig. 3.7: A BT composed of a single condition checking if the goal is achieved.

Now imagine we have a set of small BTs such as the ones in Figures 3.8 and 3.9, each on the format of the general Postcondition-Precondition-Action (PPA) BT in Figure 3.11.

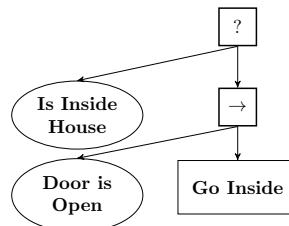


Fig. 3.8: PPA for achieving the postcondition *Is Inside House*. If the postcondition is not satisfied already, the BT checks the precondition *Door is Open*, if so it executes the action *Go Inside*.

If we have such a set, we can work our way backwards from the goal (backchaining) by replacing preconditions with PPAs having the corresponding postcondition. Thus replacing the single condition in Figure 3.7 with the PPA of Figure 3.8 we get Figure 3.8 again, since we started with a single condition. More interestingly, if we replace the precondition *Door is Open* in Figure 3.8 with the PPA of Figure 3.9 we get the BT of Figure 3.10

Thus we can iteratively build a deliberative BT by applying Algorithm 4. Looking at the BT in Figure 3.10 we note that it first checks if the agent *Is Inside House*, if so it returns Success. If not it checks if *Door is Open*, and if it is, it proceeds to *Go Inside*. If not it checks if *Door is Unlocked* and correspondingly executes *Open Door*. Else it checks if *Door is Weak*, and it *Has Crowbar* and proceeds to *Break Door Open* if that is the case. Else it returns Failure. If an action is executed it might either succeed, which will result in a new condition being satisfied and another action being executed until the task is finished, or it might fail. If *Go Inside* fails, the

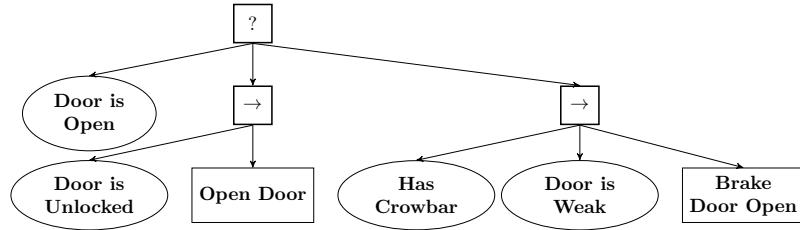


Fig. 3.9: PPA for achieving the postcondition *Door is Open*. If the postcondition is not satisfied, the BT checks the first precondition *Door is Unlocked*, if so it executes the action *Open Door*, if not it checks the second set of preconditions, starting with *Has Crowbar*, if so it checks *Door is Weak*, if both are satisfied it executes *Brake Door Open*.

Algorithm 4: Pseudocode of Backchaining Algorithm

Data: Set of Goal Conditions C_i , and a set of PPAs

Result: A reactive BT working to achieve the C_i s

- 1 Replace all C_i with PPAs having C_i as postcondition;
 - 2 **while the BT returns Failure when ticked do**
 - 3 **replace one of the preconditions returning Failure (inside a PPA) with another complete PPA having the corresponding condition as postcondition, and therefore including at least one action to achieve the failing condition ;**
-

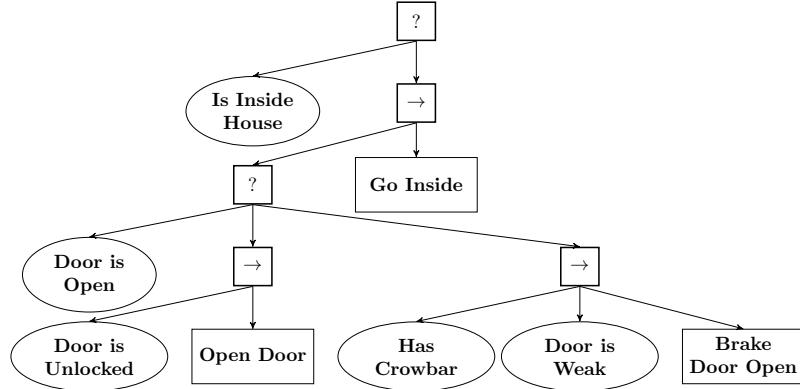


Fig. 10: The result of replacing *Door is Open* in Figure 3.8 with the PPA of Figure 3.9.

whole BT returns Failure, but if *Open Door* fails, the conditions *Door is Weak* and *Has Crowbar* are checked.

In general, we let the PPA have the form of Figure 3.11, with one postcondition C that can be achieved by either one of a set of actions A_i , each of these actions are combined in a sequence with its corresponding list of preconditions C_{ij} , and these action precondition sequences are fallbacks for achieving the same objective. We see that from an efficiency point of view it makes sense to put actions that are most

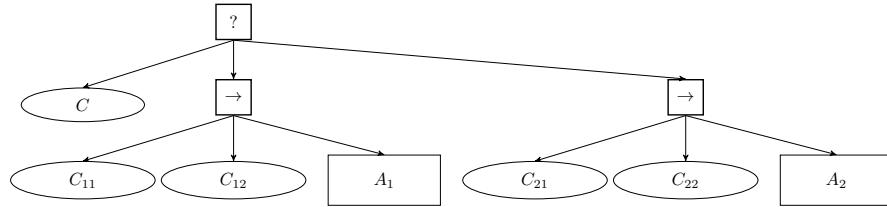


Fig. 3.11: General format of a PPA BT. The Postcondition C can be achieved by either one of actions A_1 or A_2 , which have Preconditions C_{1i} and C_{2i} respectively.

likely to succeed first (to avoid unnecessary failures) and check preconditions that are most likely to fail first (to quickly move on to the next fallback option).

3.6 Creating Un-Reactive BTs using Memory Nodes

As mentioned in Section 1.3.2, sometimes a child, once executed, does not need to be re-executed for the whole execution of a task. Control flow nodes with memory are used to simplify the design of a BT avoiding the unwanted re-execution of some nodes. The use of nodes with memory is advised exclusively for those cases where there is no unexpected event that will undo the execution of the subtree in a composition with memory, as in the example below.

Consider the behavior of an industrial manipulator in a production line that has to *pick*, *move*, and *place* objects. The robot's actions are carried out 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. In this example the robot operates in a structured environment that is fully predictable in space and time. In this case we can disregard any unexpected change enabling us to describe the desired behavior by a Sequence with memory of pick and place as in Figure 3.12. In this scenario, after picking we can be sure that the object does not slip out of the robot's grippers. Hence while the robot is moving the object, the BT does not need to check if the object is still picked.

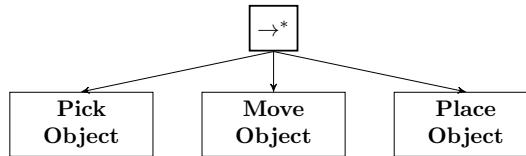


Fig. 3.12: Example of a Un-Reactive Sequence composition of the behaviors pick, move, and place.

3.7 Choosing the Proper Granularity of a BT

In any modular design, we need to decide the granularity of the modules. In a BT framework, this is translated into the choice of what to represent as a leaf node (single action or condition) and what to represent as a BT. The following two cases can be considered.

- It makes sense to encode the behavior in a single leaf when the potential subparts of the behavior are always used and executed in this particular combination.
- It makes sense to encode a behavior as a sub-BT, breaking it up into conditions, actions and flow control nodes, when the subparts are likely to be reusable in other combinations in other parts of the BT, and when the reactivity of BTs can be used to re-execute parts of the behavior when needed.

Consider the BT in Figure 3.13 describing the behavior of a humanoid robot. The actions *sit* and *stand* cannot be divided into meaningful sub-behaviors.

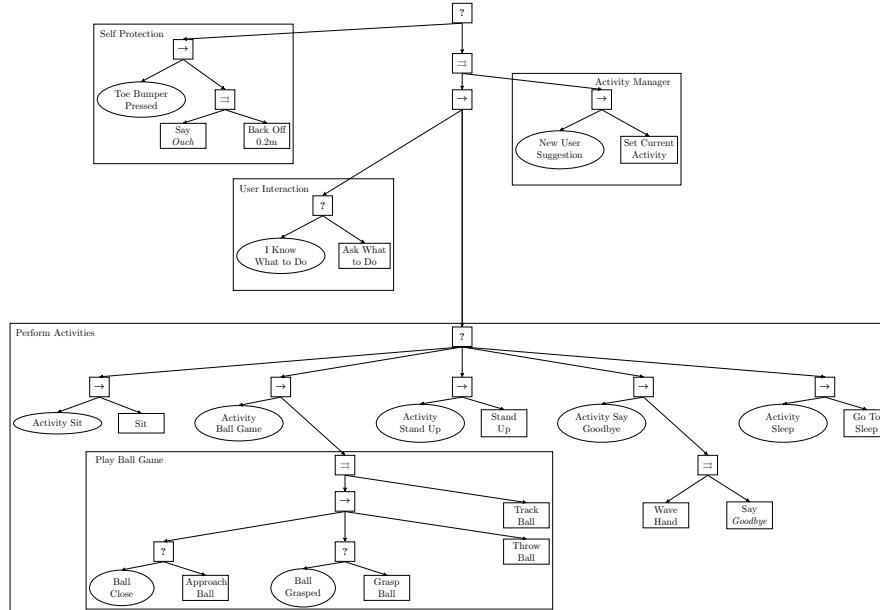


Fig. 3.13: Robot activity manager

Consider an assembly task for an industrial robot that coexists in a semi-structured environment with human workers. The tasks to perform are *pick object*, *assemble object*, and *place object*. A closed-loop execution of this task can be represented with the BT in Figure 3.14. Note that the BT can reactively handle unexpected changes, possibly produced by the human worker in the line, such as when

the worker picks up an object that the robot is trying to reach, or the object slipping out of the robot gripper while the robot is moving it, etc. If we had instead chosen to aggregate the actions *pick object*, *assemble object*, and *place object* into a single action we would lose reactivity when, for example, the robot has to re-pick an assembled object that slipped out from the robot's grippers. With a single action the robot would try to re-assemble an already assembled object.

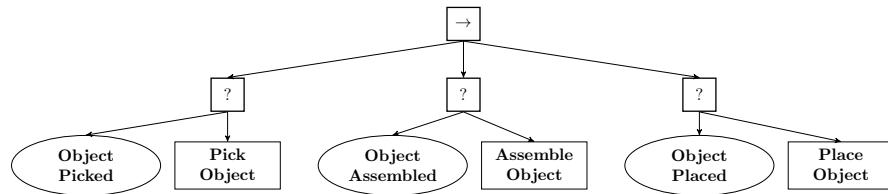


Fig. 3.14: Closed loop example

The advice above should give the designer an idea on how to reach a balanced BT that is neither too *fine grained* nor too *compact*. A fine grained BT might be unreasonably complex. While a compact BT may risk being not sufficiently reactive, by executing too many operations in a feed-forward fashion, losing one main advantage of BTs.

3.8 Putting it all together

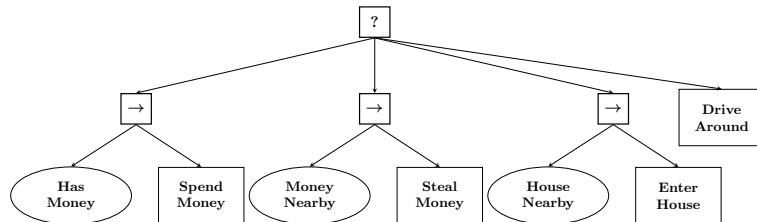


Fig. 3.15: Implicit sequence design of the activities of a burglar.

In this section, we will show how the modularity of BTs make it very straightforward to combine the design principles described in this chapter at different levels of a BT. Imagine we are designing the AI for a game character making a living as a burglar. Its daily life could be filled with stealing and spending money, as described in the BT of Figure 3.15. Note that we have used the *Implicit Sequence* design principle from Section 3.2. The intended progression is driving around until

a promising house is found, enter the house and find indications of money nearby, steal the money and then leave the house to spend the money.

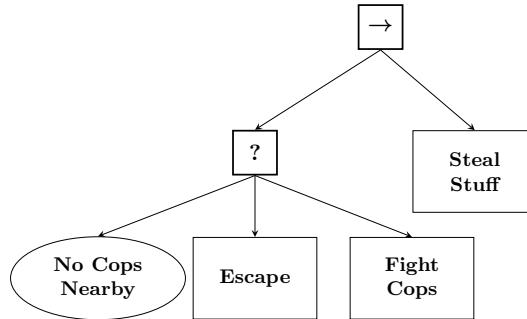


Fig. 3.16: If the escape (or fight) action is efficient enough, this sequence construction will guarantee that the burglar is never caught.

Performing the actions described above, the burglar is also interested in not being captured by the police. Therefore we might design a BT handling when to escape, and when to fight the cops trying to catch it. This might be considered a safety issue, and we can use the design principle for improving safety using sequences, as described in Section 3.4 above. The result might look like the BT in Figure 3.16. If cops are nearby the burglar will first try to escape, and if that fails fight. If anytime during the fight, the escape option is viable, the burglar will switch to escaping.

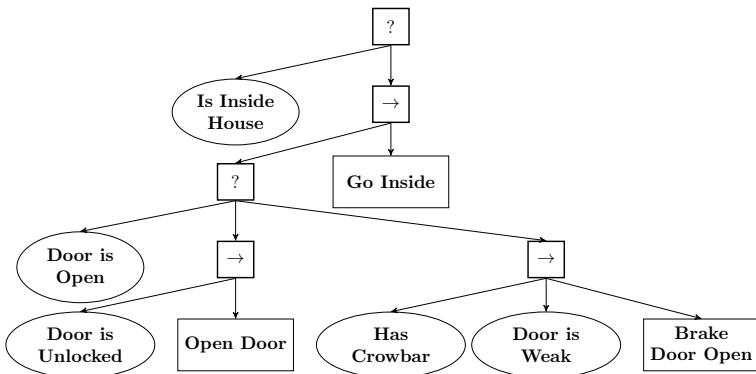


Fig. 3.17: Using backchaining, a BT of desired complexity can be created to get a burglar into a house, this is the same as Figure 3.10.

We saw in Section 3.5 how backchaining could be used to create a BT of the desired complexity for achieving a goal. That same BT is shown here in Figure 3.17 for reference.

Now, the modularity of BTs enable us to combine all these BTs, created with different design principles, into a single, more complex BT, as shown in Figure 3.18. Note that the reactivity of all parts is maintained, and the switches between different sub-BTs happen just the way they should, for example from Drive Around, to Braking a Door Open (when finding a house), to Fighting Cops (when the police arrives and escape is impossible) and then Stealing Money (when police officers are defeated). We will come back to this example in the next chapter on BT extensions.

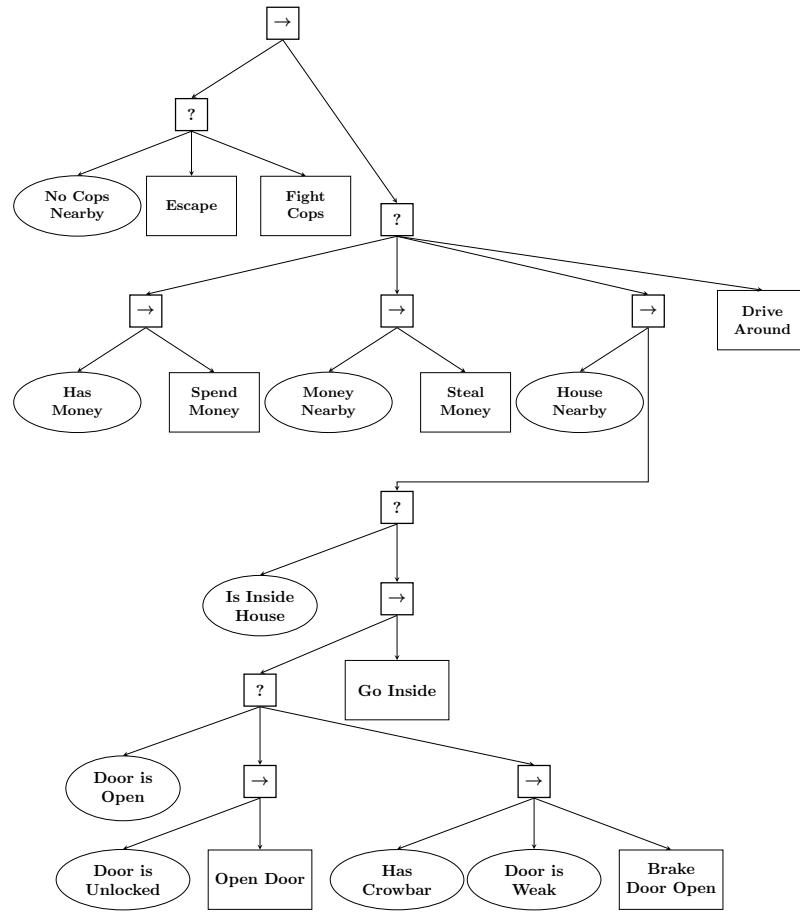


Fig. 3.18: A straightforward combination of the BTs in Figures 3.15, 3.16, and 3.17.

Chapter 4

Extensions of Behavior Trees

As the concept of BT has spread in the AI and robotics communities, a number of extensions have been proposed. Many of them revolve around the Fallback node, and the observation that the ordering of a Fallback node is often somewhat arbitrary. In the nominal case, the children of a Fallback node are different ways of achieving the same outcome, which makes the ordering itself unimportant, but note that this is not the case when Fallbacks are used to increase reactivity with implicit sequences, as described in Section 3.2.

In this chapter, we will describe a number of extensions of the BT concept that have been proposed.

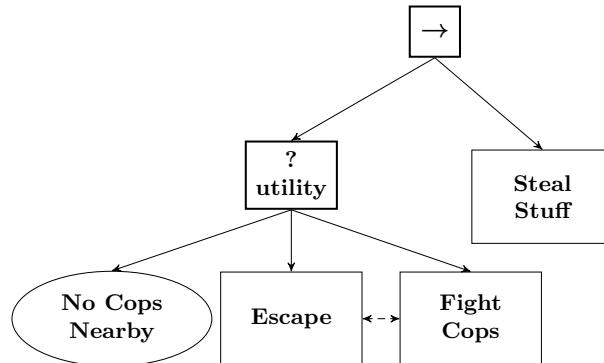


Fig. 4.1: The result of adding a utility Fallback in the BT controlling a burglar game character in Figure 3.16. Note how the Utility node enables a reactive re-ordering of the actions *Escape* and *Fight Cops*.

4.1 Utility BTs

Utility theory is the basic notion that if we can measure the utility of all potential decisions, it would make sense to choose the most useful one. In [42] it was suggested that a utility Fallback node would address what was described as the biggest drawback of BTs, i.e. having fixed priorities in the children of Fallback nodes.

A simple example can be seen in the burglar BT of Figure 4.1. How do we know that escaping is always better than fighting? This is highly dependent on the circumstances, do we have a getaway vehicle, do we have a weapon, how many opponents are there, and what are their vehicles and weapons?

By letting the children of a utility Fallback node return their expected utility, the Fallback node can start with the node of highest utility. Enabling the burglar to escape when a getaway car is available, and fight when having a superior weapon at hand. In [42] it is suggested that all values are normalized to the interval $[0, 1]$ to allow comparison between different actions.

Working with utilities is however not entirely straightforward. One of the core strengths of BTs is the modularity, how single actions are handled in the same way as a large tree. But how do we compute utility for a tree? Two possible solutions exist, either we add Decorators computing utility below every utility Fallback node, or we add a utility estimate in all actions, and create a way to propagate utility up the tree, passing both Fallbacks and Sequences. The former is a bit ad-hoc, while the latter presents some theoretical difficulties.

It is unclear how to aggregate and propagate utility in the tree. It is suggested in [42] to use the max value in both Fallbacks and Sequences. This is reasonable for Fallbacks, as the utility Fallback will prioritize the max utility child and execute it first, but one might also argue that a second Fallback child of almost as high utility should increase overall utility for the Fallback. The max rule is less clear in the Sequence case, as there is no re-ordering, and a high utility child might not be executed due to a failure of another child before it. These difficulties brings us to the next extension, the Stochastic BTs.

4.2 Stochastic BTs

A natural variation of the idea of utilities above is to consider success probabilities, as suggested in [11, 28]. If something needs to be done, the action with the highest success probability might be a good candidate. Before going into details, we note that both costs, execution times, and possible undesired outcomes also matters, but defer this discussion to a later time.

One advantage of considering success probabilities is that the aggregation across both Sequences and Fallbacks is theoretically straightforward. Let P_i^s be the success probability of a given tree, then the probabilities can be aggregated as follows [28]:

$$P_{\text{Sequence}}^s = \Pi_i P_i^s, \quad P_{\text{Fallback}}^s = 1 - \Pi_i (1 - P_i^s), \quad (4.1)$$

since Sequences need all children to succeed, while Fallbacks need only one, with probability equal to the complement of all failing. This is theoretically appealing, but relies on the implicit assumption that each action is only tried once. In a reactive BT for a robot picking and placing items, you could imagine the robot first picking an item, then accidentally dropping it halfway, and then picking it up again. Note that the formulas above do not account for this kind of events.

Now the question comes to how we compute or estimate P_i^s for the individual actions. A natural idea is to learn this from experience [28]. It is reasonable to assume that the success probability of an action, P_i^s , is a function of the world state, so it would make sense to try to learn the success probability as a function of state. Ideally we can classify situations such that one action is known to work in some situations, and another is known to work in others. The continuous maximization of success probabilities in a Fallback node would then make the BT choose the correct action depending on the situation at hand.

There might still be some randomness to the outcomes, and then the following estimate is reasonable

$$P_i^s = \frac{\# \text{ successes}}{\# \text{ trials}}. \quad (4.2)$$

However, this leads to a exploit/explore problem [28]. What if both available actions of a Fallback have high success probability? Initially we try one that works, yielding a good estimate for that action. Then the optimization might continue to favor (exploit) that action, never trying (explore) the other one that might be even better. For the estimates to converge for all actions, even the ones with lower success estimates needs to be executed sometimes. One can also note that having multiple similar robots connected to a cloud service enables much faster learning of both forms of success estimates described above.

It was mentioned above that it might also be relevant to include costs and execution times in the decision of what tree to execute. A formal treatment of both success probabilities and execution times can be found in Chapter 9. A combination of cost and success probabilities might result in a utility system, as described above, but finding the right combination of all three is still an open problem.

4.3 Temporary Modification of BTs

Both in robotics and gaming there is sometimes a need to temporary modify the behavior of a BT. In many robotics applications there is an operator or collaborator that might want to temporarily influence the actions or priorities of a robot. For instance, convincing a service robot to set the table before doing the dirty dishes, or making a delivery drone complete the final mission even though the battery is low enough to motivate an immediate recharge in normal circumstances. In computer games, the

AI is influenced by both level designers, responsible for the player experience, and AI engineers, responsible for agents behaving rationally. Thus, the level designers need a way of making some behaviors more likely, without causing irrational side effects ruining the game experience.

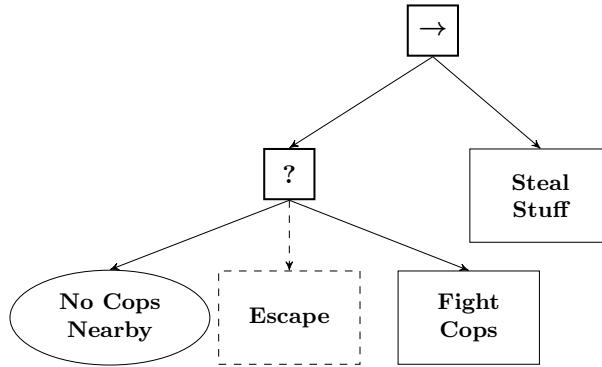


Fig. 4.2: The *aggressive burglar* style, resulting from disabling *Escape* in the BT controlling a burglar game character in Figure 3.16.

This problem was discussed in one of the first papers on BTs [31], with the proposed solutions being *styles*, with each style corresponding to disabling a subset of the BT. For instance, the style *aggressive burglar* might simply have the actions *Escape* disabled, making it disregard injuries and attack until defeated, see Figure 4.2. Similarly, the *Fight* action can be disabled in the *pacifist burglar* style, as shown in Figure 4.3. A more elaborate solution to the same problem can be found in the Hinted BTs described below.

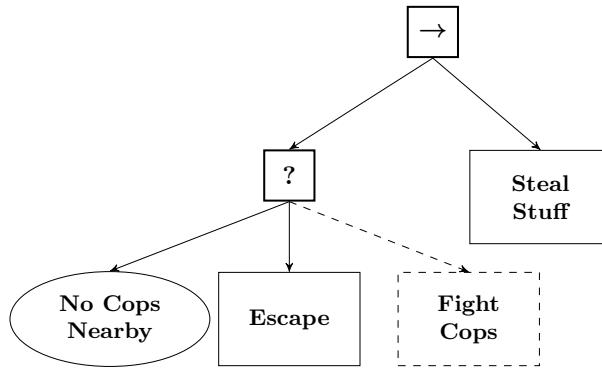


Fig. 4.3: The *pacifist burglar* style, resulting from disabling *Fight* in the BT controlling a burglar game character in Figure 3.16.

Hinted BTs were first introduced in [53, 54]. The key idea is to have an external entity, either human or machine, giving suggestions, so-called *hints*, regarding actions to execute, to a BT. In robotics, the external entity might be an operator or user suggesting something, and in a computer game it might be the level designer wanting to influence the behavior of a character without having to edit the actual BT.

The hints can be both positive (+), in terms of suggested actions, and negative (-), actions to avoid, and a somewhat complex example can be found in Figure 4.4. Multiple hints can be active simultaneously, each influencing the BT in one, or both, of two different ways. First they can effect the ordering of Fallback nodes. Actions or trees with positive hints are moved to the left, and ones with negative hints are moved to the right. Second, the BT is extended with additional conditions, checking if a specific hint is given.

In the BT of Figure 4.4, the following hints were given: *Fight Cops+*, *Brake Door Open+* and *Spend Money-*. *Fight Cops+* makes the burglar first considering the fight option, and only escaping when fighting fails. *Brake Door Open+* makes the burglar try to brake the door, before seeing if it is open or not, and the new corresponding condition makes it ignore the requirements of having a weak door and a crowbar before attempting to brake the door. Finally, *Spend Money-* makes the burglar prefer to drive around looking for promising houses rather than spending money.

4.4 Other extensions of BTs

In this section we will briefly describe a number of additional suggested extensions of BTs.

4.4.1 Dynamic Expansion of BTs

The concept of Dynamic Expansions was suggested in [17]. Here, the basic idea is to let the BT designer leave some details of the BT to a run-time search. To enable that search, some desired features of the action needed are specified, these include the category, given a proposed behavior taxonomy, including *Attack*, *Defend*, *Hunt*, and *Move*. The benefit of the proposed approach is that newly created actions can be used in BTs that were created before the actions, as long as the BTs have specified the desired features that the new action should have.

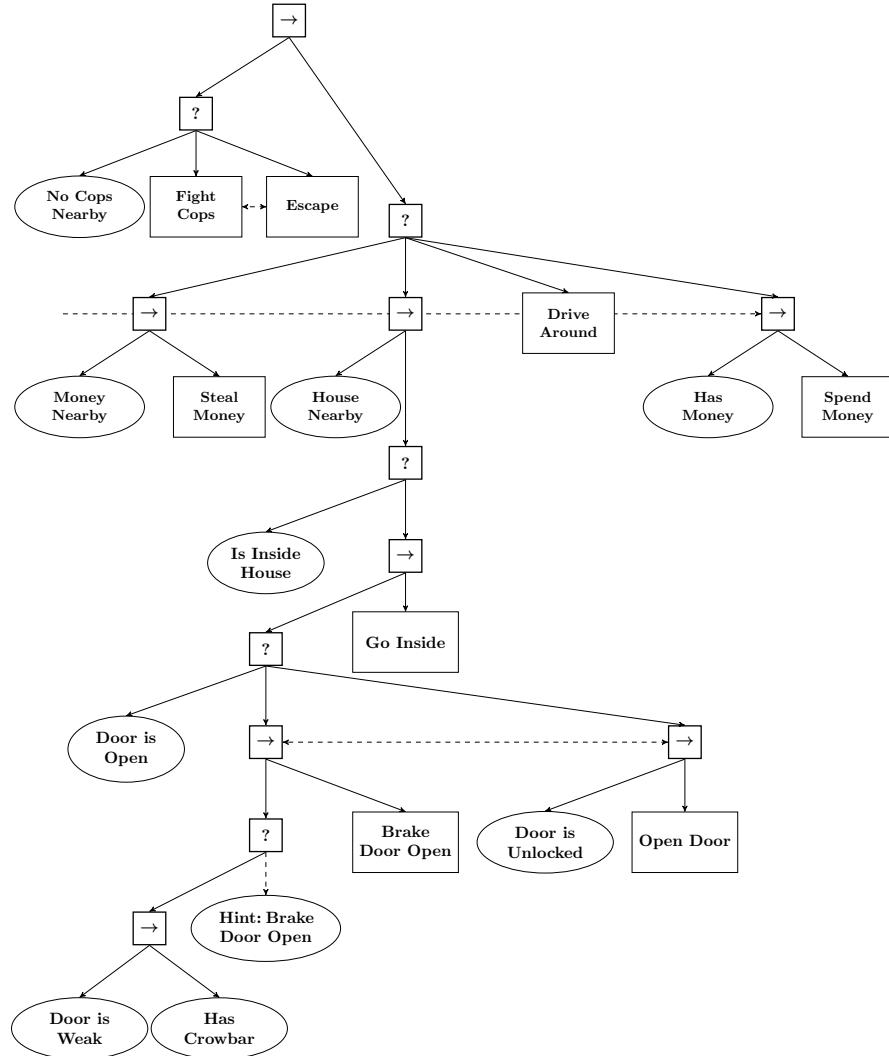


Fig. 4.4: The result of providing the hints *Fight Cops+*, *Brake Door Open+* and *Spend Money-* to the BT in Figure 3.18. The dashed arrows indicated changes in the BT.

Chapter 5

Analysis of Efficiency, Safety, and Robustness

Autonomous agents will need to be efficient, robust, and reliable in order to be used on a large scale. In this chapter, we present a mathematical framework for analyzing these properties for a BT (Section 5.1). The analysis includes efficiency (Section 5.2), in terms of execution time bounds; robustness (Section 5.2), in terms of capability to operate in large domains; and safety (Section 5.3), in terms of avoiding some particular parts of the state space. Some of the results of this chapter were previously published in the journal paper [13].

5.1 Statespace Formulation of BTs

In this section, we present a new formulation of BTs. The new formulation is more formal, and will allow us to analyze how properties are preserved over modular compositions of BTs. In the functional version, the *tick* is replaced by a recursive function call that includes both the return status, the system dynamics and the system state.

Definition 5.1 (Behavior Tree). A BT is a three-tuple

$$\mathcal{T}_i = \{f_i, r_i, \Delta t\}, \quad (5.1)$$

where $i \in \mathbb{N}$ is the index of the tree, $f_i : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is the right hand side of an ordinary difference equation, Δt is a time step and $r_i : \mathbb{R}^n \rightarrow \{\mathcal{R}, \mathcal{S}, \mathcal{F}\}$ is the return status that can be equal to either *Running* (\mathcal{R}), *Success* (\mathcal{S}), or *Failure* (\mathcal{F}). Let the Running/Activation region (R_i), Success region (S_i) and Failure region (F_i) correspond to a partitioning of the state space, defined as follows:

$$R_i = \{x : r_i(x) = \mathcal{R}\} \quad (5.2)$$

$$S_i = \{x : r_i(x) = \mathcal{S}\} \quad (5.3)$$

$$F_i = \{x : r_i(x) = \mathcal{F}\}. \quad (5.4)$$

Finally, let $x_k = x(t_k)$ be the system state at time t_k , then the execution of a BT \mathcal{T}_i is a standard ordinary difference equation

$$x_{k+1} = f_i(x_k), \quad (5.5)$$

$$t_{k+1} = t_k + \Delta t. \quad (5.6)$$

The return status r_i will be used when combining BTs recursively, as explained below.

Assumption 5.1 *From now on we will assume that all BTs evolve in the same continuous space \mathbb{R}^n using the same time step Δt .*

Remark 5.1. It is often the case, that different BTs, controlling different vehicle subsystems evolving in different state spaces, need to be combined into a single BT. Such cases can be accommodated in the assumption above by letting all systems evolve in a larger state space, that is the Cartesian product of the smaller state spaces.

Definition 5.2 (Sequence compositions of BTs). Two or more BTs can be composed into a more complex BT using a Sequence operator,

$$\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2).$$

Then r_0, f_0 are defined as follows

$$\text{If } x_k \in S_1 \quad (5.7)$$

$$r_0(x_k) = r_2(x_k) \quad (5.8)$$

$$f_0(x_k) = f_2(x_k) \quad (5.9)$$

else

$$r_0(x_k) = r_1(x_k) \quad (5.10)$$

$$f_0(x_k) = f_1(x_k). \quad (5.11)$$

\mathcal{T}_1 and \mathcal{T}_2 are called children of \mathcal{T}_0 . Note that when executing the new BT, \mathcal{T}_0 first keeps executing its first child \mathcal{T}_1 as long as it returns Running or Failure. The second child is executed only when the first returns Success, and \mathcal{T}_0 returns Success only when all children have succeeded, hence the name Sequence, just as the classical definition of Sequences in Algorithm 1 of Section 1.3.

For notational convenience, we write

$$\text{Sequence}(\mathcal{T}_1, \text{Sequence}(\mathcal{T}_2, \mathcal{T}_3)) = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3), \quad (5.12)$$

and similarly for arbitrarily long compositions.

Definition 5.3 (Fallback compositions of BTs). Two or more BTs can be composed into a more complex BT using a Fallback operator,

$$\mathcal{T}_0 = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2).$$

Then r_0, f_0 are defined as follows

$$\text{If } x_k \in F_1 \quad (5.13)$$

$$r_0(x_k) = r_2(x_k) \quad (5.14)$$

$$f_0(x_k) = f_2(x_k) \quad (5.15)$$

else

$$r_0(x_k) = r_1(x_k) \quad (5.16)$$

$$f_0(x_k) = f_1(x_k). \quad (5.17)$$

Note that when executing the new BT, \mathcal{T}_0 first keeps executing its first child \mathcal{T}_1 as long as it returns Running or Success. The second child is executed only when the first returns Failure, and \mathcal{T}_0 returns Failure only when all children have tried, but failed, hence the name Fallback, just as the classical definition of Fallbacks in Algorithm 2 of Section 1.3.

For notational convenience, we write

$$\text{Fallback}(\mathcal{T}_1, \text{Fallback}(\mathcal{T}_2, \mathcal{T}_3)) = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3), \quad (5.18)$$

and similarly for arbitrarily long compositions.

Parallel compositions only make sense if the BTs to be composed control separate parts of the state space, thus we make the following assumption.

Assumption 5.2 Whenever two BTs $\mathcal{T}_1, \mathcal{T}_2$ are composed in parallel, we assume that there is a partition of the state space $x = (x_1, x_2)$ such that $f_1(x) = (f_{11}(x), f_{12}(x))$ implies $f_{12}(x) = x$ and $f_2(x) = (f_{21}(x), f_{22}(x))$ implies $f_{21}(x) = x$ (i.e. the two BTs control different parts of the system).

Definition 5.4 (Parallel compositions of BTs). Two or more BTs can be composed into a more complex BT using a Parallel operator,

$$\mathcal{T}_0 = \text{Parallel}(\mathcal{T}_1, \mathcal{T}_2).$$

Let $x = (x_1, x_2)$ be the partitioning of the state space described in Assumption 5.2, then $f_0(x) = (f_{11}(x), f_{22}(x))$ and r_0 is defined as follows

If $M = 1$

$$r_0(x) = \mathcal{S} \text{ If } r_1(x) = \mathcal{S} \vee r_2(x) = \mathcal{S} \quad (5.19)$$

$$r_0(x) = \mathcal{F} \text{ If } r_1(x) = \mathcal{F} \wedge r_2(x) = \mathcal{F} \quad (5.20)$$

$$r_0(x) = \mathcal{R} \text{ else} \quad (5.21)$$

If $M = 2$

$$r_0(x) = \mathcal{S} \text{ If } r_1(x) = \mathcal{S} \wedge r_2(x) = \mathcal{S} \quad (5.22)$$

$$r_0(x) = \mathcal{F} \text{ If } r_1(x) = \mathcal{F} \vee r_2(x) = \mathcal{F} \quad (5.23)$$

$$r_0(x) = \mathcal{R} \text{ else} \quad (5.24)$$

5.2 Efficiency and Robustness

In this section we will show how some aspects of time efficiency and robustness carry across modular compositions of BTs. This result will then enable us to conclude, that if two BTs are efficient, then their composition will also be *efficient*, if the right conditions are satisfied. We also show how the Fallback composition can be used to increase the region of attraction of a BT, thereby making it more robust to uncertainties in the initial configuration.

Note that, as in [8], by robustness we mean large regions of attraction. We do not investigate e.g. disturbance rejection, or other forms of robustness¹

Many control problems, in particular in robotics, can be formulated in terms of achieving a given goal configuration in a way that is time efficient and robust with respect to the initial configuration. Since all BTs return either Success, Failure, or Running, the definitions below will include a finite time, at which Success must be returned.

In order to formalize the discussion above, we say that *efficiency* can be measured by the size of a time bound τ in Definition 5.5 and *robustness* can be measured by the size of the region of attraction R' in the same definition.

Definition 5.5 (Finite Time Successful). A BT is Finite Time Successful (FTS) with region of attraction R' , if for all starting points $x(0) \in R' \subset R$, there is a time τ , and a time $\tau'(x(0))$ such that $\tau'(x) \leq \tau$ for all starting points, and $x(t) \in R'$ for all $t \in [0, \tau']$ and $x(t) \in S$ for $t = \tau'$

As noted in the following lemma, exponential stability implies FTS, given the right choices of the sets S, F, R .

Lemma 5.1 (Exponential stability and FTS). A BT for which x_s is a globally exponentially stable equilibrium of the execution (5.5), and $S \supset \{x : \|x - x_s\| \leq \varepsilon\}$, $\varepsilon > 0$, $F = \emptyset$, $R = \mathbb{R}^n \setminus S$, is FTS.

¹ Both meanings of robustness are aligned with the IEEE standard glossary of software engineering terminology: “The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.”

Proof. Global exponential stability implies that there exists $a > 0$ such that $\|x(k) - x_s\| \leq e^{-ak}$ for all k . Then, for each ϵ there is a time τ such that $\|x(k) - x_s\| \leq e^{-a\tau} < \epsilon$, which implies that there is a $\tau' < \tau$ such that $x(\tau') \in S$ and the BT is FTS.

We are now ready to look at how these properties extend across compositions of BTs.

Lemma 5.2. (Robustness and Efficiency of Sequence Compositions) *If $\mathcal{T}_1, \mathcal{T}_2$ are FTS, with $S_1 = R'_2 \cup S_2$, then $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$ is FTS with $\tau_0 = \tau_1 + \tau_2$, $R'_0 = R'_1 \cup R'_2$ and $S_0 = S_1 \cap S_2 = S_2$.*

Proof. First we consider the case when $x(0) \in R'_1$. Then, as \mathcal{T}_1 is FTS, the state will reach S_1 in a time $k_1 < \tau_1$, without leaving R'_1 . Then \mathcal{T}_2 starts executing, and will keep the state inside S_1 , since $S_1 = R'_2 \cup S_2$. \mathcal{T}_2 will then bring the state into S_2 , in a time $k_2 < \tau_2$, and \mathcal{T}_0 will return Success. Thus we have the combined time $k_1 + k_2 < \tau_1 + \tau_2$.

If $x(0) \in R'_2$, \mathcal{T}_1 immediately returns Success, and \mathcal{T}_2 starts executing as above.

The lemma above is illustrated in Figure 5.2, and Example 5.1 below.

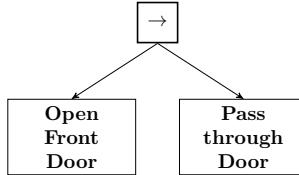


Fig. 5.1: A Sequence is used to create an *Enter Through Front Door* BT. Passing the door is only tried if the opening action succeeds. Sequences are denoted by a white box with an arrow.

Example 5.1. Consider the BT in Figure 5.1. If we know that *Open Front Door* is FTS and will finish in less than τ_1 seconds, and that *Pass through Door* is FTS and will finish in less than τ_2 seconds. Then, as long as $S_1 = R'_2 \cup S_2$, Lemma 5.2 states that the combined BT in Figure 5.1 is also FTS, with an upper bound on the execution time of $\tau_1 + \tau_2$. Note that the condition $S_1 = R'_2 \cup S_2$ implies that the action *Pass through Door* will not make the system leave S_1 , by e.g. accidentally colliding with the door and thereby closing it without having passed through it.

The result for Fallback compositions is related, but with a slightly different condition on S_i and R'_j . Note that this is the theoretical underpinning of the design principle *Implicit Sequences* described in Section 3.2.

Lemma 5.3. (Robustness and Efficiency of Fallback Compositions) *If $\mathcal{T}_1, \mathcal{T}_2$ are FTS, with $S_2 \subset R'_1$ and $R_1 = R'_1$, then $\mathcal{T}_0 = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2)$ is FTS with $\tau_0 = \tau_1 + \tau_2$, $R'_0 = R'_1 \cup R'_2$ and $S_0 = S_1$.*

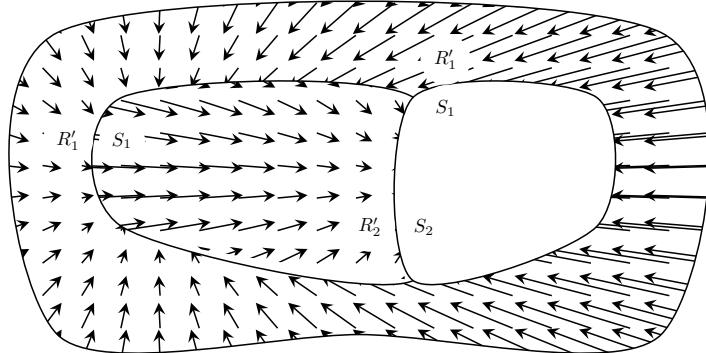


Fig. 5.2: The sets R'_1, S_1, R'_2, S_2 of Example 5.1 and Lemma 5.2.

Proof. First we consider the case when $x(0) \in R'_1$. Then, as \mathcal{T}_1 is FTS, the state will reach S_1 before $k = \tau_1 < \tau_0$, without leaving R'_1 . If $x(0) \in R'_2 \setminus R'_1$, \mathcal{T}_2 will execute, and the state will progress towards S_2 . But as $S_2 \subset R'_1$, $x(k_1) \in R'_1$ at some time $k_1 < \tau_2$. Then, we have the case above, reaching $x(k_2) \in S_1$ in a total time of $k_2 < \tau_1 + k_1 < \tau_1 + \tau_2$.

The Lemma above is illustrated in Figure 5.3, and Example 5.2 below.

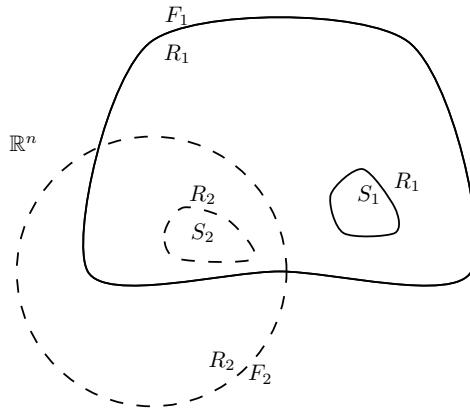


Fig. 5.3: The sets S_1, F_1, R_1 (solid boundaries) and S_2, F_2, R_2 (dashed boundaries) of Example 5.2 and Lemma 5.3.

Remark 5.2. As can be noted, the necessary conditions in Lemma 5.2, including $S_1 = R'_2 \cup S_2$ might be harder to satisfy than the conditions of Lemma 5.3, in-

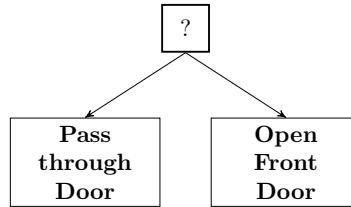


Fig. 5.4: An Implicit Sequence created using a Fallback, as described in Example 5.2 and Lemma 5.3.

cluding $S_2 \subset R'_1$. Therefore, Lemma 5.3 is often preferable from a practical point of view, e.g. using implicit sequences as shown below.

Example 5.2. This example will illustrate the design principle *Implicit sequences* of Section 3.2. Consider the BT in Figure 5.4. During execution, if the door is closed, then *Pass through Door* will fail and *Open Front Door* will start to execute. Now, right before *Open Front Door* returns Success, the first action *Pass through Door* (with higher priority) will realize that the state of the world has now changed enough to enable a possible success and starts to execute, i.e. return Running instead of Failure. The combined action of this BT will thus make the robot open the door (if necessary) and then pass through it.

Thus, even though a Fallback composition is used, the result is sometimes a sequential execution of the children in reverse order (from right to left). Hence the name Implicit sequence.

The example above illustrates how we can increase the robustness of a BT. If we want to be able to handle more diverse situations, such as a closed door, we do not have to make the door passing action more complex, instead we combine it with another BT that can handle the situation and move the system into a part of the statespace that the first BT can handle. The sets S_0, F_0, R_0 and f_0 of the combined BT are shown in Figure 5.5, together with the vector field $f_0(x) - x$. As can be seen, the combined BT can now move a larger set of initial conditions to the desired region $S_0 = S_1$.

Lemma 5.4. (Robustness and Efficiency of Parallel Compositions) *If $\mathcal{T}_1, \mathcal{T}_2$ are FTS, then $\mathcal{T}_0 = \text{Parallel}(\mathcal{T}_1, \mathcal{T}_2)$ is FTS with*

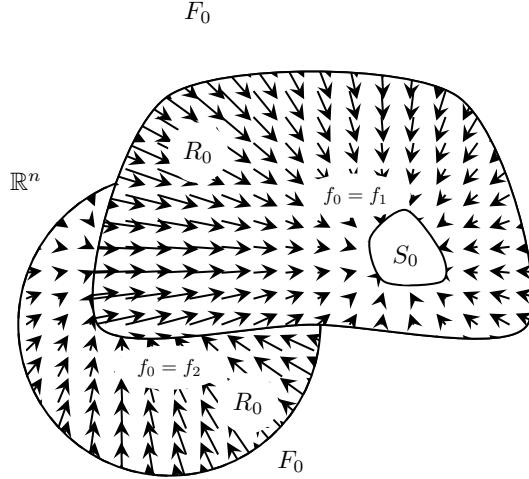


Fig. 5.5: The sets S_0, F_0, R_0 and the vector field $(f_0(x) - x)$ of Example 5.2 and Lemma 5.3.

If $M = 1$

$$R'_0 = \{R'_1 \cup R'_2\} \setminus \{S_1 \cup S_2\} \quad (5.25)$$

$$S_0 = S_1 \cup S_2 \quad (5.26)$$

$$\tau_0 = \min(\tau_1, \tau_2) \quad (5.27)$$

If $M = 2$

$$R'_0 = \{R'_1 \cap R'_2\} \setminus \{S_1 \cap S_2\} \quad (5.28)$$

$$S_0 = S_1 \cap S_2 \quad (5.29)$$

$$\tau_0 = \max(\tau_1, \tau_2) \quad (5.30)$$

Proof. The Parallel composition executes \mathcal{T}_1 and \mathcal{T}_2 independently. If $M = 1$ the Parallel composition returns Success if either \mathcal{T}_1 or \mathcal{T}_2 returns Success, thus $\tau_0 = \min(\tau_1, \tau_2)$. It returns Running if either \mathcal{T}_1 or \mathcal{T}_2 returns Running and the other does not return Success. If $M = 2$ the Parallel composition returns Success if and only if both \mathcal{T}_1 and \mathcal{T}_2 return Success, thus $\tau_0 = \max(\tau_1, \tau_2)$. It returns Running if either \mathcal{T}_1 or \mathcal{T}_2 returns Running and the other does not return Failure.

5.3 Safety

In this section we will show how some aspects of safety carry across modular compositions of BTs. The results will enable us to design a BT to handle safety guarantees and a BT to handle the task execution separately.

In order to formalize the discussion above, we say that *safety* can be measured by the ability to avoid a particular part of the statespace, which we for simplicity denote the *Obstacle Region*.

Definition 5.6 (Safe). A BT is safe, with respect to the obstacle region $O \subset \mathbb{R}^n$, and the initialization region $I \subset R$, if for all starting points $x(0) \in I$, we have that $x(t) \notin O$, for all $t \geq 0$.

In order to make statements about the safety of composite BTs we also need the following definition.

Definition 5.7 (Safeguarding). A BT is safeguarding, with respect to the step length d , the obstacle region $O \subset \mathbb{R}^n$, and the initialization region $I \subset R$, if it is safe, and FTS with region of attraction $R' \supset I$ and a success region S , such that I surrounds S in the following sense:

$$\{x \in X \subset \mathbb{R}^n : \inf_{s \in S} \|x - s\| \leq d\} \subset I, \quad (5.31)$$

where X is the reachable part of the state space \mathbb{R}^n .

This implies that the system, under the control of another BT with maximal statespace steplength d , cannot leave S without entering I , and thus avoiding O , see Lemma 5.5 below.

Example 5.3. To illustrate how safety can be improved using a Sequence composition, we consider the UAV control BT in Figure 5.6. The sets S_i, F_i, R_i are shown in Figure 5.7. As \mathcal{T}_1 is *Guarantee altitude above 1000 ft*, its failure region F_1 is a small part of the state space (corresponding to a crash) surrounded by the running region R_1 that is supposed to move the UAV away from the ground, guaranteeing a minimum altitude of 1000 ft. The success region S_1 is large, every state sufficiently distant from F_1 . The BT that performs the mission, \mathcal{T}_2 , has a smaller success region S_2 , surrounded by a very large running region R_2 , containing a small failure region F_2 . The function f_0 is governed by Equations (5.9) and (5.11) and is depicted in form of the vector field $(f_0(x) - x)$ in Figure 5.8.

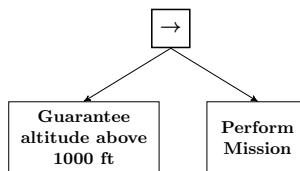


Fig. 5.6: The Safety of the UAV control BT is Guaranteed by the first Action.

The discussion above is formalized in Lemma 5.5 below.

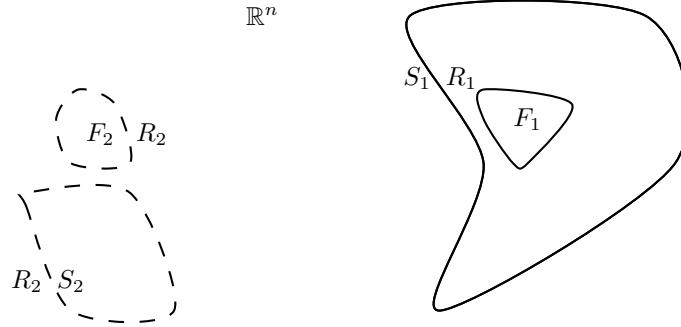


Fig. 5.7: The sets S_1, F_1, R_1 (solid boundaries) and S_2, F_2, R_2 (dashed boundaries) of Example 5.3 and Lemma 5.5.

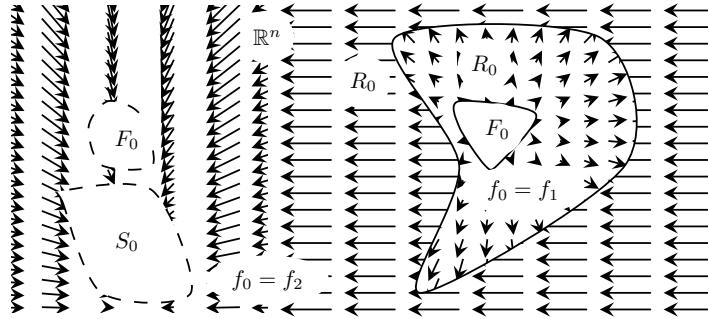


Fig. 5.8: The sets S_0, F_0, R_0 and the vector field $(f_0(x) - x)$ of Example 5.3 and Lemma 5.5.

Lemma 5.5 (Safety of Sequence Compositions). *If \mathcal{T}_1 is safeguarding, with respect to the obstacle O_1 initial region I_1 , and margin d , and \mathcal{T}_2 is an arbitrary BT with $\max_x \|x - f_2(x)\| < d$, then the composition $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$ is safe with respect to O_1 and I_1 .*

Proof. \mathcal{T}_1 is safeguarding, which implies that \mathcal{T}_1 is safe and thus any trajectory starting in I_1 will stay out of O_1 as long as \mathcal{T}_1 is executing. But if the trajectory reaches S_1 , \mathcal{T}_2 will execute until the trajectory leaves S_1 . We must now show that the trajectory cannot reach O_1 without first entering I_1 . But any trajectory leaving S_1 must immediately enter I_1 , as the first state outside S_1 must lie in the set $\{x \in \mathbb{R}^n : \inf_{s \in S_1} \|x - s\| \leq d\} \subset I_1$ due to the fact that for \mathcal{T}_2 , $\|x(k) - x(k+1)\| = \|x(k) - f_2(x(k))\| < d$.

We conclude this section with a discussion about undesired chattering in switching systems.

The issue of undesired chattering, i.e., switching back and fourth between different subcontrollers, is always an important concern when designing switched control

systems, and BTs are no exception. As is suggested by the right part of Figure 5.8, chattering can be a problem when vector fields meet at a switching surface.

Although the efficiency of some compositions can be computed using Lemma 5.2 and 5.3 above, chattering can significantly reduce the efficiency of others. Inspired by [16] the following result can give an indication of when chattering is to be expected.

Let R_i and R_j be the running region of \mathcal{T}_i and \mathcal{T}_j respectively. We want to study the behavior of the system when a composition of \mathcal{T}_i and \mathcal{T}_j is applied. In some cases the execution of a BT will lead to the running region of the other BT and vice-versa. Then, both BTs are alternatively executed and the state trajectory chatters on the boundary between R_i and R_j . We formalize this discussion in the following lemma.

Lemma 5.6. *Given a composition $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$, where f_i depend on Δt such that $\|f_i(x) - x\| \rightarrow 0$ when $\Delta t \rightarrow 0$. Let $s : \mathbb{R}^n \rightarrow \mathbb{R}$ be such that $s(x) = 0$ if $x \in \delta S_1 \cap R_2$, $s(x) < 0$ if $x \in \text{interior}(S_1) \cap R_2$, $s(x) > 0$ if $x \in \text{interior}(\mathbb{R}^n \setminus S_1) \cap R_2$, and let*

$$\lambda_i(x) = \left(\frac{\partial s}{\partial x} \right)^T (f_i(x) - x).$$

Then, $x \in \delta S_1$ is chatter free, i.e., avoids switching between \mathcal{T}_1 and \mathcal{T}_2 at every timestep, for small enough Δt , if $\lambda_1(x) < 0$ or $\lambda_2(x) > 0$.

Proof. When the condition holds, the vector field is pointing outwards on at least one side of the switching boundary.

Note that this condition is not satisfied on the right hand side of Figure 5.8. This concludes our analysis of BT compositions.

5.4 Examples

In this section, we show some BTs of example and we analyze their properties.

Section 5.4.1 Illustrates how to analyze robustness and efficiency of a robot executing a generic task. Section 5.3 illustrates to compute safety using the functional representation of Section 5.1. Section 9.4 illustrate how to compute the performance estimate of a given BT. Finally, Section 5.4.3 illustrate the properties above of a complex BT .

All BTs were implemented using the ROS BT library.² A video showing the executions of the BTs used in Sections 5.4.2-5.4.1 is publicly available.³

² library available at http://wiki.ros.org/behavior_tree.

³ <https://youtu.be/fH7jx4ZsTG8>

5.4.1 Robustness and Efficiency

To illustrate Lemma 5.3 we look at the BT of Figure 5.9 controlling a humanoid robot. The BT has three subtrees *Walk Home*, which is first tried, if that fails (the robot cannot walk if it is not standing up) it tries the subtree *Sit to Stand*, and if that fails, it tries *Lie down to Sit Up*. Thus, each fallback action brings the system into the running region of the action to its left, e.g., the result of *Sit to Stand* is to enable the execution of *Walk Home*.

Example 5.4. Let $x = (x_1, x_2) \in \mathbb{R}^2$, where $x_1 \in [0, 0.5]$ is the horizontal position of the robot head and $x_2 \in [0, 0.55]$ is vertical position (height above the floor) of the robot head. The objective of the robot is to get to the destination at $(0, 0.48)$.

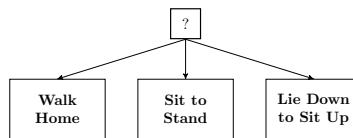


Fig. 5.9: The combination $\mathcal{T}_3 = \text{Fallback}(\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6)$ increases robustness by increasing the region of attraction.

First we describe the sets S_i, F_i, R_i and the corresponding vector fields of the functional representation. Then we apply Lemma 5.3 to see that the combination does indeed improve robustness. For this example $\Delta t = 1s$.

For *Walk Home*, \mathcal{T}_4 , we have that

$$S_4 = \{x : x_1 \leq 0\} \quad (5.32)$$

$$R_4 = \{x : x_1 \neq 0, x_2 \geq 0.48\} \quad (5.33)$$

$$F_4 = \{x : x_1 \neq 0, x_2 < 0.48\} \quad (5.34)$$

$$f_4(x) = \begin{pmatrix} x_1 - 0.1 \\ x_2 \end{pmatrix} \quad (5.35)$$

that is, it runs as long as the vertical position of the robot head, x_2 , is at least $0.48m$ above the floor, and moves towards the origin with a speed of $0.1m/s$. If the robot is not standing up $x_2 < 0.48m$ it returns Failure. A phase portrait of $f_4(x) - x$ is shown in Figure 5.10. Note that \mathcal{T}_4 is FTS with the completion time bound $\tau_4 = 0.5/0.1 = 10$ and region of attraction $R'_4 = R_4$.

For *Sit to Stand*, \mathcal{T}_5 , we have that

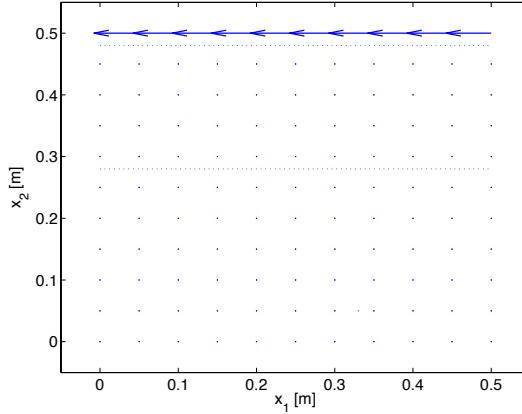


Fig. 5.10: The Action *Walk Home*, keeps the head around $x_2 = 0.5$ and moves it towards the destination $x_1 = 0$.

$$S_5 = \{x : 0.48 \leq x_2\} \quad (5.36)$$

$$R_5 = \{x : 0.3 \leq x_2 < 0.48\} \quad (5.37)$$

$$F_5 = \{x : x_2 < 0.3\} \quad (5.38)$$

$$f_5(x) = \begin{pmatrix} x_1 \\ x_2 + 0.05 \end{pmatrix} \quad (5.39)$$

that is, it runs as long as the vertical position of the robot head, x_2 , is in between $0.3m$ and $0.48m$ above the floor. If $0.48 \leq x_2$ the robot is standing up, and it returns Success. If $x_2 \leq 0.3$ the robot is lying down, and it returns Failure. A phase portrait of $f_5(x) - x$ is shown in Figure 5.11. Note that \mathcal{T}_5 is FTS with the completion time bound $\tau_5 = \text{ceil}(0.18/0.05) = \text{ceil}(3.6) = 4$ and region of attraction $R'_5 = R_5$

For *Lie down to Sit Up*, \mathcal{T}_6 , we have that

$$S_6 = \{x : 0.3 \leq x_2\} \quad (5.40)$$

$$R_6 = \{x : 0 \leq x_2 < 0.3\} \quad (5.41)$$

$$F_6 = \emptyset \quad (5.42)$$

$$f_6(x) = \begin{pmatrix} x_1 \\ x_2 + 0.03 \end{pmatrix} \quad (5.43)$$

that is, it runs as long as the vertical position of the robot head, x_2 , is below $0.3m$ above the floor. If $0.3 \leq x_2$ the robot is sitting up (or standing up), and it returns Success. If $x_2 < 0.3$ the robot is lying down, and it returns Running. A phase portrait of $f_6(x) - x$ is shown in Figure 5.12. Note that \mathcal{T}_6 is FTS with the completion time bound $\tau_6 = 0.3/0.03 = 10$ and region of attraction $R'_6 = R_6$

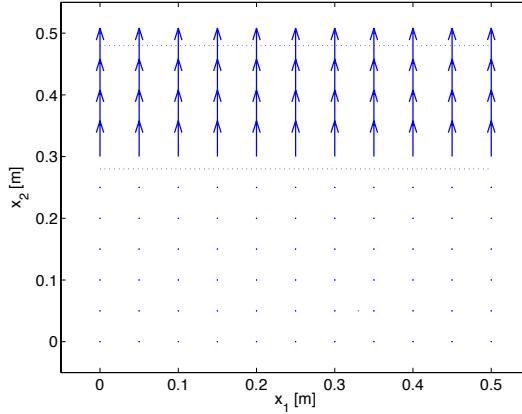


Fig. 5.11: The Action *Sit to Stand* moves the head upwards in the vertical direction towards standing.

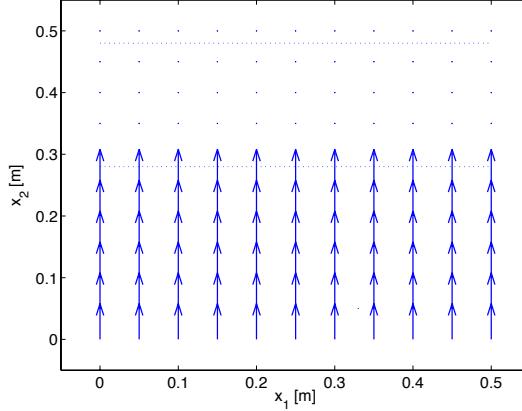


Fig. 5.12: The Action *Lie down to Sit Up* moves the head upwards in the vertical direction towards sitting.

Informally, we can look at the phase portrait in Figure 5.13 to get a feeling for what is going on. As can be seen the Fallbacks make sure that the robot gets on its feet and walks back, independently of where it started in $\{x : 0 < x_1 \leq 0.5, 0 \leq x_2 \leq 0.55\}$.

Formally, we can use Lemma 5.3 to compute robustness in terms of the region of attraction R'_3 , and efficiency in terms of bounds on completion time τ_3 . The results are described in the following Lemma.

Lemma 5.7. *Given $\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6$ defined in Equations (5.32)-(5.43).*

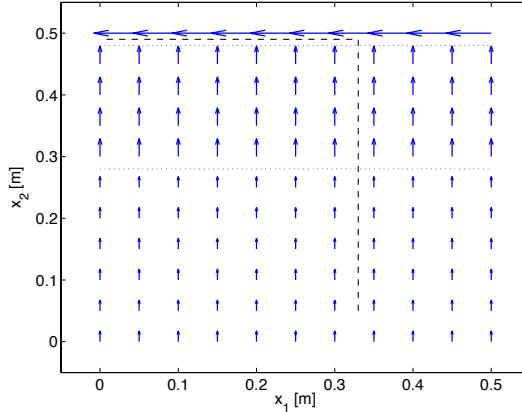


Fig. 5.13: The combination Fallback($\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6$) first gets up, and then walks home.

The combined BT $\mathcal{T}_3 = \text{Fallback}(\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6)$ is FTS, with region of attraction $R'_3 = \{x : 0 < x_1 \leq 0.5, 0 \leq x_2 \leq 0.55\}$, completion time bound $\tau_3 = 24$.

Proof. We note that $\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6$ are FTS with $\tau_4 = 10, \tau_5 = 4, \tau_6 = 10$ and regions of attractions equal to the running regions $R'_i = R_i$. Thus we have that $S_6 \subset R_5 = R'_5$ and $S_5 \subset R_4 = R'_4$. Applying Lemma 5.3 twice now gives the desired results, $R'_3 = R'_4 \cup R'_5 \cup R'_6 = \{x : 0 \leq x_1 \leq 0.5, 0 \leq x_2 \leq 0.55\}$ and $\tau_3 = \tau_4 + \tau_5 + \tau_6 = 10 + 4 + 10 = 24$.

5.4.2 Safety

To illustrate Lemma 5.5 we choose the BT of Figure 5.14. The idea is that the first subtree in the Sequence (named *Guarantee Power Supply*) is to guarantee that the combination does not run out of power, under very general assumptions about what is going on in the second BT.

First we describe the sets S_i, F_i, R_i and the corresponding vector fields of the functional representation. Then we apply Lemma 5.5 to see that the combination does indeed guarantee against running out of batteries.

Example 5.5. Let \mathcal{T}_1 be *Guarantee Power Supply* and \mathcal{T}_2 be *Do other tasks*. Let furthermore $x = (x_1, x_2) \in \mathbb{R}^2$, where $x_1 \in [0, 100]$ is the distance from the current position to the recharging station and $x_2 \in [0, 100]$ is the battery level. For this example $\Delta t = 10s$.

For *Guarantee Power Supply*, \mathcal{T}_1 , we have that

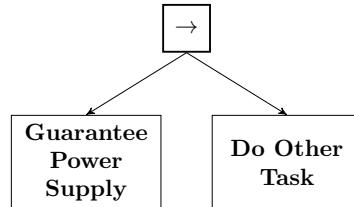


Fig. 5.14: A BT where the first action guarantees that the combination does not run out of battery.

$$S_1 = \{x : 100 \leq x_2 \text{ or } (0.1 \leq x_1, 20 < x_2)\} \quad (5.44)$$

$$R_1 = \{x : x_2 \leq 20 \text{ or } (x_2 < 100 \text{ and } x_1 < 0.1)\} \quad (5.45)$$

$$F_1 = \emptyset \quad (5.46)$$

$$f_1(x) = \begin{cases} x_1 \\ x_2 + 1 \end{cases} \text{ if } x_1 < 0.1, x_2 < 100 \quad (5.47)$$

$$= \begin{cases} x_1 - 1 \\ x_2 - 0.1 \end{cases} \text{ else} \quad (5.48)$$

that is, when running, the robot moves to $x_1 < 0.1$ and recharges. While moving, the battery level decreases and while charging the battery level increases. If at the recharge position, it returns Success only after reaching $x_2 \geq 100$. Outside of the recharge area, it returns Success as long as the battery level is above 20%. A phase portrait of $f_1(x) - x$ is shown in Figure 5.15.

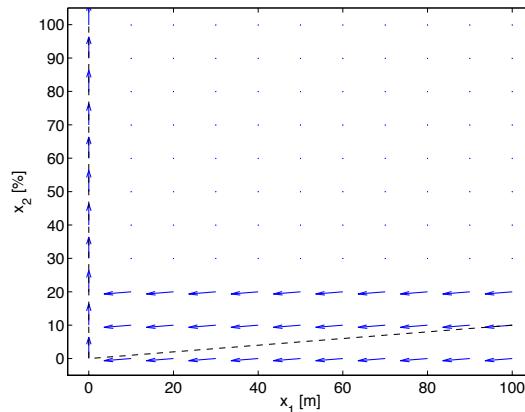


Fig. 5.15: The *Guarantee Power Supply* Action

For *Do Other Task*, \mathcal{T}_2 , we have that

$$S_2 = \emptyset \quad (5.49)$$

$$R_2 = \mathbb{R}^2 \quad (5.50)$$

$$F_2 = \emptyset \quad (5.51)$$

$$f_2(x) = \begin{pmatrix} x_1 + (50 - x_1)/50 \\ x_2 - 0.1 \end{pmatrix} \quad (5.52)$$

that is, when running, the robot moves towards $x_1 = 50$ and does some important task, while the battery level keeps on decreasing. A phase portrait of $f_2(x) - x$ is shown in Figure 5.15.

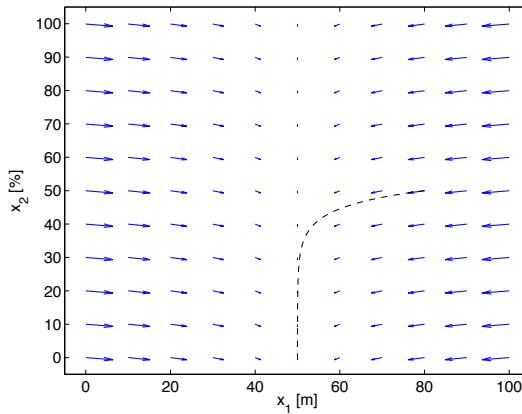


Fig. 5.16: The Do Other Task Action

Given \mathcal{T}_1 and \mathcal{T}_2 , the composition $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$ is created to improve the safety of \mathcal{T}_2 , as described below.

Informally, we can look at the phase portrait in Figure 5.17 to get a feeling for what is going on. The obstacle to be avoided is the Empty Battery state $O = \{x : x_2 = 0\}$, and \mathcal{T}_0 makes sure that this state is never reached, since the *Guarantee Power Supply* action starts executing as soon as *Do Other Task* brings the battery level below 20%. The remaining battery level is also enough for the robot to move back to the recharging station, given that the robot position is limited by the reachable space, i.e., $x_{1k} \in [0, 100]$.

Formally, we state the following Lemma

Lemma 5.8. *Let the obstacle region be $O = \{x : x_2 = 0\}$ and the initialization region be $I = \{x : x_1 \in [0, 100], x_2 \geq 15\}$.*

Furthermore, let \mathcal{T}_1 be given by (5.44)-(5.48) and \mathcal{T}_2 be an arbitrary BT satisfying $\max_x \|x - f_2(x)\| < d = 5$, then $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$ is safe with respect to I and O , i.e. if $x(0) \in I$, then $x(t) \notin O$, for all $t > 0$.

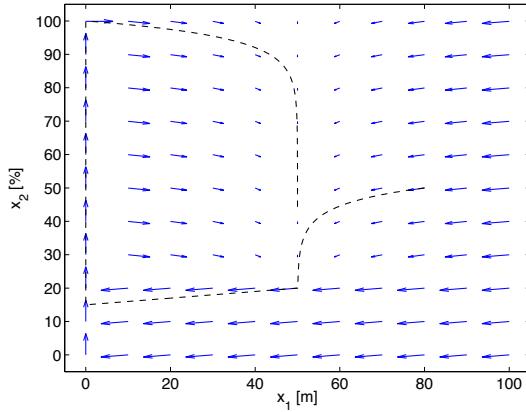


Fig. 5.17: Phase portrait of $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$. Note that \mathcal{T}_1 guarantees that the combination does not run out of battery. The dashed line is a simulated execution, starting at (80,50).

Proof. First we see that \mathcal{T}_1 is safe with respect to O and I . Then we notice that \mathcal{T}_1 is safeguarding with margin $d = 10$ for the reachable set $X = \{x : x_1 \in [0, 100], x_2 \in [0, 100]\}$. Finally we conclude that \mathcal{T}_0 is Safe, according to Lemma 5.5.

Note that if we did not constraint the robot to move in some reachable set $X = \{x : x_1 \in [0, 100], x_2 \in [0, 100]\}$, it would be able to move so far away from the recharging station that the battery would not be sufficient to bring it back again before reaching $x_2 = 0$.

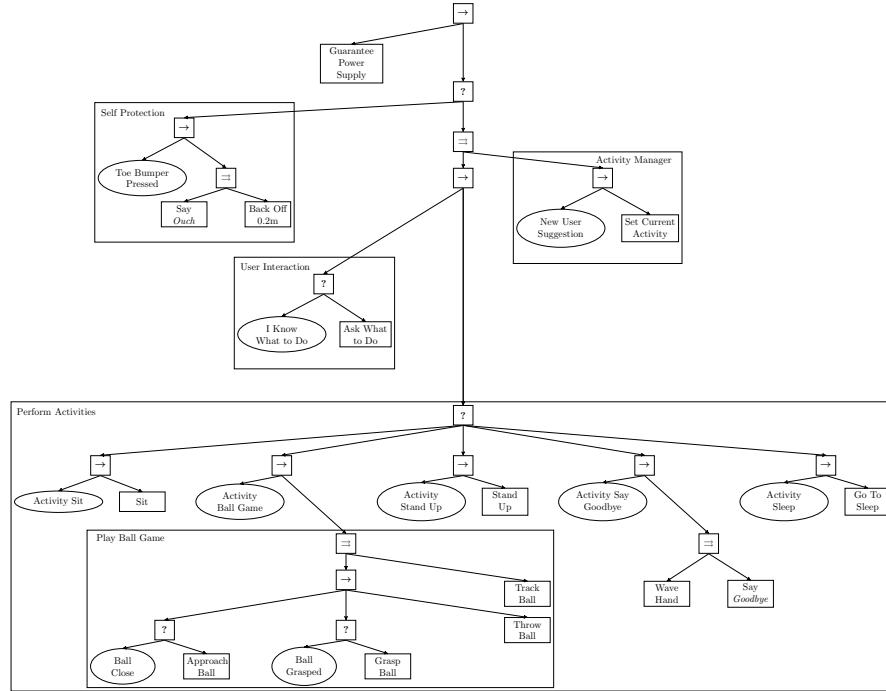


Fig. 5.18: A BT that combines some capabilities of the humanoid robot in an interactive and modular way. Note how atomic actions can easily be replaced by more complex sub-BTs.

5.4.3 A More Complex BT

Below we will use a larger BT to illustrate modularity, as well as the applicability of the proposed analysis tools to more complex problems.

Example 5.6. The BT in Figure 5.18 is designed for controlling a humanoid robot in an interactive capability demo, and includes the BTs of Figures 5.14 and 5.9 as subtrees, as discussed below.

The top left part of the tree includes some exception handling, in terms of battery management, and backing up and complaining in case the toe bumpers are pressed. The top right part of the tree is a Parallel node, listening for new user commands, along with a request for such commands if none are given and an execution of the corresponding activities if a command has been received.

The subtree *Perform Activities* is composed of checking of what activity to do, and execution of the corresponding command. Since the activities are mutually exclusive, we let the Current Activity hold only the latest command and no ambiguities of control commands will occur.

The subtree *Play Ball Game* runs the ball tracker, in parallel with moving closer to the ball, grasping it, and throwing it.

As can be seen, the design is quite modular. A HDS implementation of the same functionality would need an extensive amount of transition arrows going in between the different actions.

We will now apply the analysis tools of the paper to the example, initially assuming that all atomic actions are FTS, as described in Definition 5.5.

Comparing Figures 5.14 and 5.18 we see that they are identical, if we let *Do Other Task* correspond to the whole right part of the larger BT. Thus, according to Lemma 5.8, the complete BT is safe, i.e. it will not run out of batteries, as long as the reachable state space is bounded by 100 distance units from the recharging station and the time steps are small enough so that $\max_x \|x - f_2(x)\| < d = 5$, i.e. the battery does not decrease more than 5% in a single time step.

The design of the right subtree in *Play Ball Game* is made to satisfy Lemma 5.2, with the condition $S_1 = R'_2 \cup S_2$. Let $\mathcal{T}_1 = \text{Fallback}(\text{Ball Close?}, \text{Approach Ball})$, $\mathcal{T}_2 = \text{Fallback}(\text{Ball Grasped?}, \text{Grasp Ball})$, $\mathcal{T}_3 = \text{Throw Ball}$. Note that the use of condition-action pairs makes the success regions explicit. Thus $S_1 = R'_2 \cup S_2$, i.e. Ball Close is designed to describe the Region of Attraction of Grasp Ball, and $S_2 = R'_3 \cup S_3$, i.e. Ball Grasped is designed to describe the Region of Attraction of Throw Ball. Finally, applying Lemma 5.2 twice we conclude that the right part of *Play Ball Game* is FTS with completion time bound $\tau_1 + \tau_2 + \tau_3$, region of attraction $R'_1 \cup R'_2 \cup R'_3$ and success region $S_1 \cap S_2 \cap S_3$.

The Parallel composition at the top of *Play Ball Game* combines *Ball Tracker* which always returns Running, with the subtree discussed above. The Parallel node has $M = 1$, i.e. it only needs the Success of one child to return Success. Thus, it is clear from Definition 5.4 that the whole BT *Play Ball Game* has the same properties regarding FTS as the right subtree.

Finally, we note that *Play Ball Game* fails if the robot is not standing up. Therefore, we improve the robustness of that subtree in a way similar to Example 5.4 in Figure 5.9. Thus we create the composition $\text{Fallback}(\text{Play Ball Game}, \mathcal{T}_5, \mathcal{T}_6)$, with $\mathcal{T}_5 = \text{Sit to Stand}$, $\mathcal{T}_6 = \text{Lie Down to Sit Up}$.

Assuming that that high dimensional dynamics of *Play Ball Game* is somehow captured in the x_1 dimension we can apply an argument similar to Lemma 5.7 to conclude that the combined BT is indeed also FTS with completion time bound $\tau_1 + \tau_2 + \tau_3 + \tau_5 + \tau_6$, region of attraction $R'_1 \cup R'_2 \cup R'_3 \cup R'_5 \cup R'_6$ and success region $S_1 \cap S_2 \cap S_3$.

The rest of the BT concerns user interaction and is thus not suitable for doing performance analysis.

Note that the assumption on all atomic actions being FTS is fairly strong. For example, the humanoid's grasping capabilities are somewhat unreliable. A deterministic analysis such as this one is still useful for making good design choices, but in order to capture the stochastic properties of a BT, we need the tools of Chapter 9.

But first we will use the tools developed in this chapter to formally investigate how BTs relate to other control architectures.

Chapter 6

Formal Analysis of How Behavior Trees Generalize Earlier Ideas

In this chapter, we will formalize the arguments of Chapter 2, using the tools developed in Chapter 5. In particular, we prove that BTs generalize Decision Trees (6.1), the Subsumptions Architecture (6.2), Sequential Behavior Compositions (6.3) and the Teleo-Reactive Approach (6.4). Some of the results of this chapter were previously published in the journal paper [13].

6.1 How BTs Generalize Decision Trees

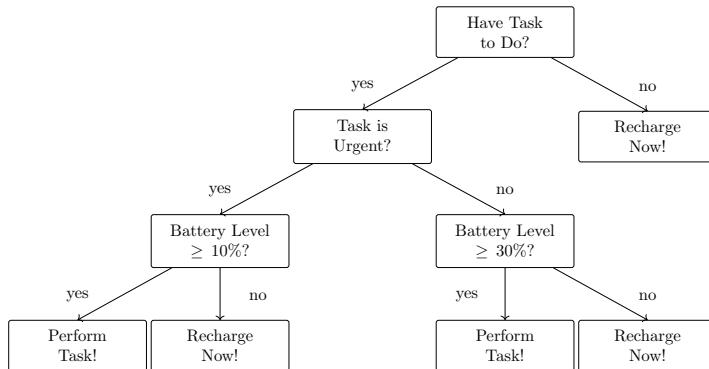


Fig. 6.1: The Decision Tree of a robot control system. The decisions are interior nodes, and the actions are leaves.

Consider the Decision Tree of Figure 6.1, the robot has to decide whether to perform a given task or recharge its batteries. This decision is taken based upon the urgency of the task, and the current battery level. The following Lemma shows how to create an equivalent BT from a given Decision Tree.

Lemma 6.1. *Given a Decision Tree as follows*

$$DT_i = \begin{cases} DT_{i1} & \text{if predicate } P_i \text{ is true} \\ DT_{i2} & \text{if predicate } P_i \text{ is false} \end{cases} \quad (6.1)$$

where DT_{i1} , DT_{i2} are either atomic actions, or subtrees with identical structure, we can create an equivalent BT by setting

$$\mathcal{T}_i = \text{Fallback}(\text{Sequence}(P_i, \mathcal{T}_{i1}), \mathcal{T}_{i2}) \quad (6.2)$$

for non-atomic actions, $\mathcal{T}_i = DT_i$ for atomic actions and requiring all actions to return Running all the time.

The original Decision Tree and the new BT are equivalent in the sense that the same values for P_i will always lead to the same atomic action being executed. The lemma is illustrated in Figure 6.2.

Proof. The BT equivalent of the Decision Tree is given by

$$\mathcal{T}_i = \text{Fallback}(\text{Sequence}(P_i, \mathcal{T}_{i1}), \mathcal{T}_{i2})$$

For the atomic actions always returning Running we have $r_i = R$, for the actions being predicates we have that $r_i = P_i$. This, together with Definitions 5.2-5.3 gives that

$$f_i(x) = \begin{cases} f_{i1} & \text{if predicate } P_i \text{ is true} \\ f_{i2} & \text{if predicate } P_i \text{ is false} \end{cases} \quad (6.3)$$

which is equivalent to (6.1).

Informally, first we note that by requiring all actions to return Running, we basically disable the feedback functionality that is built into the BT. Instead whatever action that is ticked will be the one that executes, just as the Decision Tree. Second the result is a direct consequence of the fact that the predicates of the Decision Trees are essentially ‘If ... then ... else ...’ statements, that can be captured by BTs as shown in Figure 6.2.

Note that this observation opens possibilities of using the extensive literature on learning Decision Trees from human operators, see e.g. [65], to create BTs. These learned BTs can then be extended with safety or robustness features, as described in Section 5.2.

We finish this section with an example of how BTs generalize Decision Trees. Consider the Decision Tree in Figure 6.1. Applying Lemma 6.1 we get the equivalent BT of Figure 6.3. However the direct mapping does not always take full advan-

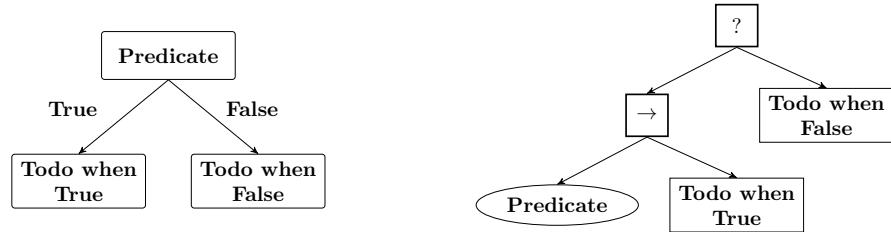


Fig. 6.2: The basic building blocks of Decision Trees are ‘If ... then ... else ...’ statements (left), and those can be created in BTs as illustrated above (right).

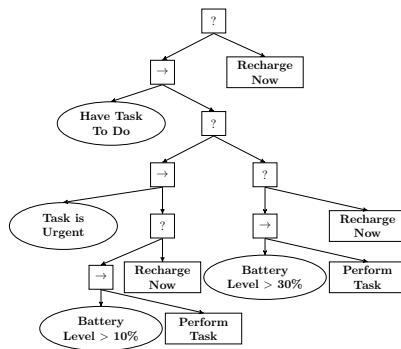


Fig. 6.3: A BT that is equivalent to the Decision Tree in Figure 6.1. A compact version of the same tree can be found in Figure 6.4.

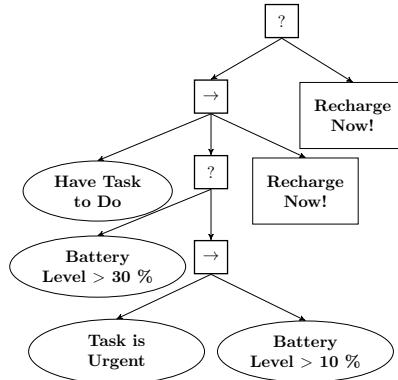


Fig. 6.4: A compact formulation of the BT in Figure 6.3.

tage of the features of BTs. Thus a more compact, and still equivalent, BT can be found in Figure 6.4, where again, we assume that all actions always return *Running*.

6.2 How BTs Generalize the Subsumption Architecture

In this section, we will see how the Subsumption Architecture, proposed by Brooks [6], can be realized using a Fallback composition. The basic idea in [6] was to have a number of controllers set up in parallel and each controller was allowed to output both actuator commands, and a binary value, signaling if it wanted to control the robot or not. The controllers were then ordered according to some priority, and the highest priority controller, out of the ones signaling for action, was allowed to control the robot. Thus, a higher level controller was able to *subsume* the actions of a lower level one.

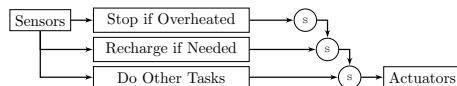


Fig. 6.5: The Subsumption Architecture. A higher level behavior can subsume (or suppress) a lower level one.

An example of a Subsumption architecture can be found in Figure 6.5. Here, the basic level controller *Do Other Tasks* is assumed to be controlling the robot for most of the time. However, when the battery level is low enough, the *Recharge if Needed* controller will signal that it needs to command the robot, subsume the lower level controller, and guide the robot towards the recharging station. Similarly, if there is risk for overheating, the top level controller *Stop if Overheated* will subsume both of the lower level ones, and stop the robot until it has cooled down.

Lemma 6.2. *Given a Subsumption architecture, we can create an equivalent BT by arranging the controllers as actions under a Fallback composition, in order from higher to lower priority. Furthermore, we let the return status of the actions be Failure if they do not need to execute, and Running if they do. They never return Success. Formally, a subsumption architecture composition $S_i(x) = \text{Sub}(S_{i1}(x), S_{i2}(x))$ can be defined by*

$$S_i(x) = \begin{cases} S_{i1}(x) & \text{if } S_{i1} \text{ needs to execute} \\ S_{i2}(x) & \text{else} \end{cases} \quad (6.4)$$

Then we write an equivalent BT as follows

$$\mathcal{T}_i = \text{Fallback}(\mathcal{T}_{i1}, \mathcal{T}_{i2}) \quad (6.5)$$

where \mathcal{T}_{ij} is defined by $f_{ij}(x) = S_{ij}(x)$ and

$$r_{ij}(x) = \begin{cases} \mathcal{R} & \text{if } S_{ij} \text{ needs to execute} \\ \mathcal{F} & \text{else.} \end{cases} \quad (6.6)$$

Proof. By the above arrangement, and Definition 5.3 we have that

$$f_i(x) = \begin{cases} f_{i1}(x) & \text{if } S_{i1} \text{ needs to execute} \\ f_{i2}(x) & \text{else,} \end{cases} \quad (6.7)$$

which is equivalent to (6.4) above. In other words, actions will be checked in order of priority, until one that returns Running is found.

A BT version of the example in Figure 6.5 can be found in Figure 6.6. Table 6.1 illustrates how the two control structures are equivalent, listing all the 2^3 possible return status combinations. Note that no action is executed if all actions return Failure.

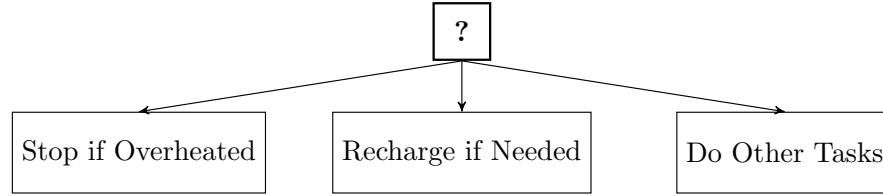


Fig. 6.6: A BT version of the Subsumption example in Figure 6.5.

Stop if overheated	Recharge if Needed	Do Other Tasks	Action Executed
Running	Running	Running	Stop ...
Running	Running	Failure	Stop ...
Running	Failure	Running	Stop ...
Running	Failure	Failure	Stop ...
Failure	Running	Running	Recharge ...
Failure	Running	Failure	Recharge ...
Failure	Failure	Running	Do other ...
Failure	Failure	Failure	-

Table 6.1: Possible outcomes of Subsumption-BT example.

6.3 How BTs Generalize Sequential Behavior Compositions

In this section, we will see how the Fallback composition, and Lemma 5.3, can also be used to implement the Sequential Behavior Compositions proposed in [8].

The basic idea proposed by [8] is to extend the region of attraction by using a family of controllers, where the asymptotically stable equilibrium of each controller was either the goal state, or inside the region of attraction of another controller, positioned earlier in the sequence.

We will now describe the construction of [8] in some detail, and then see how this concept is captured in the BT framework. Given a family of controllers $U = \{\Phi_i\}$, we say that Φ_i prepares Φ_j if the goal $G(\Phi_i)$ is inside the domain $D(\Phi_j)$. Assume the overall goal is located at $G(\Phi_1)$. A set of execution regions $C(\Phi_i)$ for each controller was then calculated according to the following scheme:

1. Let a Queue contain Φ_1 . Let $C(\Phi_1) = D(\Phi_1)$, $N = 1$, $D_1 = D(\Phi_1)$.
2. Remove the first element of the queue and append all controllers that prepare it to the back of the queue.
3. Remove all elements in the queue that already has a defined $C(\Phi_i)$.
4. Let Φ_j be the first element in the queue. Let $C(\Phi_j) = D(\Phi_j) \setminus D_N$, $D_{N+1} = D_N \cup D(\Phi_j)$ and $N \leftarrow N + 1$.
5. Repeat steps 2, 3 and 4 until the queue is empty.

The combined controller is then executed by finding j such that $x \in C(\Phi_j)$ and then invoking controller Φ_j .

Looking at the design of the Fallback operator in BTs, it turns out that it does exactly the job of the Burridge algorithm above, as long as the subtrees of the Fallback are ordered in the same fashion as the queue above. We formalize this in Lemma 6.3 below.

Lemma 6.3. *Given a set of controllers $U = \{\Phi_i\}$ we define the corresponding regions $S_i = G(\Phi_i)$, $R'_i = D(\Phi_i)$, $F_i = \text{Complement}(D(\Phi_i))$, and consider the controllers as atomic BTs, $\mathcal{T}_i = \Phi_i$. Assume S_1 is the overall goal region. Iteratively create a larger BT \mathcal{T}_L as follows*

1. Let $\mathcal{T}_L = \mathcal{T}_1$.
2. Find a BT $\mathcal{T}_* \in U$ such that $S_* \subset R'_L$
3. Let $\mathcal{T}_L \leftarrow \text{Fallback}(\mathcal{T}_L, \mathcal{T}_*)$
4. Let $U \leftarrow U \setminus \mathcal{T}_*$
5. Repeat steps 2, 3 and 4 until U is empty.

If all \mathcal{T}_i are FTS, then so is \mathcal{T}_L .

Proof. The statement is a direct consequence of iteratively applying Lemma 5.3.

Thus, we see that BTs generalize the Sequential Behavior Compositions of [8], with the execution region computations and controller switching replaced by the Fallback composition, as long as the ordering is given by Lemma 6.3 above.

6.4 How BTs Generalize the Teleo-Reactive approach

In this section, we use the following Lemma to show how to create a BT with the same execution as a given Teleo-Reactive program. The lemma is illustrated by Example 6.1 and Figure 6.7.

Lemma 6.4 (Teleo-Reactive BT analogy). *Given a TR in terms of conditions c_i and actions a_i , an equivalent BT can be constructed as follows*

$$\mathcal{T}_{TR} = \text{Fallback}(\text{Sequence}(c_1, a_1), \dots, \text{Sequence}(c_m, a_m)), \quad (6.8)$$

where we convert the True/False of the conditions to Success/Failure, and let the actions only return Running.

Proof. It is straightforward to see that the BT above executes the exact same a_i as the original TR would have, depending on the values of the conditions c_i , i.e. it finds the first condition c_i that returns Success, and executes the corresponding a_i .

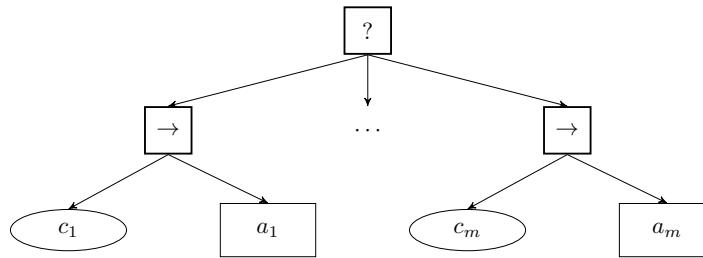


Fig. 6.7: The BT that is analogous to a given TR.

We will now illustrate the lemma with an example from Nilssons original paper [51].

Example 6.1. The Teleo-Reactive program *Goto(loc)* is described as follows, with conditions on the left and corresponding actions to the right:

$$\text{Equal(pos,loc)} \rightarrow \text{Idle} \quad (6.9)$$

$$\text{Heading Towards (loc)} \rightarrow \text{Go Forwards} \quad (6.10)$$

$$(\text{else}) \rightarrow \text{Rotate} \quad (6.11)$$

where *pos* is the current robot position and *loc* is the current destination.

Executing this Teleo-Reactive program, we get the following behavior. If the robot is at the destination it does nothing. If it is heading the right way it moves forward, and else it rotates on the spot. In a perfect world without obstacles, this

will get the robot to the goal, just as predicted in Lemma 6.5. Applying Lemma 6.4, the Teleo-Reactive program Goto is translated to a BT in Figure 6.8.

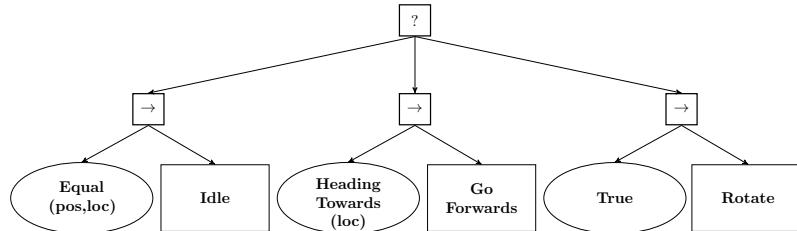


Fig. 6.8: The BT version of the Teleo-Reactive program Goto.

The example continues in [51] with a higher level recursive Teleo-Reactive program, called *Amble(loc)*, designed to add a basic obstacle avoidance behavior

$$\text{Equal}(\text{pos}, \text{loc}) \rightarrow \text{Idle} \quad (6.12)$$

$$\text{Clear Path}(\text{pos}, \text{loc}) \rightarrow \text{GoTo(loc)} \quad (6.13)$$

$$(\text{else}) \rightarrow \text{Amble}(\text{new point}(\text{pos}, \text{loc})) \quad (6.14)$$

where *new point* picks a new random point in the vicinity of *pos* and *loc*.

Again, if the robot is at the destination it does nothing. If the path to goal is clear it executes the Teleo-Reactive program Goto. Else it picks a new point relative to its current position and destination (loc) and recursively executes a new copy of Amble with that destination. Applying Lemma 6.4, the Amble TR is translated to a BT in Figure 6.9.

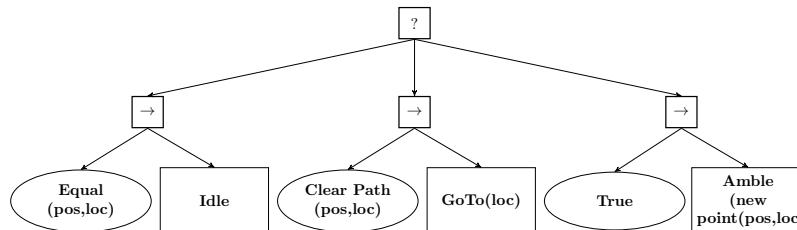


Fig. 6.9: The BT version of the TR Amble.

6.4.1 Universal Teleo-Reactive programs and FTS BTs

Using the functional form of BTs introduced in 5.1 we can show that Lemma 5.3 is a richer version of Lemma 6.5 below, and also fix one of its assumptions. Lemma 5.3 includes execution time, but more importantly builds on a finite difference equation system model over a continuous state space. Thus control theory concepts can be used to include phenomena such as imperfect sensing and actuation into the analysis, that was removed in the strong assumptions of Lemma 6.5. Thus, the BT analogy provides a powerful tool for analyzing Teleo-Reactive designs.

Lemma 6.5 (Nilsson 1994). *If a Teleo-Reactive program is Universal, and there are no sensing and execution errors, then the execution of the program will lead to the satisfaction of c_1 .*

Proof. In [51] it is stated that it is easy to see that this is the case.

The idea of the proof is indeed straight forward, but as we will see when we compare it to the BT results in Section 6.4.1 below, the proof is incomplete.

In Lemma 5.3, S_i, R_i, F_i correspond to Success, Running and Failure regions and R' denotes the region of attraction.

Lemma 5.3 shows under what conditions we can guarantee that the Success region S_0 is reached in finite time. If we for illustrative purposes assume that the regions of attraction are identical to the running regions $R_i = R'_i$, the lemma states that as long as the system starts in $R'_0 = R'_1 \cup R'_2$ it will reach $S_0 = S_1$ in less than $\tau_0 = \tau_1 + \tau_2$ time units. The condition analogous to the *regression property* is that $S_2 \subset R'_1$, i.e. that the Success region of the second BT is a subset of the region of attraction R'_1 of the first BT. The regions of attraction, R'_1 and R'_2 are very important, but there is no corresponding concept in Lemma 6.5. In fact, we can construct a counter example showing that Lemma 6.5 does not hold.

Example 6.2 (Counter Example). Assume that a Teleo-Reactive program is Universal in the sense described above. Thus, the execution of action a_i eventually leads to the satisfaction of c_j where $j < i$ for all $i \neq 1$. However, assume it is also the case that the execution of a_i , on its way towards satisfying c_j actually leads to a violation of c_i . This would lead to the first true condition being some c_m , with $m > i$ and the execution of the corresponding action a_m . Thus, the chain of decreasing condition numbers is broken, and the goal condition a_1 might never be reached.

The fix is however quite straightforward, and amounts to using the following definition with a stronger assumption.

Definition 6.1 (Stronger Regression property). For each $c_i, i > 1$ there is $c_j, j < i$ such that the execution of action a_i leads to the satisfaction of c_j , without ever violating c_i .

Chapter 7

Behavior Trees and Automated Planning

In this chapter, we describe how automatic planning can be used to create BTs, exploiting ideas from [17, 75, 74, 10]. First, in Section 7.1, we present an extension of the Backchaining design principle, introduced in Section 3.5, including a robotics example. Then we present an alternative approach using A Behavior Language (ABL), in Section 7.2, including a game example.

In classical planning research, the world is often assumed to be static and known, with all changes occurring as a result of the actions executed by one controlled agent [25]. Therefore, most approaches return a static plan, i.e. a sequence of actions that brings the system from the initial state to the goal state, with a corresponding execution handled by a classical FSM.

However, many agents, both real and virtual, act in an uncertain world populated by other agents, each with their own set of goals and objectives. Thus, the effect of an action can be unexpected, diverging from the planned state trajectory, making the next planned action infeasible. A common way of handling this problem is to re-plan from scratch on a regular basis, which can be expensive both in terms of time and computational load. To address these problems, the following two open challenges were identified within the planning community [24]:

- “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 [25] describes the need for an actor that “reacts to events and extends, updates, and repairs its plan on the basis of its perception”.

Combining planning with BTs is one way of addressing these challenges. The reactivity of BTs enables the agent to re-execute previous subplans without having to replan at all, and the modularity enables extending the plan recursively, without having to re-plan the whole task. Thus, using BTs as the control architecture in an automated planning algorithm addresses the above challenges by enabling a reason-

ing process that is both hierarchical and modular in its deliberation, and can monitor, update, and extend its plans while acting.

In practice, and as will be seen in the examples below, using BTs enables reactivity, in the sense that if an object slips out of a robot's gripper, the robot will automatically stop and pick it up again without the need to replan or change the BT, see Fig. 7.16. Using BTs also enables iterative plan refinement, in the sense that if an object is moved to block the path, the original BT can be extended to include a removal of the blocking obstacle. Then, if the obstacle is removed by an external actor, the robot reactively skips the obstacle removal, and goes on to pick up the main object without having to change the BT, see Fig. 7.23.

7.1 The Planning and Acting (PA-BT) approach

In this section, we describe an extension of the Backchaining approach, called *Planning and Acting using Behavior Trees* (PA-BT).

PA-BT was inspired by the Hybrid Backward-Forward (HBF) algorithm, a task planner for dealing with infinite state spaces [22]. 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 PA-BT framework combines the planning capability in an infinite state space from HBF with the advantages of BTs compared to FSMs in terms of *reactivity* and *modularity*. Looking back at the example above, the *reactivity* of BTs enables the robot to pick up a dropped object without having to replan at all. The *modularity* enables extending the plan by adding actions for handling the blocking sphere, without having to replan the whole task. Finally, when the sphere moves away, once again the *reactivity* enables the correct execution without changing the plan. Thus, PA-BT is indeed hierarchical and modular in its deliberation, and it does monitor, update and extend its plans while acting, addressing the needs described in [25, 24]. The interleaved plan-and-act process of PA-BT is similar to the one of *Hierarchical Planning in the Now* (HPN) [33] with the addition of improved reactivity, safety and fault-tolerance.

The core concept in BT Backchaining was to replace a condition by a small BT achieving that condition, on the form of a PPA BT (see Section 3.5), as shown in Figure 7.1.

To get familiar with PA-BT, we look at a simple planning example. The planning algorithm iteratively creates the BTs in Figure 7.2 with the final BT used in the example in Figure 7.2e. The setup is shown in Figure 7.3. We can see how each of the BTs in the figure is the result of applying the PPA expansion to a condition in the previous BT. However, a number of details needs to be taken care of, as explained in sections below.

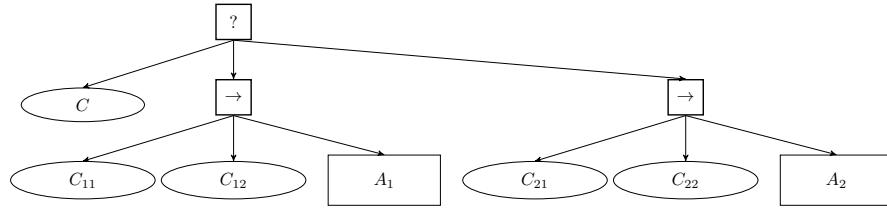


Fig. 7.1: Copy of Figure 3.11. The general format of a PPA BT. The Postcondition C can be achieved by either one of actions A_1 or A_2 , which have Preconditions C_{1i} and C_{2i} respectively.)

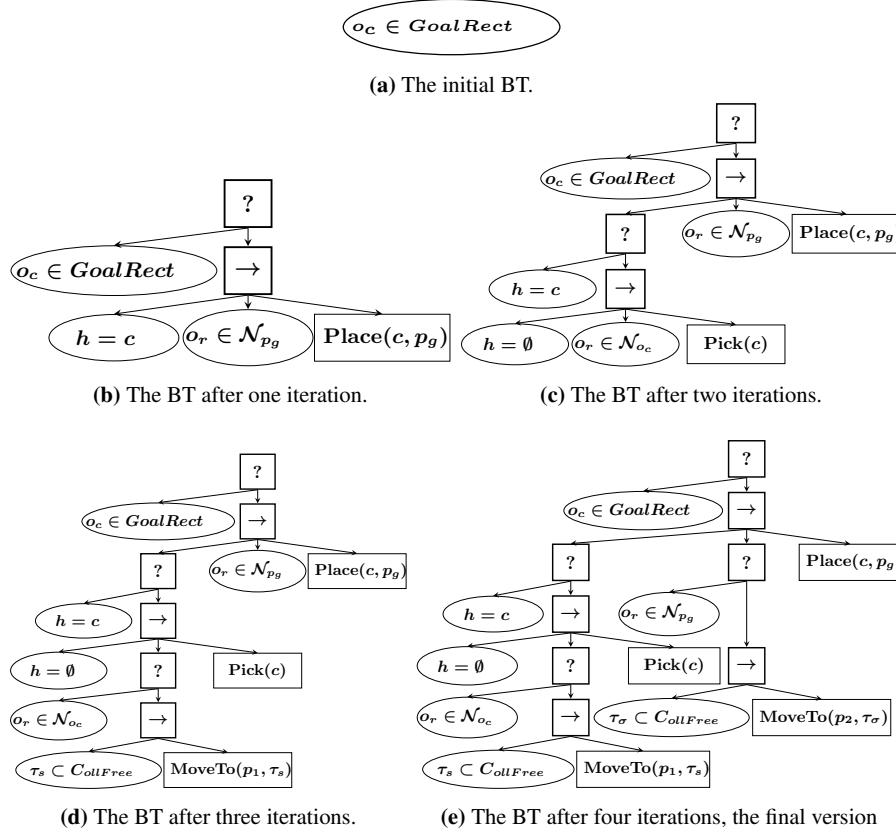


Fig. 7.2: BT updates during the execution.

Example 7.1. The robot in Figure 7.3 is given the task to move the green cube into the rectangle marked GOAL (the red sphere is handled in Example 7.2 below, in this initial example it is ignored). The BT in Figure 7.2e is executed, and in each time step the root of the BT is ticked. The root is a Fallback node, which ticks is

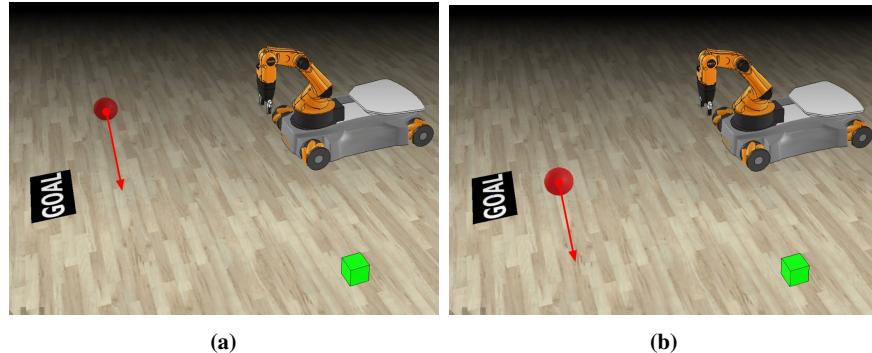


Fig. 7.3: A simple example scenario where the goal is to place the green cube C onto the goal region G . The fact that the sphere S is suddenly moved (red arrow) by an external agent to block the path must be handled. In (a) the nominal plan is to $\text{MoveTo}(c) \rightarrow \text{Pick}(c) \rightarrow \text{MoveTo}(g) \rightarrow \text{Drop}()$ when the sphere suddenly moves to block the path. In (b), after refining the plan, the extended plan is to $\text{MoveTo}(s) \rightarrow \text{Push}(s) \rightarrow \text{MoveTo}(c) \rightarrow \text{Pick}(c) \rightarrow \text{MoveTo}(g) \rightarrow \text{Drop}()$ when the sphere is again suddenly moved by another agent, before being pushed. Thus our agent must smoothly revert to the original set of actions. PA-BT does this without re-planning. Note that when S appears, $\tau_\sigma \subset C_{\text{collFree}}$ returns Failure and the BT in Figure 7.2e is expanded further, to push it out of the way.

first child, the condition $o_c \in \text{GoalRect}$ (cube on goal). If the cube is indeed in the rectangle we are done, and the BT returns Success.

If not, the second child, a Sequence node, is ticked. The node ticks its first child, which is a Fallback, which again ticks its first child, the condition $h = c$ (object in hand is cube). If the cube is indeed in the hand, the Condition node returns Success, its parent, the Fallback node returns Success, and its parent, the Sequence node ticks its second child, which is a different Fallback, ticking its first child which is the condition $o_r \in \mathcal{N}_{p_g}$ (robot in the neighborhood of p_g). If the robot is in the neighborhood of the goal, the condition and its parent node (the Fallback) returns Success, followed by the sequence ticking its third child, the action $\text{Place}(c, p_g)$ (place cube in a position p_g on the goal), and we are done.

If $o_r \in \mathcal{N}_{p_g}$ does not hold, the action $\text{MoveTo}(p_g, \tau_g)$ (move to position p_g on the goal region using the trajectory τ_g) is executed, given that the trajectory is free $\tau \subset C_{\text{collFree}}$. Similarly, if the cube is not in the hand, the robot does a $\text{MoveTo}(p_c, \tau_c)$ (move to cube, using the trajectory τ_c) followed by a $\text{Pick}(c)$ after checking that the hand is empty, the robot is not in the neighborhood of c and that the corresponding trajectory is free.

We conclude the example by noting that the BT is ticked every timestep, e.g. every 0.1 second. Thus, when actions return Running (i.e. they are not finished yet) the return status of Running is progressed up the BT and the corresponding action is allowed to control the robot. However, if e.g., the cube slips out of the gripper, the condition $h = c$ instantly returns Failure, and the robot starts checking if it is in the neighborhood of the cube or if it has to move before picking it up again.

We are now ready to study PA-BT in detail. The approach is described in Algorithms 5 (finding what condition to replace with a PPA) and 6 (creating the PPA and adding it to the BT). First we will give an overview of the algorithms and see how they are applied to the robot in Figure 7.3, to iteratively create the BTs of Figure 7.2. We will then discuss the key steps in more detail.

Algorithm 5: Main Loop, finding conditions to expand and resolving conflicts

```

1  $\mathcal{T} \leftarrow \emptyset$ 
2 for  $c$  in  $\mathcal{C}_{goal}$  do
3    $\mathcal{T} \leftarrow \text{SequenceNode}(\mathcal{T}, c)$ 
4 while True do
5    $T \leftarrow \text{RefineActions}(\mathcal{T})$ 
6   do
7      $r \leftarrow \text{Tick}(T)$ 
8   while  $r \neq \text{Failure}$ 
9    $c_f \leftarrow \text{GetConditionToExpand}(\mathcal{T})$ 
10   $\mathcal{T}, \mathcal{T}_{new\_subtree} \leftarrow \text{ExpandTree}(\mathcal{T}, c_f)$ 
11  while  $\text{Conflict}(\mathcal{T})$  do
12     $\mathcal{T} \leftarrow \text{IncreasePriority}(\mathcal{T}_{new\_subtree})$ 

```

Algorithm 6: Behavior Tree Expansion, Creating the PPA

```

1 Function  $\text{ExpandTree}(\mathcal{T}, c_f)$ 
2    $A_T \leftarrow \text{ GetAllActTemplatesFor}(c_f)$ 
3    $\mathcal{T}_{fall} \leftarrow c_f$ 
4   for  $a$  in  $A_T$  do
5      $\mathcal{T}_{seq} \leftarrow \emptyset$ 
6     for  $c_a$  in  $a.con$  do
7        $\mathcal{T}_{seq} \leftarrow \text{SequenceNode}(\mathcal{T}_{seq}, c_a)$ 
8      $\mathcal{T}_{seq} \leftarrow \text{SequenceNode}(\mathcal{T}_{seq}, a)$ 
9      $\mathcal{T}_{fall} \leftarrow \text{FallbackNodeWithMemory}(\mathcal{T}_{fall}, \mathcal{T}_{seq})$ 
10     $\mathcal{T} \leftarrow \text{Substitute}(\mathcal{T}, c_f, \mathcal{T}_{fall})$ 
11    return  $\mathcal{T}, \mathcal{T}_{fall}$ 

```

Remark 7.1. Note that the conditions of an action template can contain a disjunction of propositions. This can be encoded by a Fallback composition of the corresponding Condition nodes.

Algorithm 7: Get Condition to Expand

```

1 Function GetConditionToExpand( $\mathcal{T}$ )
2   for  $c_{next}$  in GetConditionsBFS() do
3     if  $c_{next}.status = \text{Failure}$  and  $c_{next} \notin ExpandedNodes$  then
4        $ExpandedNodes.push\_back(c_{next})$  return  $c_{next}$ 
5   return None

```

7.1.1 Algorithm Overview

Running Algorithm 5 we have the set of goal constraint $\mathcal{C}_{goal} = \{o_c \in \{\text{GoalRect}\}\}$, thus the initial BT is composed of a single condition $\mathcal{T} = (o_c \in \{\text{GoalRect}\})$, as shown in Figure 7.2a. The first iteration of the loop starting on Line 4 of Algorithm 5 now produces the next BT shown in Figure 7.2b, the second iteration produces the BT in Figure 7.2c and so on until the final BT in Figure 7.2e.

In detail, at the initial state, running \mathcal{T} on Line 7 returns a Failure, since the cube is not in the goal area. Trivially, the *GetConditionToExpand* returns $c_f = (o_c \in \{\text{GoalRect}\})$, and a call to *ExpandTree* (Algorithm 6) is made on Line 10. On Line 2 of Algorithm 6 we get all action templates that satisfy c_f i.e. $A_T = Place$. Then on Line 7 and 8 a Sequence node \mathcal{T}_{seq} is created of the conditions of *Place* (the hand holding the cube, $h = c$, and the robot being near the goal area, $o_r \in \mathcal{N}_{pg}$) and *Place* itself. On Line 9 a Fallback node \mathcal{T}_{seq} is created of c_f and the sequence above. Finally, a BT is returned where this new sub-BT is replacing c_f . The resulting BT is shown in Figure 7.2b.

Note that Algorithm 6 describes *the core principle* of the PA-BT approach. A *condition is replaced by the corresponding PPA, a check if the condition is met, and an action to meet it. The action is only executed if needed*. If there are several such actions, these are added in Fallbacks with memory. Finally, the action is preceded by conditions checking its own preconditions. If needed, these conditions will be expanded in the same way in the next iteration.

Running the next iteration of Algorithm 5, a similar expansion of the condition $h = c$ transforms the BT in Figure 7.2b to the BT in Fig. 7.2c. Then, an expansion of the condition $o_r \in \mathcal{N}_{oc}$ transforms the BT in Figure 7.2c to the BT in Figure 7.2d. Finally, an expansion of the condition $o_r \in \mathcal{N}_{pg}$ transforms the BT in Figure 7.2d to the BT in Figure 7.2e, and this BT is able to solve the problem shown in Figure 7.3, until the red sphere shows up. Then additional iterations are needed.

7.1.2 The Algorithm Steps in Detail

Refine Actions (Algorithm 5 Line 5)

PA-BT is based on the definition of the *action templates*, which contains the descriptive model of an action. An action template is characterized by conditions *con* (sometimes called preconditions) and effects *eff* (sometimes called postconditions) that are both constraints on the world (e.g. door open, robot in position). An action template is mapped online into an *action primitive*, which contains the operational model of an action and is executable. Figure 7.4 shows an example of an action template and its corresponding action refinement.

To plan in infinite state space, PA-BT relies on a so-called Reachability Graph (RG) provided by the HBF algorithm, see [22] for details. The RG provides efficient sampling for the actions in the BT, allowing us to map the descriptive model of an action into its operational model.

$\text{Pick}(i)$ $\text{con} : o_r \in \mathcal{N}_{o_i}$ $h = \emptyset$ $\text{eff} : h = i$	$\text{Pick}(cube)$ $\text{con} : o_r \in \mathcal{N}_{o_{cube}}$ $h = \emptyset$ $\text{eff} : h = cube$
(a) Action Template for picking a generic object denoted i.	(b) Action primitive created from the Template in (a), where the object is given as $i = cube$.

Fig. 7.4: Action Template for Pick and its corresponding Action primitive. o_r is the robot's position, \mathcal{N}_{o_i} is a set that defines a neighborhood of the object o_i , h is the object currently in the robot's hand. The conditions are that the robot is in the neighborhood of the object, and that the robot hand is empty. The effect is that the object is in the robot hand.

Get Condition To Expand and Expand Tree (Algorithm 5 Lines 9 and 10)

If the BT returns Failure, Line 9 of Algorithm 5 invokes Algorithm 7, which finds the condition to expand by searching through the conditions returning Failure using a Breadth First Search (BFS). If no such condition is found (Algorithm 7 Line 5) that means that an action returned Failure due to an old refinement that is no longer valid. In that case, at the next loop of Algorithm 5 a new refinement is found (Algorithm 5 Line 5), assuming that such a refinement always exists. If Algorithm 7 returns a condition, this will be expanded (Algorithm 5 Line 10), as shown in the example of Figure 7.2. Example 7.3 highlights the BFS expansion. Thus, \mathcal{T} is expanded until it can perform an action (i.e. until \mathcal{T} contains an action template whose condition are satisfied in the current state). In [10] is proved that \mathcal{T} is expanded a finite number of times. If there exists more than one valid action that satisfies a condition, their respective trees (sequence composition of the action and its conditions) are collected in a Fallback composition with memory, which implements the different options the agent has, to satisfy such a condition. If needed, these options will be executed in turn. PA-BT does not investigate which action is the optimal one. As stressed in [25]

the cost of minor mistakes (e.g. non-optimal actions order) is often much lower than the cost of the extensive modeling, information gathering and thorough deliberation needed to achieve optimality.

Conflicts and Increases in Priority (Algorithm 5 Lines 11 and 12)

Similar to any STRIPS-style planner, adding a new action in the plan can cause a *conflict* (i.e. the execution of this new action reverses the effects of a previous action). In PA-BT, this possibility is checked in Algorithm 5 Line 11 by analyzing the conditions of the new action added with the effects of the actions that the subtree executes before executing the new action. If this effects/conditions pair is in conflict, the goal will not be reached. An example of this situation is described in Example 7.2 below.

Again, following the approach used in STRIPS-style planners, we resolve this conflict by finding the correct action order. Exploiting the structure of BTs we can do so by moving the tree composed by the new action and its condition leftward (a BT executes its children from left to right, thus moving a subtree leftward implies executing the new action earlier). If it is the leftmost one, this means that it must be executed before its parent (i.e. it must be placed at the same depth of the parent but to its left). This operation is done in Algorithm 5 Line 12. PA-BT incrementally increases the priority of this subtree in this way, until it finds a feasible tree. In [10] it is proved that, under certain assumptions, a feasible tree always exists .

Get All Action Templates

Let's look again at Example 7.1 above and see how the BT in Figure 7.2e was created using the PA-BT approach. In this example, the action templates are summarized below with conditions and effect:

$\text{MoveTo}(p, \tau)$	$\text{Pick}(i)$	$\text{Place}(i, p)$
$\text{con} : \tau \subset C_{\text{collFree}}$	$\text{con} : o_r \in \mathcal{N}_{o_i}$	$\text{con} : o_r \in \mathcal{N}_p$
	$h = \emptyset$	$h = i$
$\text{eff} : o_r = p$	$\text{eff} : h = i$	$\text{eff} : o_i = p$

where τ is a trajectory, C_{collFree} is the set of all collision free trajectories, o_r is the robot pose, p is a pose in the state space, h is the object currently in the end effector, i is the label of the i -th object in the scene, and \mathcal{N}_x is the set of all the poses near the pose x .

The descriptive model of the action *MoveTo* is parametrized over the destination p and the trajectory τ . It requires that the trajectory is collision free ($\tau \subset C_{\text{collFree}}$). As effect the action *MoveTo* places the robot at p (i.e. $o_r = p$), the descriptive model of the action *Pick* is parametrized over object i . It requires having the end effector free

(i.e. $h = \emptyset$) and the robot to be in a neighborhood \mathcal{N}_{o_i} of the object i . (i.e. $o_r \in \mathcal{N}_{o_i}$). As effect the action *Pick* sets the object in the end effector to i (i.e $h = i$); Finally, the descriptive model of the action *Place* is parametrized over object i and final position p . It requires the robot to hold i (i.e. $h = i$), and the robot to be in the neighborhood of the final position p . As effect the action *Place* places the object i at p (i.e. $o_i = p$).

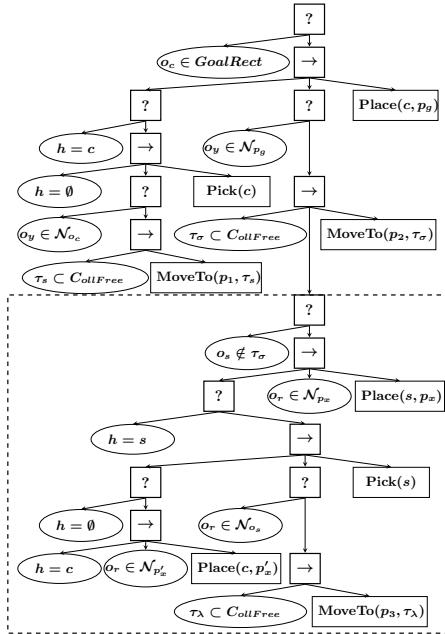
Example 7.2. Here we show a more complex example highlighting two main properties of PA-BT: the livelock freedom and the continual deliberative plan and act cycle. This example is an extension of Example 7.1 where, due to the dynamic environment, the robot needs to replan.

Consider the execution of the final BT in Figure 7.2e of Example 7.1, where the robot is carrying the desired object to the goal location. Suddenly, as in Figure 7.3 (b), an object s obstructs the (only possible) path. Then the condition $\tau \subset C_{ollFree}$ returns Failure and Algorithm 5 expands the tree accordingly (Line 10) as in Figure 7.5a.

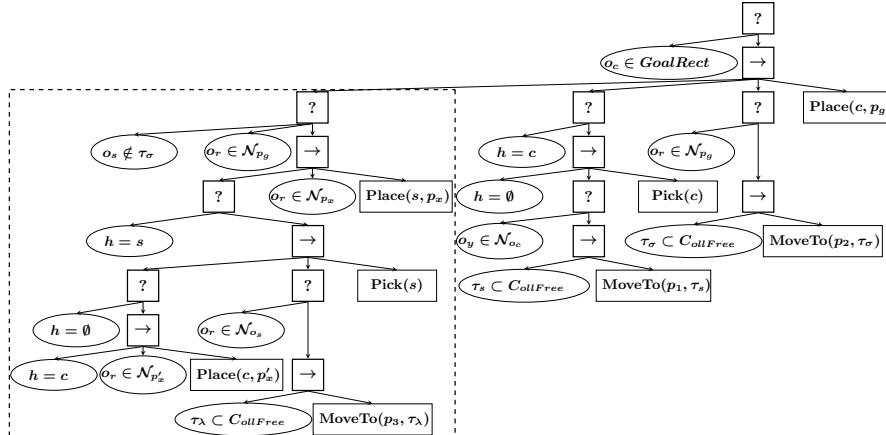
The new subtree has as condition $h = \emptyset$ (no objects in hand) but the effect of the left branch (i.e. the main part in Figure 7.2e) of the BT is $h = c$ (cube in hand) (i.e. the new subtree will be executed if and only if $h = c$ holds). Clearly the expanded tree has a conflict (Algorithm 5 Line 11) and the priority of the new subtree is increased (Line 12), until the expanded tree is in form of Figure 7.5b. Now the BT is free from conflicts as the first subtree has as effect $h = \emptyset$ and the second subtree has a condition $h = \emptyset$. Executing the tree the robot approaches the obstructing object, now the condition $h = \emptyset$ returns Failure and the tree is expanded accordingly, letting the robot drop the current object grasped, satisfying $h = \emptyset$, then it picks up the obstructing object and places it on the side of the path. Now the condition $\tau \subset C_{ollFree}$ finally returns Success. The robot can then again approach the desired object and move to the goal region and place the object in it.

7.1.3 Comments on the Algorithm

It is clear that this type of continual planning and acting exhibits both the important principles of deliberation stressed in [25, 24]: Hierarchically organized deliberation and continual online deliberation. For example, if the robot drops the object, then the condition $h = c$ is no longer satisfied and the BT will execute the corresponding subtree to pick the object, with no need for re-planning. This type of deliberative reactiveness is built into BTs. On the other hand, if during its navigation a new object pops up obstructing the robot's path, the condition $\tau \subset C_{ollFree}$ will no longer return Success and the BT will be expanded accordingly. This case was described in Example 7.2. Moreover, note that PA-BT refines the BT every time it returns Failure. This is to encompass the case where an older refinement is no longer valid. In such cases an action will return Failure. This Failure is propagated up to the root. The function *ExpandTree* (Algorithm 5 Line 10) will return the very same tree (the



(a) Unfeasible expanded tree. The new subtree is highlighted in red.



(b) Expanded Feasible subtree.

Fig. 7.5: Steps to increase the priority of the new subtree added in Example 7.2.

tree needs no extension as there is no failed condition of an action) which gets re-refined in the next loop (Algorithm 5 Line 5). For example, if the robot planned to

place the object in a particular position on the desk but this position was no longer feasible (e.g. another object was placed in that position by an external agent).

7.1.4 Algorithm Execution on Graphs

Here, for illustrative purposes, we show the result of PA-BT when applied to a standard shortest path problem in a graph.

Example 7.3. Consider an agent moving in different states modeled by the graph in Figure 7.6 where the initial state is s_0 and the goal state is s_g . Every arc represents an action that moves an agent from one state to another. The action that moves the agent from a state s_i to a state s_j is denoted by $s_i \rightarrow s_j$. The initial tree, depicted in Figure 7.7a, is defined as a Condition node s_g which returns Success if and only if the robot is at the state s_g in the graph. The current state is s_0 (the initial state). Hence the BT returns a status of *Failure*. Algorithm 5 invokes the BT expansion routine. The state s_g can be reached from the state s_5 , through the action $s_5 \rightarrow s_g$, or from the state s_3 , through the action $s_3 \rightarrow s_g$. The tree is expanded accordingly as depicted in Figure 7.7b. Now executing this tree, it returns a status of *Failure*. Since the current state is neither s_g nor s_3 nor s_5 . Now the tree is expanded in a BFS fashion, finding a subtree for condition s_5 as in Figure 7.7c. The process continues for two more iterations. Note that at iteration 4 (See Figure 7.8b) Algorithm 5 did not expand the condition s_g as it was previously expanded (Algorithm 7 line 3) this avoids infinite loops in the search. The same argument applies for conditions s_4 and s_g in iteration 5 (See Figure 7.8c). The BT at iteration 5 includes the action $s_0 \rightarrow s_1$ whose precondition is satisfied (the current state is s_0). The action is then executed. Performing that action (and moving to s_1), the condition s_1 is satisfied. The BT executes the action $s_1 \rightarrow s_3$ and then $s_3 \rightarrow s_g$, making the agent reach the goal state.

It is clear that the resulting execution is the same as a BFS on the graph would have rendered. Note however that PA-BT is designed for more complex problems than graph search.

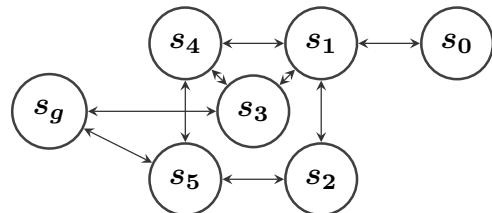


Fig. 7.6: Graph representing the task of Example 7.3.

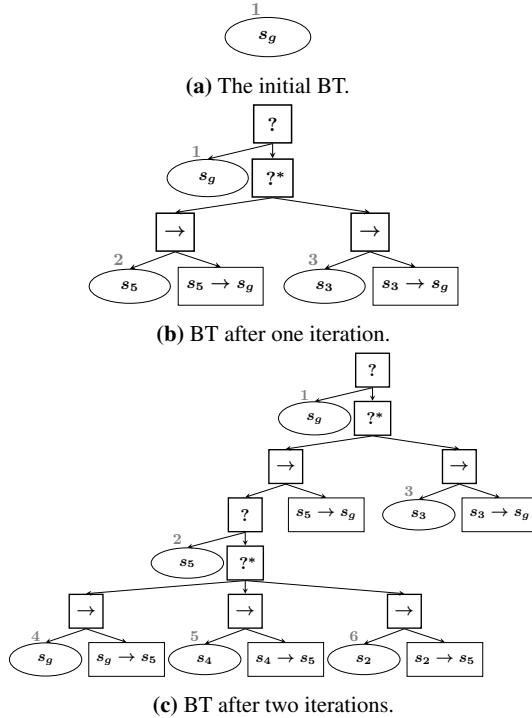


Fig. 7.7: First three BT updates during execution of Example 7.3. The numbers represent the index of the BFS of Algorithm 7. Note that the node labeled with $?^*$ is a Fallback node with memory.

7.1.5 Algorithm Execution on an existing Example

In this section we apply the PA-BT approach in a more complex example adapted from [33].

Example 7.4 (From [33]). Consider a multipurpose robot that is asked to clean the object A and then put it in the storage room as shown in Figure 7.9 (in this first example we ignore the other robots as they are not in [33]). The goal is specified as a conjunction $\text{Clean}(A) \wedge \text{In}(A, \text{storage})$. Using PA-BT, the initial BT is defined as a sequence composition of $\text{Clean}(A)$ with $\text{In}(A, \text{storage})$ as in Figure 7.10a. At execution, the Condition node $\text{Clean}(A)$ returns *Failure* and the tree is expanded accordingly, as in Figure 7.10b. Executing the expanded tree, the Condition node $\text{In}(A, \text{Washer})$ returns *Failure* and the BT is expanded again, as in Figure 7.10c. This iterative process of planning and acting continues until it creates a BT such that the robot is able to reach object C and remove it. After cleaning object A , the approach constructs the tree to satisfy the condition $\text{In}(A, \text{storage})$ as depicted in Figure 7.12. This subtree requires picking object A and then placing it into the storage. However after the BT is expanded to place A into the storage it contains a conflict: in order

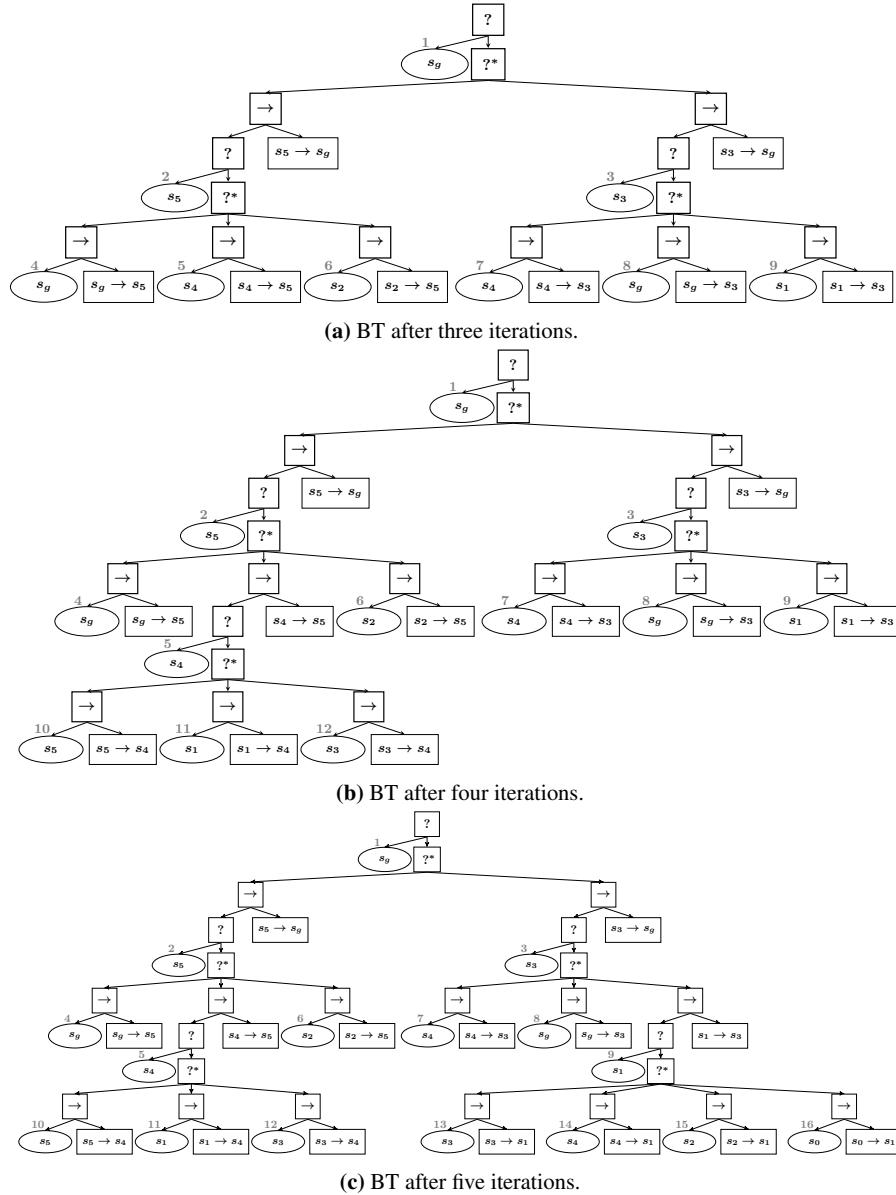


Fig. 7.8: Next BT updates during execution of Example 7.3. The numbers represent the index of the BFS of Algorithm 3. Note that the node labeled with $?^*$ is a Fallback node with memory.

to remove object D the robot needs to grasp it. But to let the ticks reach this tree, the condition $Holding() = a$ needs to return *Success*. Clearly the robot cannot hold

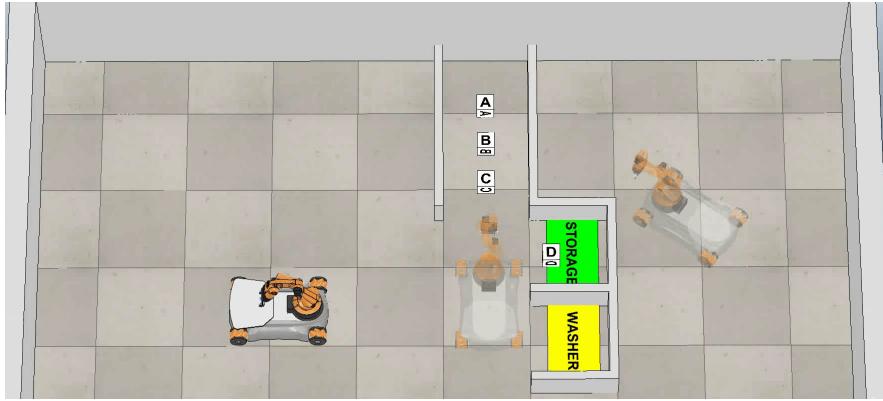


Fig. 7.9: An example scenario from [33], with the addition of two externally controlled robots (semi-transparent) providing disturbances. The robot must wash the object "A" and then put it into the storage.

both A and D. The new subtree is moved in the BT to a position with a higher priority (See Algorithm 5 Line 12) and the resulting BT is the one depicted in Figure 7.13.

Note that the final BT depicted in Figure 7.13 is similar to the planning and execution tree of [33] with the key difference that the BT enables the continual monitoring of the plan progress, as described in [49]. For example, if A slips out of the robot gripper, the robot will automatically stop and pick it up again without the need to re-plan or change the BT. Moreover if D moves away from the storage while the robot is approaching it to remove it, the robot aborts the plan to remove D and continues with the plan to place A in the storage room. Hence we can claim that the PA-BT is reactive. The execution is exactly the same as [33]: the robot removes the obstructing objects B and C then places A into the washer. When A is clean, the robot picks it up, but then it has to unpick it since it has to move D away from the storage. This is a small drawback of this type of planning algorithms. Again, as stressed in [25] the cost of a non-optimal plan is often much lower than the cost of extensive modeling, information gathering and thorough deliberation needed to achieve optimality.

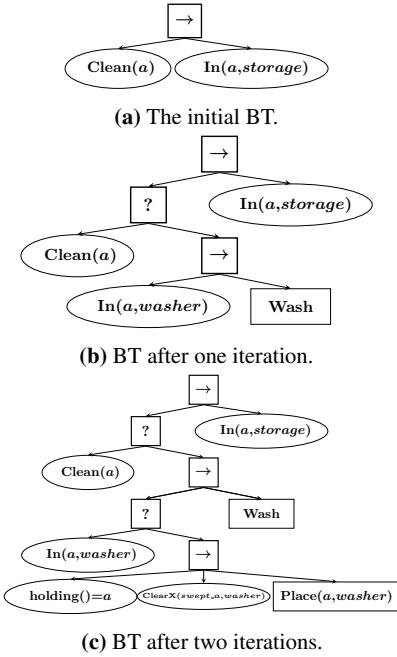
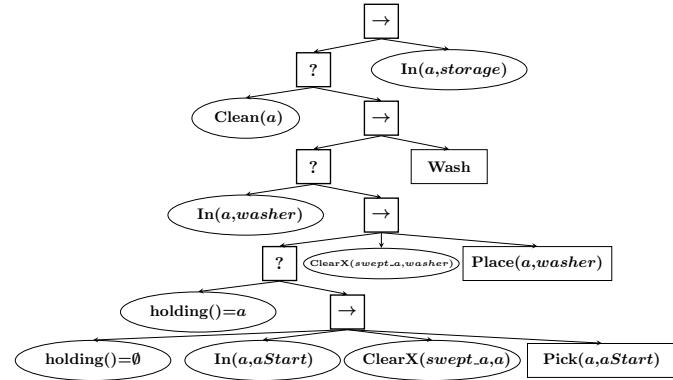
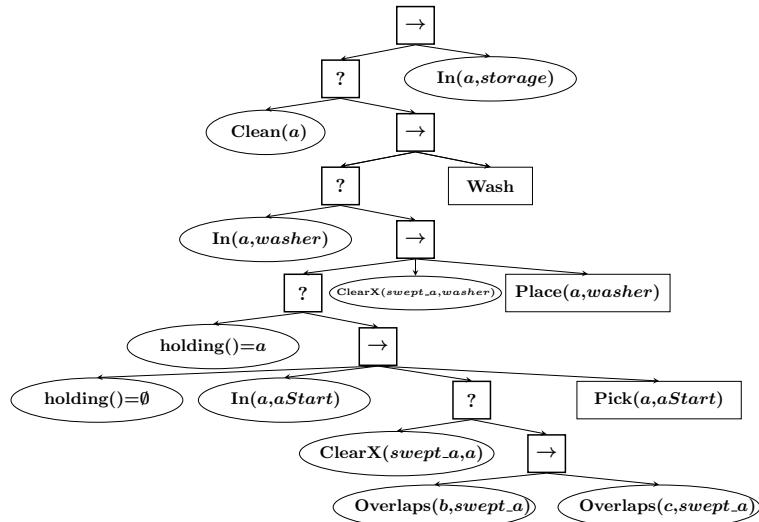


Fig. 7.10: BT updates during the execution of Example 7.4.



(a) BT after three iterations.



(b) BT after four iterations.

Fig. 7.11: BT updates during the execution of Example 7.4.

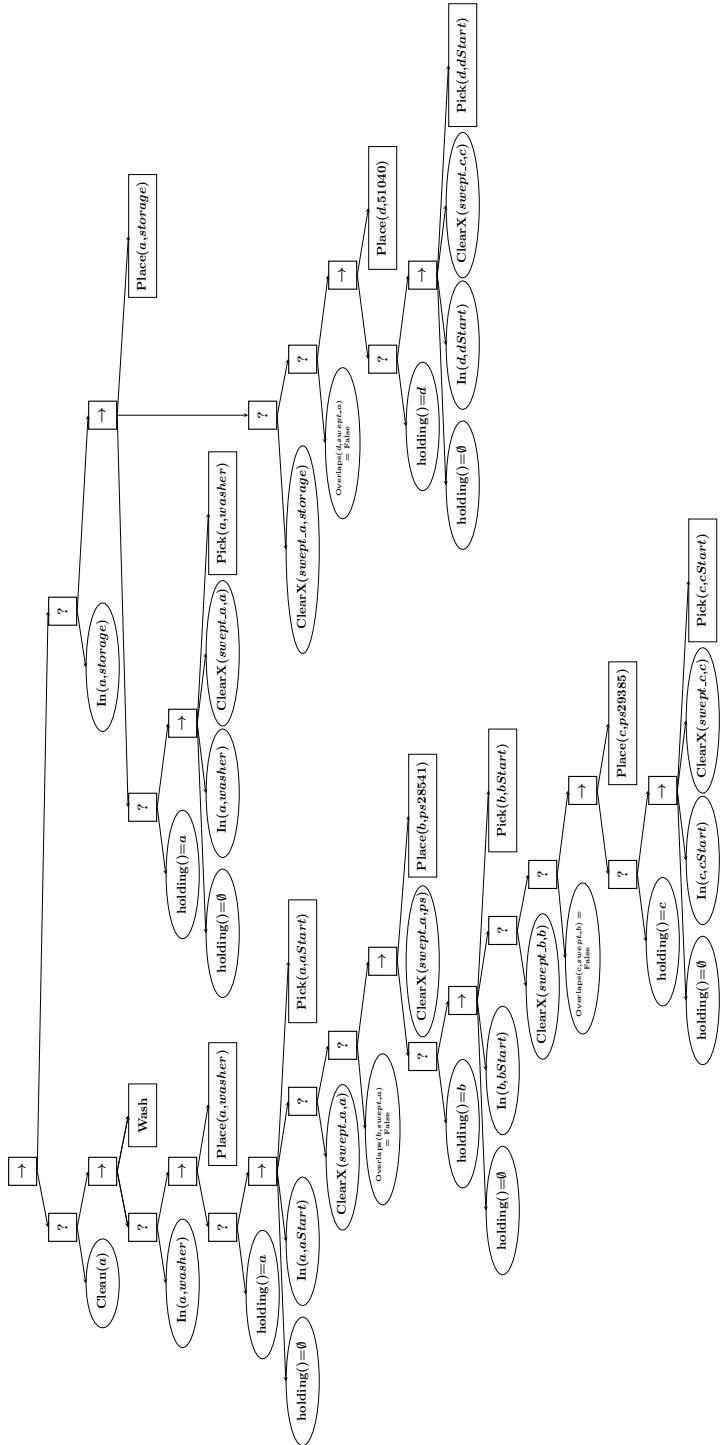


Fig. 7.12: BT containing a conflict. The subtree created to achieve $\text{ClearX}(\text{swept}, a, \text{storage})$ is in conflict with the subtree created to achieve $\text{holding}() = a$.

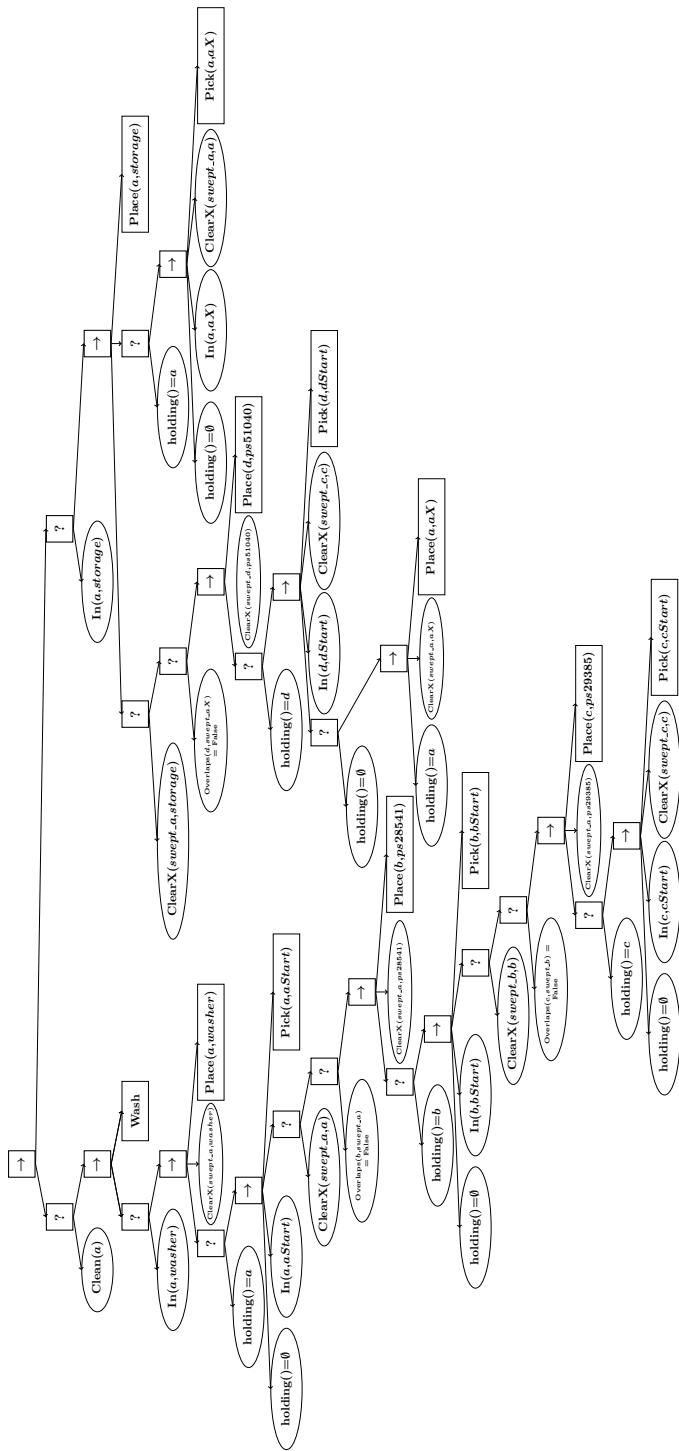


Fig. 7.13: Conflict free BT obtained.

7.1.6 Reactiveness

In this section we show how BTs enable a *reactive* blended acting and planning, providing concrete examples that highlight the importance of reactivity in robotic applications.

Reactivity is a key property for online deliberation. By reactivity we mean *the capability of dealing with drastic changes in the environment in short time*. The domains we consider are highly dynamic and unpredictable. To deal with such domains, a robot must be reactive in both planning and acting.

If an external event happens that the robot needs to react to, one or more conditions will change. The next tick after the change will happen at most a time $\frac{1}{f_t}$ after the change. Then a subset of all Conditions, Sequences and Fallbacks will be evaluated to either reach an Action, or to invoke additional planning. This takes less than Δt . The combined reaction time is thus bounded above by $\frac{1}{f_t} + \Delta t$.

Remark 7.2. Note that Δt is strictly dependent on the real world implementation. Faster computers and better code implementation allows a smaller Δt .

We are now ready to show three colloquial examples, one highlighting the reactive acting (preemption and re-execution of subplans if needed), and two highlighting the reactive planning, expanding the current plan as well as exploiting serendipity [36].

Example 7.5 (Reactive Acting). Consider the robot in Figure 7.9, running the BT of Figure 7.13. The object A is not clean and it is not in the washer, however the robot is holding it. This results in the Condition nodes *Clean(a)* and *In(a,washer)* returning *Failure* and the Condition node *holding()=a* *Success*. According to the BT's logic, the ticks traverse the tree and reach the action *Place(a,washer)*. Now, due to vibrations during movements, the object slips out of the robot's grippers. In this new situation the Condition node *holding()=a* now returns *Failure*. The ticks now traverse the tree and reach the action *Pick(a,aCurrent)* (i.e. pick A from the current position, whose preconditions are satisfied). The robot then re-picks the object, making the Condition node *holding()=a* returning *Success* and letting the robot resume the execution of *Place(a,washer)*.

Example 7.6 (Reactive Planning). Consider the robot in Figure 7.9, running the BT of Figure 7.13. The object A is clean, it is not in the storage and the robot is not holding it. This results in the Condition nodes *holding()=a* and *In(a,storage)* returning *Failure* and the Condition nodes *holding()=∅*, *Clean(a)* and *ClearX(swept_a,washer)* returning *Success*. According to the BT's logic, the ticks traverse the tree and reach the action *Pick(a,washer)* that let the robot approach the object and then grasp it. While the robot is approaching A, an external uncontrolled robot places an object in front of A, obstructing the passage. In this new situation the Condition node *ClearX(swept_a,washer)* now returns *Failure*. The action *Pick(a,washer)* no longer receives ticks and it is then preempted. The BT is expanded accordingly finding a subtree to make *ClearX(swept_a,washer)* return

Success. This subtree will make the robot pick the obstructing object and remove it. Then the robot can finally reach *A* and place it into the storage.

Example 7.7 (Serendipity Exploitation). Consider the robot in Figure 7.9, running the BT of Figure 7.13. The object *A* is not clean, it is not in the washer and the robot is not holding it. According to the BT logic, the ticks traverse the tree and reach the action *Pick(b,bStart)*. While the robot is reaching the object, an external uncontrolled agent picks *B* and removes it. Now the condition *Overlaps(b,swept_a) = False* returns *Success* and the BT preempts the execution of *Pick(b,bStart)* and skips the execution of *Places(b,ps28541)* going directly to execute *Pick(a,aStart)*.

Hence the BT encodes reactivity, (re)satisfaction of subgoals whenever these are no longer achieved, and the exploitation of external subgoal satisfaction after a change in the environment. Note that in the Examples 7.5 and 7.7 above, PA-BT did not replan.

7.1.7 Safety

In this section we show how BTs allow a *safe* blended acting and planning, providing a concrete example that highlights the importance of safety in robotics applications.

Safety is a key property for robotics applications. By safety we mean *the capability of avoiding undesired outcomes*. The domains we usually consider have few catastrophic outcomes of actions and, as highlighted in [33], the result of an action can usually be undone by a finite sequence of actions. However there are some cases in which the outcome of the plan can damage the robot or its surroundings. These cases are assumed to be identified in advance by human operators, who then add the corresponding sub-BT to guarantee the avoidance of them. Then, the rest of the BT is expanded using the algorithm described above.

We are now ready to show a colloquial example.

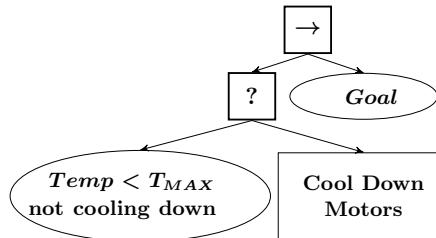


Fig. 7.14: A safe BT for Example 7.8. The BT guaranteeing safety is combined with the mission objective constraint.

Example 7.8 (Safe Execution). Consider the multipurpose robot of Example 7.4. Now, due to overheating, the robot has to stop whenever the motors' temperatures reach a given threshold, allowing them to cool down. This situation is relatively easy to model, and a subtree to avoid it can be designed as shown in Figure 7.14. When running this BT, the robot will preempt any action whenever the temperature is above the given threshold and stay inactive until the motors have cooled down to the temperature where Cool Down Motors return Success. Note that the Not Cooling Down part of the condition is needed to provide hystereses. The robot stops when T_{MAX} is reached, and waits until Cool Down Motors return Success at some given temperature below T_{MAX} . To perform the actual mission, the BT in Figure 7.14 is executed and expanded as explained above.

Thus, we first identify and handle the safety critical events separately, and then progress as above without jeopardizing the safety guarantees. Note that the tree in Figure 7.14, as well as all possible expansions of it using the PA-BT algorithm is *safe* (see Section 5.3).

7.1.8 Fault Tolerance

In this section we show how BTs enable a *fault tolerant* blended acting and planning, providing concrete examples that highlight the importance of fault tolerance in robotics application.

Fault tolerance is a key property for real world problem domains where no actions are guaranteed to succeed. By fault tolerant we mean *the capability of operating properly in the event of Failures*. The robots we consider usually have more than a single way of carrying out a high level task. With multiple options, we can define a priority measure for these options. In PA-BT, the actions that achieve a given goal are collected in a Fallback composition (Algorithm 6 Line 9). The BT execution is such that if, at execution time, one option fails (e.g. a motor breaks) the next option is chosen without replanning or extending the BT.

Example 7.9 (Fault Tolerant Execution). Consider a more advanced version the multipurpose robot of Example 7.4 having a second arm. Due to this redundancy, all the pick-and-place tasks can be carried out with either hands. In the approach (Algorithm 6 Line 9) all the actions that can achieve a given subgoal are collected in a Fallback composition. Thus, whenever the robot fails to pick an object with a gripper, it can directly try to pick it with the other gripper.

7.1.9 Complex Execution on Realistic Robots

In this section, we show how the PA-BT approach scales to complex problems using two different scenarios. First, a KUKA Youbot scenario, where we show the applicability of PA-BT on dynamic and unpredictable environments, highlighting the importance of continually planning and acting. Second, an ABB Yumi industrial manipulator scenario, where we highlight the applicability of PA-BT to real world plans that require the execution of a long sequence of actions. The experiments were carried out using the physics simulator V-REP, in-house implementations of low level controllers for actions and conditions, and an open source BT library¹. Figures 7.16 and 7.21 show the execution of two KUKA youbot experiments and Figure 7.24 show the execution of one ABB Yumi robot experiment. A video showing the executions of all the experiments is publicly available.² All experiments show the reactivity of the PA-BT approach, one experiment is then extended to show safety maintenance and fault tolerance.

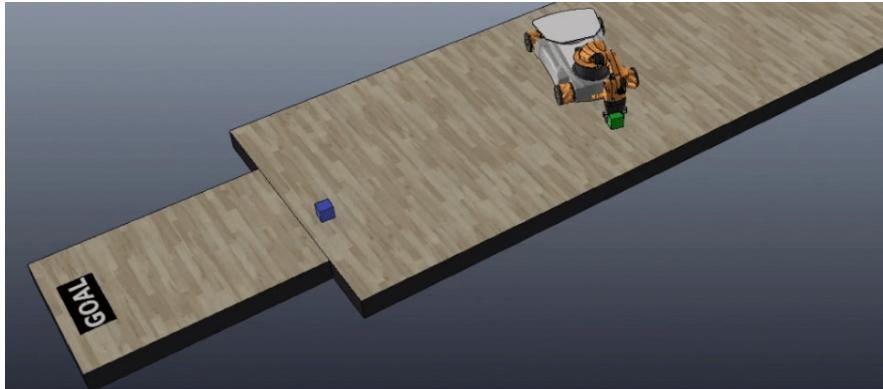
7.1.9.1 KUKA Youbot experiments

In these scenarios, which are an extension of Examples 1 and 2, a KUKA Youbot has to place a green cube on a goal area, see Figures 7.16 and 7.21. The robot is equipped with a single or dual arm with a simple parallel gripper. Additional objects may obstruct the feasible paths to the goal, and the robot has to plan when to pick and where to place the obstructing objects. Moreover, external actors co-exist in the scene and may force the robot to replan by modifying the environment, e.g. by picking and placing the objects.

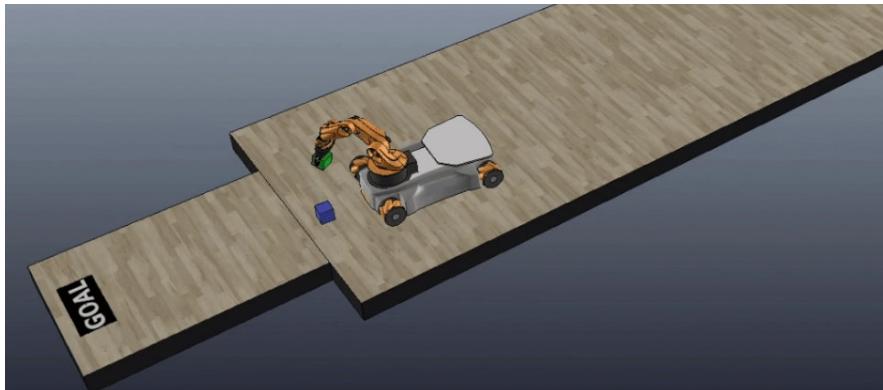
Experiment 7.1 (Static Environment) *In this experiment the single armed version of robot is asked to place the green cube in the goal area. First expansions of the BT allow the robot to pick up the desired object (see Figure 7.15a). Now the robot has to find a collision free path to the goal. Due to the shape of the floor and the position of the obstacle (a blue cube) the robot has to place the obstacle to the side. To do so the robot has to reach the obstacle and pick it up. Since this robot has a single arm, it needs to ungrasp the green cube (see Figure 7.15b) before placing the blue cube on the side (see Figure 7.15c). The robot then can re-grasp the green cube (without extending the plan) and approach the goal region. Now, due to vibrations, the green cube slips out of the robot gripper (see Figure 7.16a). The robot aborts its subplan to reach the goal and re-executes the subplan to grasp the object. Finally the robot places the green cube in the desired area (see Figure 7.16b).*

¹ http://wiki.ros.org/behavior_tree

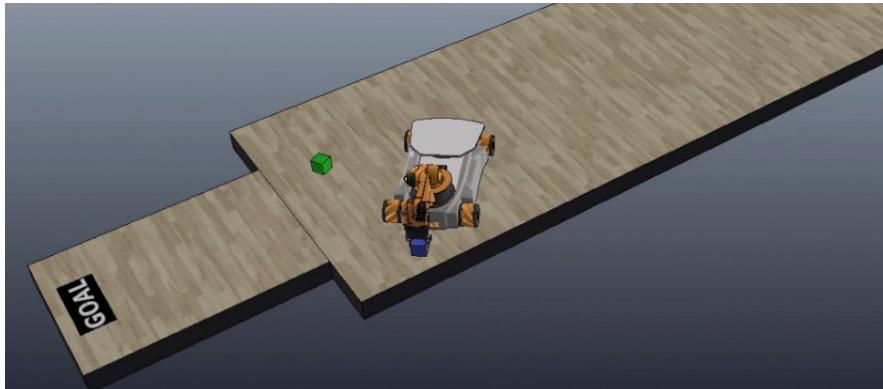
² <https://youtu.be/mhYuyB0uCLM>



(a) The robot picks the desired object: a green cube.

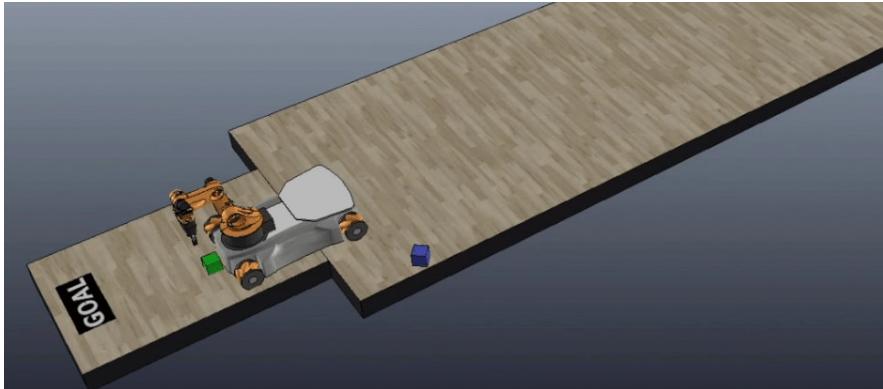


(b) The robot has to move the blue cube away from the path to the goal. But the robot is currently grasping the green cube. Hence the subtree created to move the blue cube needs to have a higher priority.

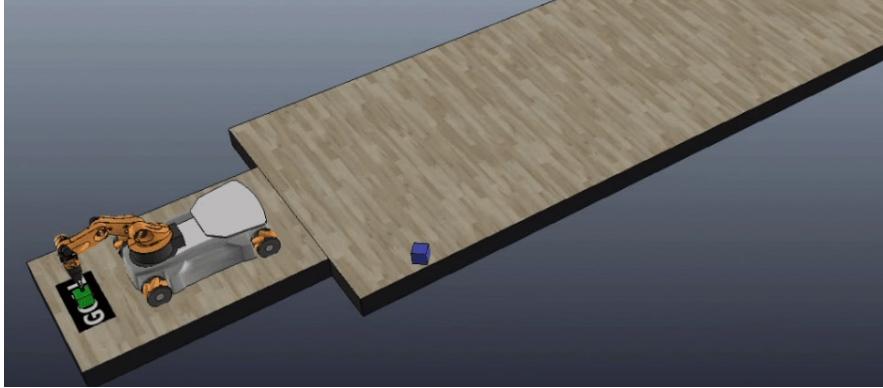


(c) The blue cube is moved to the side.

Fig. 7.15: Execution of a Simple KUKA Youbot experiment.



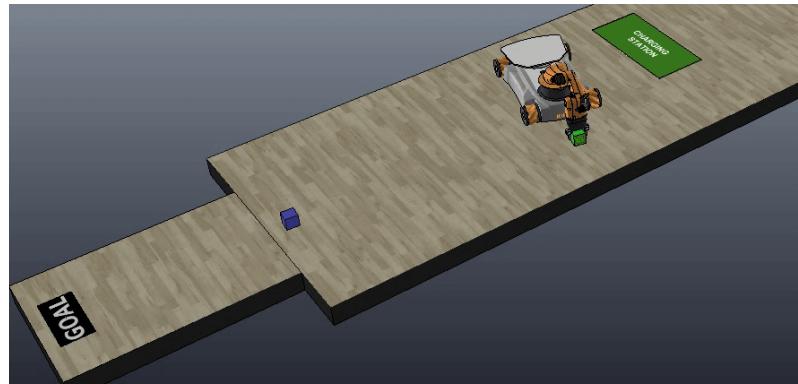
(a) While the robot is moving towards the goal region, the green cube slips out of the gripper. The robot reactively preempts the subtree to move to the goal and re-executes the subtree to grasp the green cube. Without replanning.



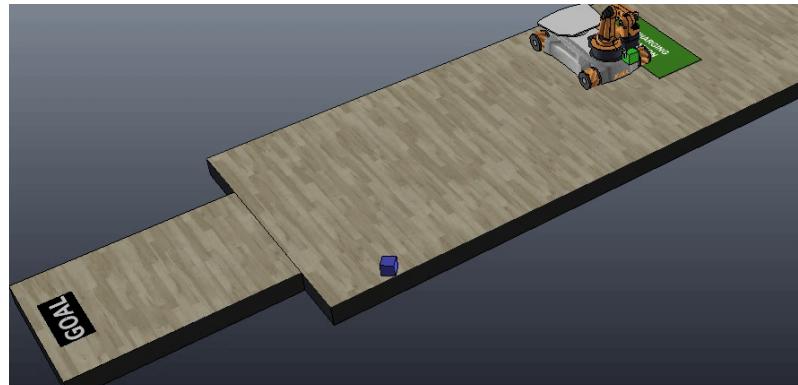
(b) The robot places the object onto the desired location.

Fig. 7.16: Execution of a Simple KUKA Youbot experiment.

Experiment 7.2 (Safety) In this experiment the robot is asked to perform the same task as in Experiment 7.1 with the main difference that now the robot's battery can run out of power. To avoid this undesired irreversible outcome, the initial BT is manually created in a way that is similar to the one in Figure 7.14, managing the battery charging instead. As might be expected, the execution is similar to the one described in Experiment 7.1 with the difference that the robot reaches the charging station whenever needed: The robot first reaches the green cube (see Figure 7.17a). Then, while the robot is approaching the blue cube, the battery level becomes low. Hence the subplan to reach the blue cube is aborted and the subplan to charge the battery takes over (see Figure 7.17b). When the battery is charged the robot can resume its plan (see Figure 7.18a) and complete the task (see Figure 7.18b).

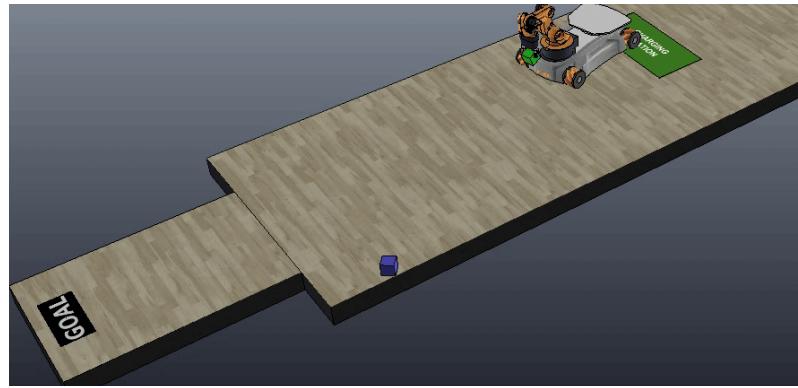


(a) The robot picks the desired object, a green cube.

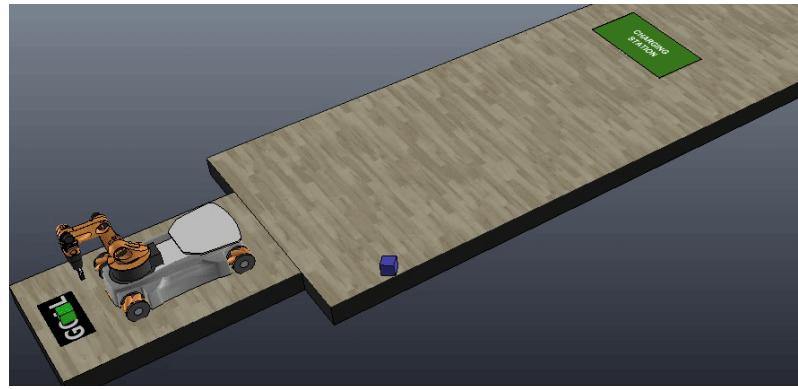


(b) Due to the low battery level, the robot moves to the charging station.

Fig. 7.17: Execution of a KUKA Youbot experiment illustrating safety.



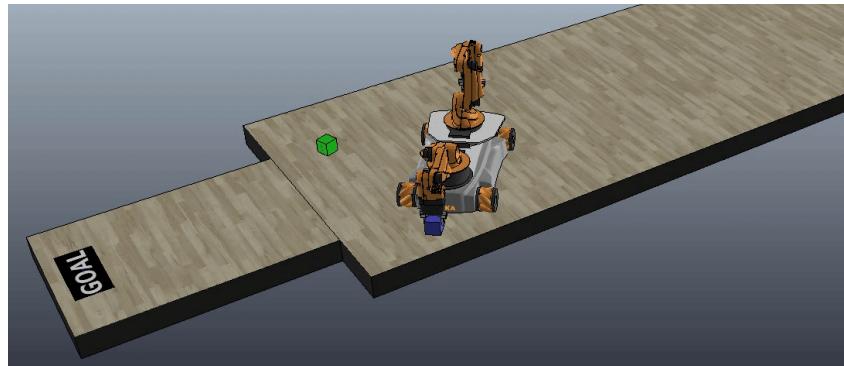
(a) Once the battery is charged, the robot resumes its plan.



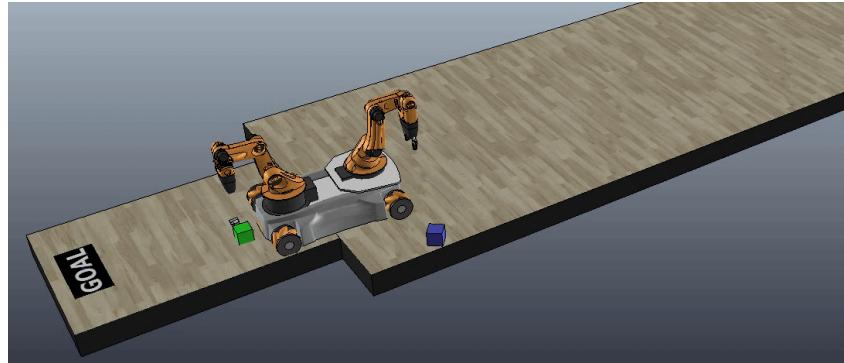
(b) The robot places the object onto the desired location.

Fig. 7.18: Execution of a KUKA Youbot experiment illustrating safety.

Experiment 7.3 (Fault Tolerance) In this experiment the robot is asked to perform the same task as in Experiment 7.1 with the main difference that the robot is equipped with an auxiliary arm and a fault can occur to either arm, causing the arm to stop functioning properly. The robot starts the execution as in the previous experiments (see Figure 7.19a). However while the robot is approaching the goal area, the primary arm breaks, making the green cube fall on the ground (see Figure 7.19b). The robot now tries to re-grasp the object with the primary arm, but this action fails since the grippers are no longer attached to the primary arm, hence the robot tries to grasp the robot with the auxiliary arm. However the auxiliary arm is too far from the object, and thus the robot has to move in a different position (see Figure 7.20a) such that the object can be grasped (see Figure 7.20b) and the execution can continue.

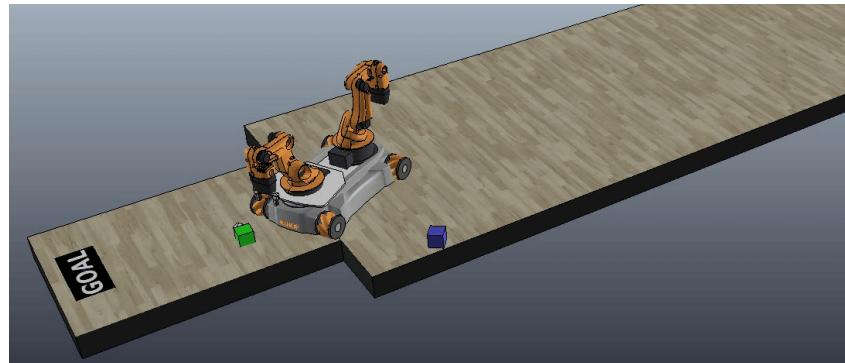


(a) The robot moves the blue cube away from the path to the goal.

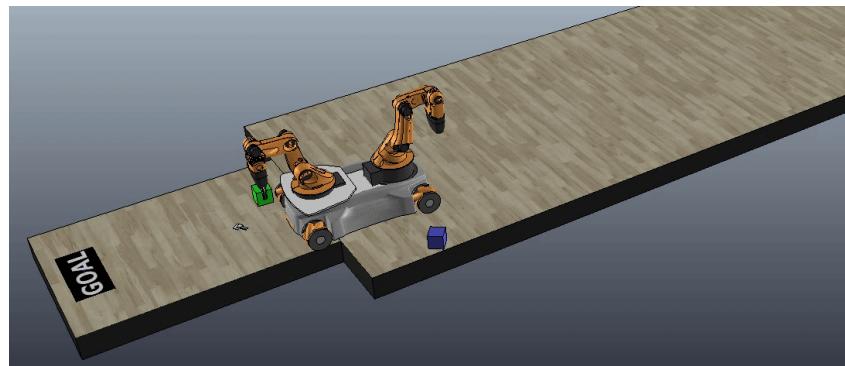


(b) A fault occurs on the primary arm (the grippers break) and the green cube falls to the floor.

Fig. 7.19: Execution of a KUKA Youbot experiment illustrating fault tolerance.



(a) The robot rotates to have the object closer to the auxiliary arm.



(b) The robot grasps the object with the auxiliary arm.

Fig. 7.20: Execution of a KUKA Youbot experiment illustrating fault tolerance.

Experiment 7.4 (Dynamic Environment) In this experiment the single armed version of the robot co-exists with other uncontrolled external robots. The robot is asked to place the green cube in the goal area on the opposite side of the room. The robot starts picking up the green cube and moves towards an obstructing object (a blue cube) to place it to the side (see Figure 7.21a). Being single armed, the robot has to ungrasp the green cube (see Figure 7.21b) to grasp the blue one (see Figure 7.22a). While the robot is placing the blue cube to the side, an external robot places a new object between the controlled robot and the green cube (see Figure 7.22b). The plan of the robot is then expanded to include the removal of this new object (see Figure 7.22c). Then the robot can continue its plan by re-picking the green cube, without replanning. Now the robot approaches the yellow cube to remove it (see Figure 7.23a), but before the robot is able to grasp the yellow cube, another external robot picks up the yellow cube (see Figure 7.23b) and places it to the side. The subplan for removing the yellow cube is skipped (without replanning) and the robot continues its task until the goal is reached (see Figure 7.23c).



(a) The robot picks the desired object, a green cube.

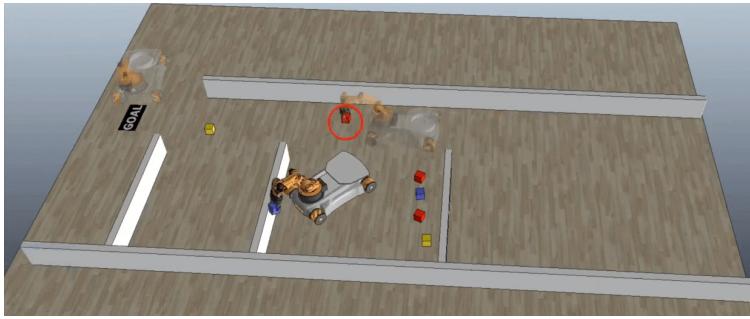


(b) The blue cube obstructs the path to the goal region. The robot drops the green cube in order to pick the blue cube.

Fig. 7.21: Execution of a complex KUKA Youbot experiment.



(a) The robot picks the blue cube.

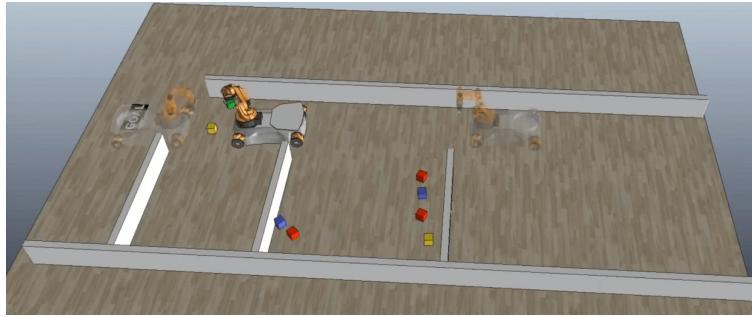


(b) While the robot moves the blue cube away from the path to the goal, an external agent places a red cube between the robot and the green cube.

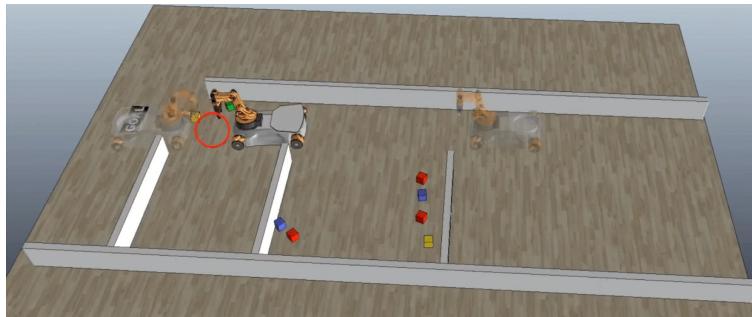


(c) The robot moves the red cube away from the path to the goal.

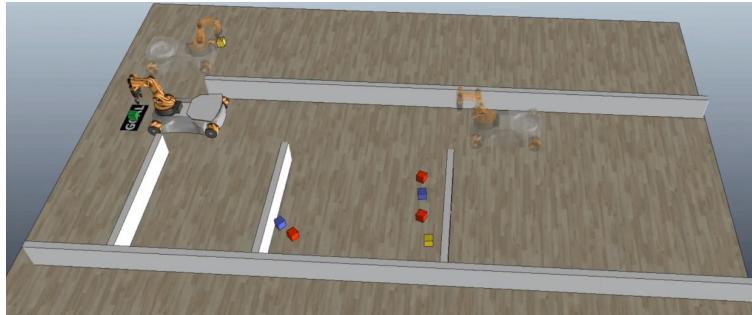
Fig. 7.22: Execution of a complex KUKA Youbot experiment.



(a) The yellow cube obstructs the path to the goal region. The robot drops the green cube in order to pick the yellow cube.



(b) While the robot approaches the yellow cube, an external agent moves the yellow cube away.



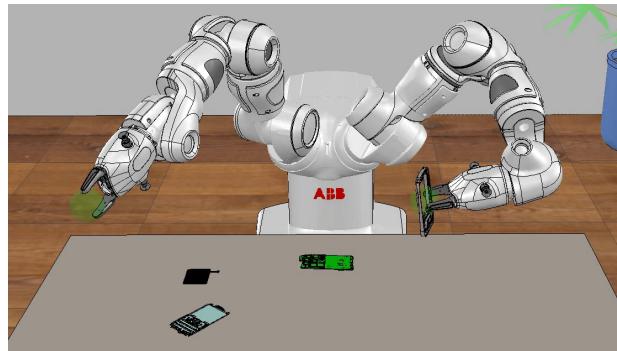
(c) The robot picks the green cube and places it onto the goal region.

Fig. 7.23: Execution of a complex KUKA Youbot experiment.

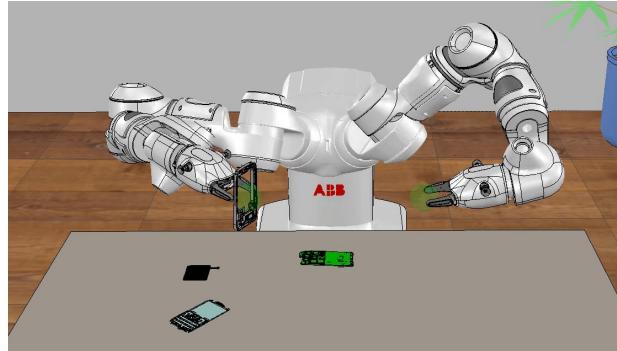
7.1.9.2 ABB Yumi experiments

In these scenarios, an ABB Yumi has to assemble a cellphone whose parts are scattered across a table, see Figure 7.24. The robot is equipped with two arms with simple parallel grippers, which are not suitable for dexterous manipulation. Furthermore, some parts must be grasped in a particular way to enable the assembly operation.

Experiment 7.5 *In this experiment, the robot needs to re-orient some cellphone's parts to expose them for assembly. Due to the gripper design, the robot must reorient the parts by performing multiple grasps transferring the part to the other gripper, see Figure 7.24b, effectively changing its orientation (see Figures 7.25a-7.25b).*

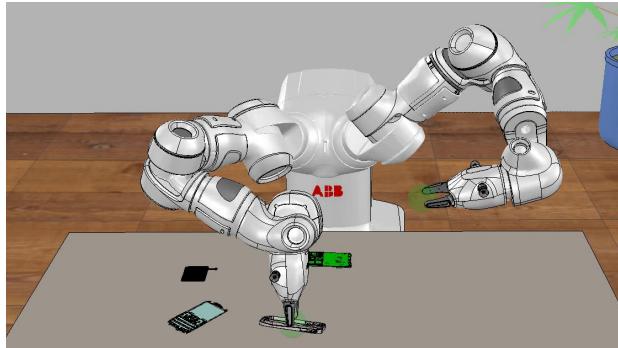


(a) The robot picks the cellphone's chassis. The chassis cannot be assembled with this orientation.

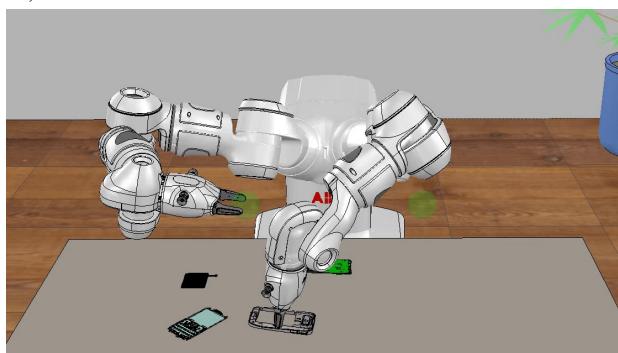


(b) The chassis is handed over the other gripper.

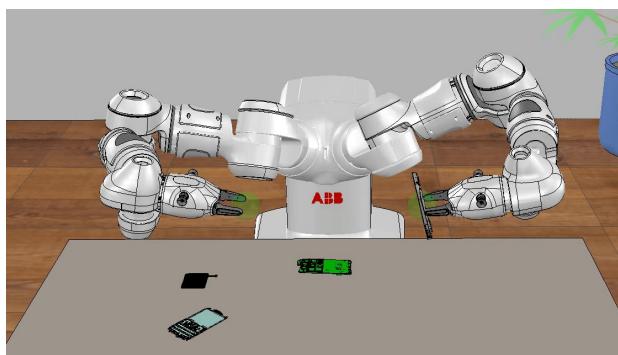
Fig. 7.24: Execution of an ABB Yumi experiment.



(a) The chassis is placed onto the table with a different orientation than before (the opening part is facing down now).

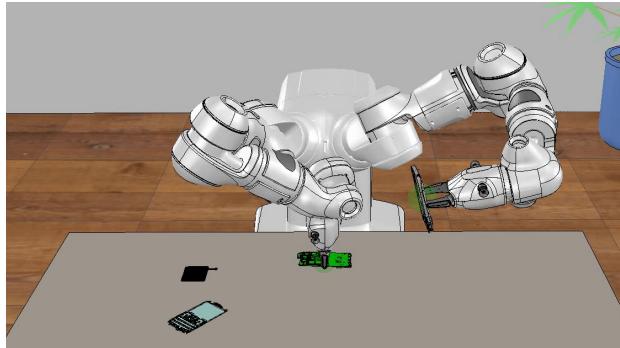


(b) The robot picks the chassis with the new orientation.

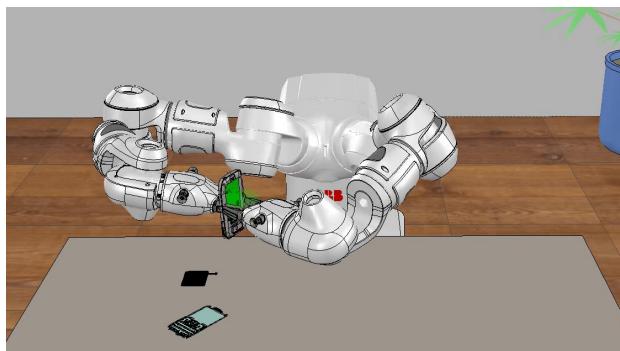


(c) The chassis can be assembled with this orientation.

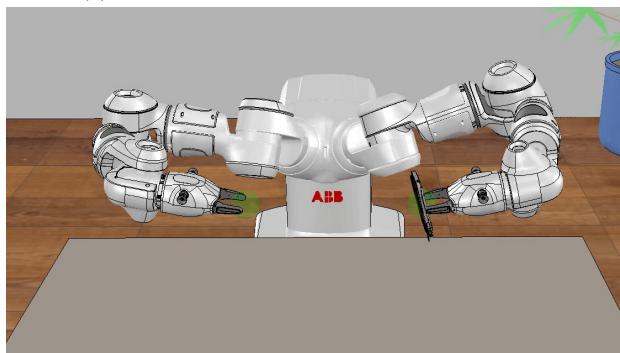
Fig. 7.25: Execution of an ABB Yumi experiment.



(a) The robot pick the next cellphone's part to be assembled (the motherboard).



(b) The motherboard and the chassis are assembled.



(c) The robot assembles the cellphone correctly.

Fig. 7.26: Execution of an ABB Yumi experiment.

7.2 Planning using A Behavior Language (ABL)

To contrast the PA-BT approach described above we will more briefly present planning using A Behavior Language³ (ABL, pronounced “able”) [74, 75].

ABL was designed for the dialogue game Facade [39], but is also appreciated for its ability to handle the planning and acting on multiple scales that is often needed in both robotics and games, and in particular essential in so-called Real-Time Strategy games. In such games, events take place both on a long term time scale, where strategic decisions has to be made regarding e.g., what buildings to construct in the next few minutes, and where to locate them, and on a short term time scale, where tactical decisions has to be made regarding e.g., what opponents to attack in the next few seconds. Thus, performing well in multi-scale games requires the ability to execute short-term tasks while working towards long-term goals. In this section we will first use the the Pac-Man game as an example, where the short term decisions concern avoiding ghosts and the long term decisions concern eating all the available pills.

7.2.1 An ABL Agent

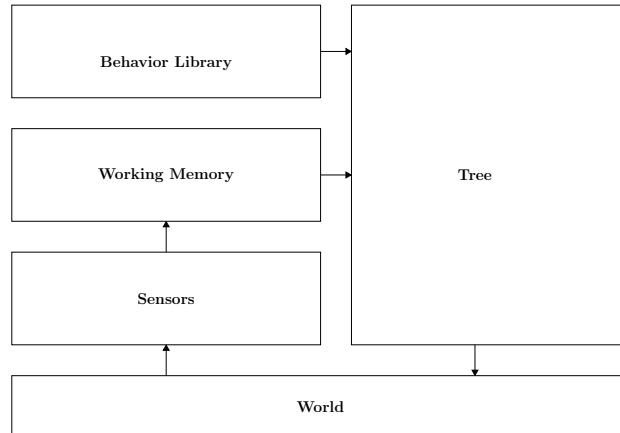


Fig. 7.27: Architecture of a ABL agent.

³ In the first version of ABL, the tree structure that stores all the goals is called “Active Behavior Tree”. This tree is related to, but different from the BTs we cover in this book (e.g. no Fallbacks and no ticks). Later work used the classic BT formulation also for ABL.

An agent running an ABL planner is called an *ABL Agent*. Figure 7.27 depicts the architecture of an ABL agent. The **behavior library** is a repository of pre-defined behaviors where each behavior consists of a set of actions to execute to accomplish a goal (e.g. move to given location). There are two kinds of behaviors in ABL, *sequential behaviors* and *parallel behaviors*. The **working memory** is a container for any information the agent must access during execution (e.g. unit's position on the map). The **sensors** report information about changes in the world by writing that information into the working memory (e.g. when another agent is within sight). The **tree** (henceforth denoted ABL tree to avoid confusion) is an execution structure that describes how the agent will act, and it is dynamically extended. The ABL tree is initially defined as a collection of all the agent's goals, then it is recursively extended using a set of instructions that describe how to expand the tree. Figure 7.28 shows the initial ABL tree for the ABL Pac-Man Agent. Below we describe the semantic of ABL tree instructions.

```
initial-tree{
    subgoal handleGhosts();
    subgoal eatAllPills();
}
```

Fig. 7.28: Example of an initial ABL tree instruction of the ABL agent for Pac-Man.

A **subgoal** instruction establishes goals that must be accomplished in order to achieve the main task.

An **act** instruction describes an action what will change the physical state of the agent or the environment. A **mental act** instruction describes pure computation, as mathematical computations or modifications to working memory.

Act and mental act are both parts of behaviors, listed in the behavior library, as the examples in Figures 7.29-7.30.

```
sequential behavior eatAllPills(){
    mental_act computeOptimalPath();
    act followOptimalPath();
}
```

Fig. 7.29: Example of the content of the behavior library.

A **spawngoal** instruction is the key component for expanding the BT. It defines the subgoals that must be accomplished to achieve a behavior.

Remark 7.3. The main difference between the instructions *subgoal* and *spawngoal* is that the spawngoal instruction is evaluated in a lazy fashion, expanding the tree only when the goal spawned is needed for the first time, whereas the subgoal instruction

```
parallel behavior eatAllPills(){
    mental_act recordData();
    act exploreRoom();
}
```

Fig. 7.30: Example of the content of the behavior library.

```
parallel behavior handleGhosts(){
    spawngoal handleDeadlyGhosts();
    spawngoal handleScaredGhosts();
}
```

Fig. 7.31: Example of the content of the behavior library.

is evaluated in a greedy fashion, requiring the details on how to carry out the subgoal at design time.

```
sequential behavior handleDeadlyGhost(){
    precondition {
        (deadlyGhostClose);
    }
    act keepDistanceFromDeadlyGhost();
}
```

Fig. 7.32: Example of a precondition instruction of the ABL agent for Pac-Man.

A **precondition** instruction specifies under which conditions the behavior can be selected. When all of the preconditions are satisfied, the behavior can be selected for execution or expansion, as in Figure 7.32.

```
conflict keepDistanceFromDeadlyGhost followOptimalPath;
```

Fig. 7.33: Example of a conflict instruction of the ABL agent for Pac-Man.

A **conflict** instruction specifies priority order if two or more actions are scheduled for execution at the same time, using the same (virtual) actuator.

7.2.2 The ABL Planning Approach

In this section, we present the ABL planning approach. Formally, the approach is described in Algorithms 8-10. First, will give an overview of the algorithms and see how they are applied to the problem described in *Example 7.10*, to iteratively create the BTs of Figure 7.36. Then, we will discuss the key steps in more detail.

Algorithm 8: main loop - input(initial ABL tree)

```

1  $\mathcal{T} \leftarrow \text{ParallelNode}$ 
2 for subgoal in initial-tree do
3    $\mathcal{T}_g \leftarrow \text{GetBT}(\text{subgoal})$ 
4    $\mathcal{T}.\text{AddChild}(\mathcal{T}_g)$ 
5 while True do
6    $\text{Execute}(\mathcal{T})$ 

```

Algorithm 9: GetBT - input(goal)

```

1  $\mathcal{T}_g \leftarrow \emptyset$ 
2 if goal.behavior is sequential then
3    $\mathcal{T}_g \leftarrow \text{SequenceNode}$ 
4 else
5    $\mathcal{T}_g \leftarrow \text{ParallelNode}$ 
6 Instructions  $\leftarrow \text{GetInstructions}(goal)$ 
7 for instruction in Instructions do
8   switch instruction do
9     case act do
10     $\mathcal{T}_g.\text{AddChild}(\text{ActionNode}(\text{act}))$ 
11    case mental act do
12     $\mathcal{T}_g.\text{AddChild}(\text{ActionNode}(\text{mental act}))$ 
13    case spawngoal do
14     $\mathcal{T}_g.\text{AddChild}(\text{PlaceholderNode}(\text{spawngoal}))$ 
15 if goal.precondition is not empty then
16    $\mathcal{T}'_g \leftarrow \text{SequenceNode}$ 
17   for proposition in precondition do
18      $\mathcal{T}'_g.\text{AddChild}(\text{ConditionNode}(\text{proposition}))$ 
19    $\mathcal{T}'_g.\text{AddChild}(\mathcal{T}_g)$ 
20   return  $\mathcal{T}'_g$ 
21 else
22   return  $\mathcal{T}_g$ 

```

Algorithm 10: Execute - input(node)

```

1 switch node.Type do
2   case ActionNode do
3     if not in conflict then
4         |_ Tick(node)
5   case PlaceholderNode do
6     node ← GetBT(node.goal)
7     Execute(node)
8   otherwise do
9       |_ Tick(node)

```

The execution of the algorithm is simple. It first creates a BT from the initial ABL tree t , collecting all the subgoals in a BT Parallel composition (Algorithm 8, Line 1), then, the tree is extended by finding a BT for each subgoal (Algorithm 8, Line 3). Each subgoal is translated in a corresponding BT node (Sequence or Parallel, according to the behavior in the behavior library) whose children are the instruction of the subgoal. If a behavior has precondition instructions, they are translated into BT Condition nodes, added first as children (Algorithm 9, Line 15). If a behavior has act or mental act instruction, they are translated into BT Action nodes (Algorithm 9, Lines 9-12) and set as children. If a behavior has spawngoal instruction (Algorithm 9, Line 13), this is added as a *placeholder node*, which, when ticked, extends itself as done for the subgoals (Algorithm 10, Line 6).

We are now ready to see how the algorithm is executed in a simple Pac-Man game.

Example 7.10 (Simple Execution in Pac-Man). While Pac-Man has to avoid being eaten by the ghosts, he has to compute the path to take in order to eat all Pills. The ABL tree Pac-Man agent is shown in Figure 7.34. Running Algorithm 8, the initial tree is translated into the BT in Figure 7.35. The subgoal *eatAllPills* is expanded as the sequence of the two BT's Action nodes *computeOptimalPath* and *followOptimalPath*. The subgoal *handleGhosts* is extended as a Sequence composition of the Condition node *ghostClose* (which is a precondition for *handleGhosts*) and a Parallel composition of placeholder nodes *handleDeadlyGhosts* and *handleScaredGhosts*. The BT is ready to be executed. Let's imagine that for a while Pac-Man is free to eat pills without being disturbed by the ghosts. For this time the condition *ghostClose* is always false and the spawn of neither *handleDeadlyGhosts* nor *handleScaredGhosts* is invoked. Imagine now that a Ghost is close for the first time. This will trigger the expansion of *handleDeadlyGhosts* and *handleScaredGhosts*. The expanded tree is shown in Figure 7.36.

```

pacman_agent{
    initial-tree{
        subgoal handleGhosts();
        subgoal eatAllPills();
    }

    sequential behavior eatAllPills(){
        mental_act computeOptimalPath();
        act followOptimalPath();
    }

    parallel behavior handleGhosts(){
        precondition {
            (ghostClose);
        }
        spawngoal handleDeadlyGhosts();
        spawngoal handleScaredGhosts();
    }

    sequential behavior handleScaredGhost(){
        precondition {
            (scaredGhostClose);
        }
        act moveToScaredGhost();
    }

    sequential behavior handleDeadlyGhost(){
        precondition {
            (deadlyGhostClose);
        }
        act keepDistanceFromDeadlyGhost();
    }

    conflict keepDistanceFromDeadlyGhost moveToScaredGhost followOptimalPath;
}

```

Fig. 7.34: ABT tree for Pac-Man.

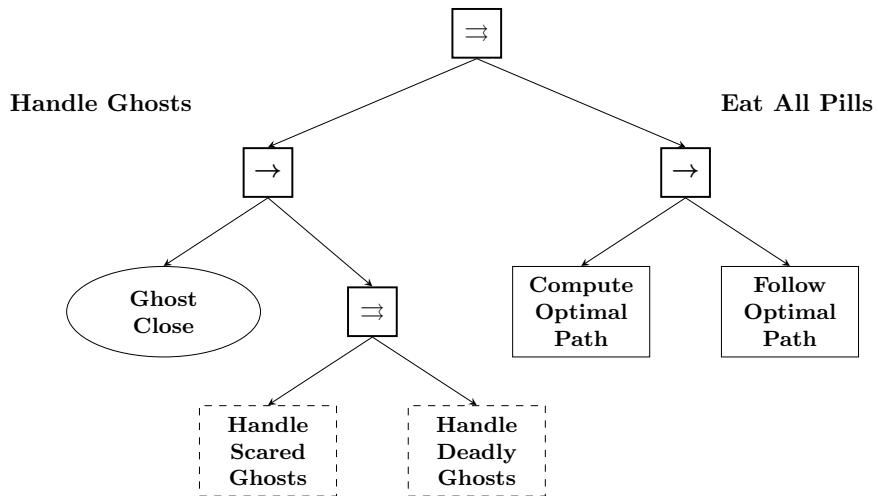


Fig. 7.35: Initial BT of Example 7.10.

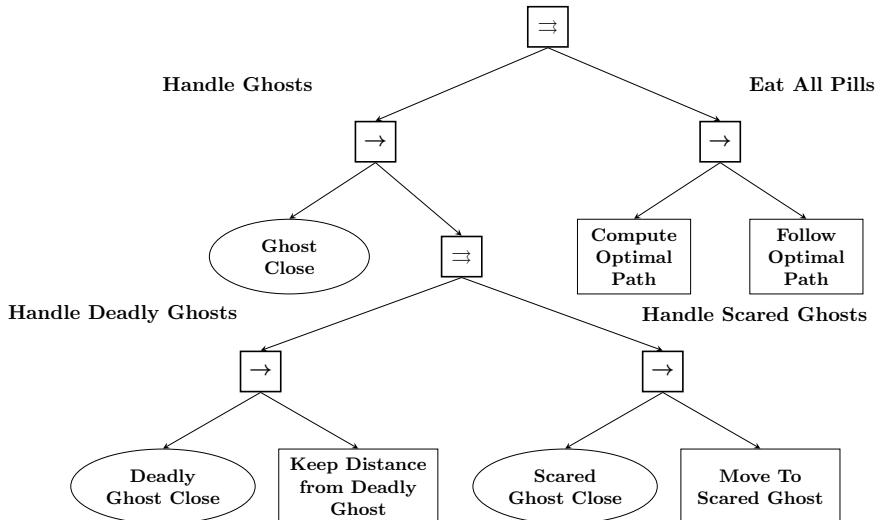


Fig. 7.36: Final BT of Example 7.10.

7.2.3 Brief Results of a Complex Execution in StarCraft



Fig. 7.37: A screenshot of StarCraft showing two players engaged in combat [75].

We will now very briefly describe the results from a more complex scenario, from [75]. One of the most well known strategy computer games that require multi-scale reasoning is the real-time strategy game StarCraft. In StarCraft the players manage groups of units to compete for the control of the map by gathering resources to produce buildings and units, and by researching technologies that unlock more advanced abilities. Building agents that perform well in this domain is challenging due to the large decision space [1]. StarCraft is also a very fast-paced game, with top players performing above 300 actions per minute during peak intensity episodes [40]. This shows that a competitive agent for StarCraft must reason quickly at multiple scales in order to achieve efficient game play.

Example 7.11. The StarCraft ABL agent is composed of three high-level managers: Strategy manager, responsible for the strategy selection and attack timing competencies; Production manager, responsible for the worker units, resource collection, and expansion; and Tactics manager, responsible for the combat tasks and micro-management unit behaviors. The initial ABL tree takes the form of Figure 7.38.

```
initial-tree {
    subgoal ManageTactic();
    subgoal ManageProduction();
    subgoal ManageStrategy();
}
```

Fig. 7.38: ABT tree for StarCraft.

Further discussions of the specific managers and behaviors are available in [75], and a portion of the BT after some rounds of the game is shown in Figure 7.39.

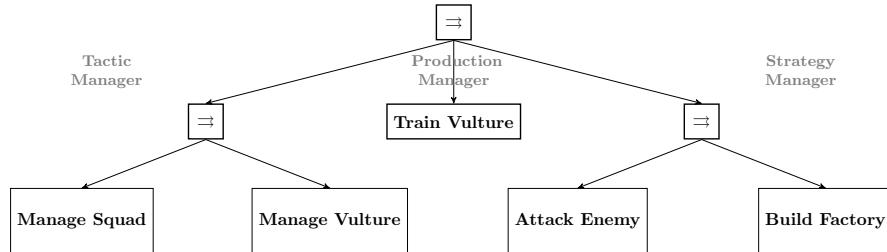


Fig. 7.39: A Portion of the tree of the ABL StarCraft agent.

Map / Race	Protoss	Terran	Zerg
Andromeda	85%	55%	75%
Destination	60%	60%	45%
Heartbreak Ridge	70%	70%	75%
Overall	72%	62%	65%

Table 7.1: Win rate on different map/race combination over 20 trials [74].

In [74] the ABL agent of Example 7.11 was evaluated against the build-in StarCraft AI. The agent was tested against three professional gaming maps: Andromeda, Destination, and Heartbreak Ridge; against all three races: Protoss, Terran, and Zerg over 20 trials. The result are shown in Table 7.1. The ABL agent scored an overall win rate of over 60%, additionally, the agent was capable to perform over 200 game actions per minute, highlighting the capability of the agent to combine low-level tactical task with high-level strategic reasoning.

7.3 Comparison between PA-BT and ABL

So, faced with a BT planning problem, should we choose PA-BT or ABL? The short answer is that PA-BT is focused on creating a BT using a planning approach, whereas ABL is a complete planning language in itself, using BT as an execution tool. PA-BT is also better at exploiting the Fallback constructs of BTs, by iteratively expanding conditions into PPAs, that explicitly include fallback options for making a given condition true.

Chapter 8

Behavior Trees and Machine Learning

In this chapter, we describe how learning algorithms can be used to automatically create BTs, using ideas from [57, 14]. First, in Section 8.2, we present a mixed learning strategy that combines a greedy element with Genetic Programming (GP) and show the result in a game and a robotic example. Then, in Section 8.3, we present a Reinforcement Learning (RL) algorithm applied to BTs and show the results in a game example. Finally in Section 8.5 we overview the main approaches used to learn BTs from demonstration.

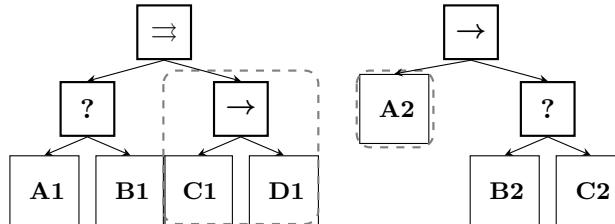
8.1 Genetic Programming Applied to BTs

The capability of an agent to learn from its own experience can be realized by imitating natural evolution. GP is an optimization algorithm that takes inspiration from biological evolution [61] where a set of *individual* policies are evolved until one of them solves a given optimization problem good enough.

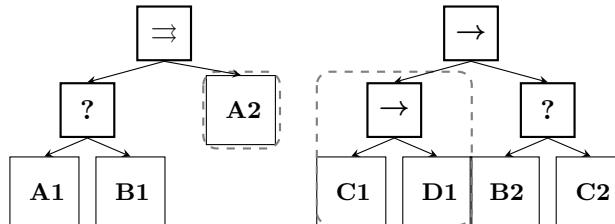
In a GP approach a particular set of individuals is called a *generation*. At each GP iteration, a new generation is created from the previous one. First, a set of individuals are created by applying the operations *cross-over* and *mutation* to the previous generation. Then a subset of the individuals are chosen through *selection* for the next generation based upon a user-defined reward. We will now describe how these three operators can be applied to BTs.

Crossover of two BTs The crossover is performed by randomly swapping a subtree from one BT with a subtree of another BT at any level. Figure 8.1a and Figure 8.1b show two BTs before and after a cross-over operation.

Remark 8.1. The use of BTs as the knowledge representation framework in the GP avoids the problem of logic violation during cross-over stressed in [19]. The logic violation occurs when, after the cross-over, the resulting individuals might not have a consistent logic structure. One example of this is the crossover combination of two



(a) BTs before the cross-over of the highlighted subtrees.



(b) BTs after the cross-over of the highlighted subtrees.

Fig. 8.1: Cross-over operation on two BTs.

FSMs that might lead to a logic violation in terms of some transitions not leading to an existing state.

Mutation of a BT The mutation is an unary operation that replaces a node in a BT with another node of the same type (i.e. it does not replace an execution node with a control flow node or vice versa). Mutations increase diversity, which is crucial in GP. To improve convergence properties it is common to use so-called *simulated annealing*, performing the mutation on a large number of nodes of the first generation of BTs and gradually reducing the number of mutated nodes in each new generation. In this way we start with a very high diversity to avoid getting stuck in possible local minima of the objective function of the optimization problem, and reduce diversity over time as we get closer to the goal.

Selection of BTs In the selection step, a subset of the individuals created by mutation and crossover are selected for the next generation. The process is random, giving each BT a survival probability p_i . This probability is based upon the *reward function* which quantitatively measures the fitness of the agent, i.e., how close the agent gets to the goal. A common method to compute the survival probability of an individual is the Rank Space Method [44], where the designer first sets P_c as the probability of the highest ranking individual, then we sort the BTs in descending order w.r.t. the reward. Finally, the probabilities are defined as follows:

$$p_k = (1 - P_c)^{k-1} P_c \quad \forall k \in \{1, 2, \dots, N-1\} \quad (8.1)$$

$$p_N = (1 - P_c)^{N-1} \quad (8.2)$$

where N is the number of individuals in a generation.

8.2 The GP-BT Approach

In this section we outline the GP-BT approach [14]. We begin with an example, describing the algorithm informally, and then give a formal description in Algorithm 11. GP-BT follows a mixed learning strategy, trying a greedy algorithm first and then applying a GP algorithm when needed. This mixed approach reduces the learning time significantly, compared to using pure GP, while still resulting in a fairly compact BT that achieves the given objective.

We now provide an example to describe the algorithm informally.

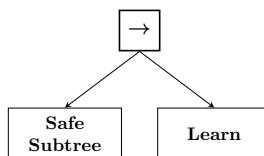


Fig. 8.2: The initial BT is a combination of the BT guaranteeing safety, and the BT *Learn* that will be expanded during the learning.

Example 8.1. Consider the case of the Mario AI setup in Figure 8.5a, starting with the BT in Figure 8.2.

The objective of Mario is to reach the rightmost end of the level. The safety BT is optional, but motivated by the need to enable guarantees that the agent avoids some regions of the state space that are known by the user to be un-safe. Thus, this is the only part of GP-BT that requires user input.

The un-safe regions must have a conservative margin to enable the safety action to act before it is too late (see Section 5.3). Thus we cannot use the enemies as unsafe regions as Mario needs to move very close to those to complete the level. Therefore, for illustrative purposes, we let the safety action guarantee that Mario never reaches the leftmost wall of the level.

Mario starts really close to the left most wall, so the first thing that happens is that the safety action moves Mario a bit to the right. Then the Learn action is executed.

This action first checks all inputs and creates a BT, \mathcal{T}_{cond_t} , of conditions that returns Success if and only if all inputs correspond to the current “situation”, as will be explained below.

Then the learning action executes all single actions available to Mario, e.g. go left, go right, jump, fire etc. and checks the resulting reward.

All actions yielding an increase in the reward are collected in a Fallback composition BT, \mathcal{T}_{acts_i} , sorted with respect to performance, with the best performing action first.

If no single action results in a reward increase, the GP is invoked to find a combination of actions (arbitrary BTs are explored, possibly including parallel, Sequence and Fallback nodes) that produces an increase. Given some realistic assumptions, described in [14], such a combination exists and will eventually be found by the GP according to previous results in [63], and stored in \mathcal{T}_{acts_i} .

Then, the condition BT, \mathcal{T}_{cond_τ} , is composed with the corresponding action BT, \mathcal{T}_{acts_i} , in a Sequence node and the result is added to the previously learned BT, with a higher priority than the learning action.

Finally, the new BT is executed until once again the learning action is invoked.

Algorithm 11: Pseudocode of the learning algorithm

```

1  $\mathcal{T} \leftarrow \text{"Action Learn"}$ 
2 do
3   Tick(SequenceNode( $\mathcal{T}_{safe}, \mathcal{T}$ ))
4   if IsExecuted(Action Learn) then
5      $\mathcal{T}_{cond} \leftarrow \text{GetSituation()} \quad \% \text{Eq (8.3)}$ 
6      $\mathcal{T}_{acts} \leftarrow \text{LearnSingleAction}(\mathcal{T}) \quad \% \text{Eq (8.4)}$ 
7     if  $\mathcal{T}_{acts}.\text{NumOfChildren} = 0$  then
8        $\mathcal{T}_{acts} \leftarrow \text{GetActionsUsingGP}(\mathcal{T}) \quad \% \text{Eq (8.5)}$ 
9     if IsAlreadyPresent( $\mathcal{T}_{acts}$ ) then
10       $\mathcal{T}_{cond_{exist}} \leftarrow \text{GetConditions}(\mathcal{T}_{acts})$ 
11       $\mathcal{T}_{cond_{exist}} \leftarrow \text{Simplify}(\text{FallbackNode}((\mathcal{T}_{cond_{exist}}, \mathcal{T}_{cond})))$ 
12    else
13       $\mathcal{T} \leftarrow \text{FallbackNode}(\text{SequenceNode}(\mathcal{T}_{cond}, \mathcal{T}_{acts}), \mathcal{T}) \quad \% \text{Eq (8.6)}$ 
14     $\rho \leftarrow \text{GetReward}(\text{SequenceNode}(\mathcal{T}_{safe}, \mathcal{T}))$ 
15  while  $\rho < 1$ ;
16 return  $\mathcal{T}$ 

```

8.2.1 Algorithm Overview

To describe the algorithm in detail, we need a set of definitions. First we let a situation be a collection of all conditions, sorted on whether they are true or not, then we define the following:

Situation

$\mathcal{S}(t) = [C_T^{(t)}, C_F^{(t)}]$ is the situation vector, where $C_T^{(t)} = \{C_{T1}, \dots, C_{TN}\}$ is the set of conditions that are true at time t and $C_F^{(t)} = \{C_{F1}, \dots, C_{FM}\}$ is the set of conditions that are false at time t .

Then, using the analogy between AND-OR trees and BTs [12], we create the BT that returns success only when a given situation occurs.

$$\mathcal{T}_{\text{cond}_\tau}$$

$\mathcal{T}_{\text{cond}_\tau}$ is the BT representation of $\mathcal{S}(\tau)$.

$$\begin{aligned} \mathcal{T}_{\text{cond}_\tau} \triangleq & \text{Sequence}(\text{Sequence}(C_{T1}, \dots, C_{TN}), \\ & \text{Sequence}(\text{invert}(C_{F1}), \dots, \text{invert}(C_{FM}))) \end{aligned} \quad (8.3)$$

Figure 8.3 shows a BT composition representing a situation $\mathcal{S}(\tau)$.

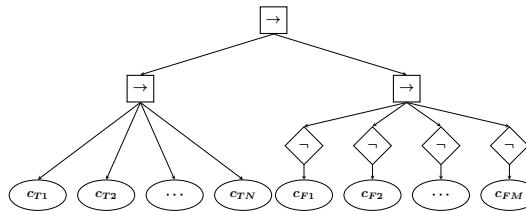


Fig. 8.3: Graphical representation of $\mathcal{T}_{\text{cond}_\tau}$, $c_{Fi} \in C_F^\tau$, $c_{Tj} \in C_T^\tau$. The Decorator is the negation Decorator (i.e. it inverts the Success/Failure status). This BT returns success only when all c_{Tj} are true and all c_{Fi} are false.

$$\mathcal{T}_{\text{acts}_i}$$

Given a small $\varepsilon > 0$, if at least one action results in an increase in reward, $\Delta\rho > \varepsilon$, let A_{P1}, \dots, A_{PN} be the list of actions that result in such an improvement, sorted on the size of $\Delta\rho$, then, $\mathcal{T}_{\text{acts}_i}$ is defined as:

$$\mathcal{T}_{\text{acts}_i} = \text{FallbackWithMemory}(A_{P1}, \dots, A_{PN}) \quad (8.4)$$

else, $\mathcal{T}_{\text{acts}_i}$ is defined as the solution of a GP algorithm that terminates when an improvement such that $\Delta\rho > \varepsilon$ or $\rho(x) = 1$ is found, i.e.:

$$\mathcal{T}_{\text{acts}_i} = GP(\mathcal{S}(t)) \quad (8.5)$$

New BT

If \mathcal{T}_{acts_i} is not contained in the BT, the new BT learned is given as follows:

$$\mathcal{T}_i \triangleq \text{Fallback}(\text{Sequence}(\mathcal{T}_{cond_i}, \mathcal{T}_{acts_i}), \mathcal{T}_{i-1}) \quad (8.6)$$

Else, if \mathcal{T}_{acts_i} is contained in the BT, i.e., there exists an index $j \neq i$ such that $\mathcal{T}_{acts_i} = \mathcal{T}_{acts_j}$, this means there is already a situation identified where \mathcal{T}_{acts_i} is the appropriate response. Then, to reduce the number of nodes in the BT, we generalize the two special cases where this response is needed. This is done by combining \mathcal{T}_{cond_i} with \mathcal{T}_{cond_j} , that is, we find the BT, $\mathcal{T}_{cond_{ij}}$, that returns success if and only if \mathcal{T}_{cond_i} or \mathcal{T}_{cond_j} return success. Formally, this can be written as

$$\mathcal{T}_i \triangleq \mathcal{T}_{i-1}.\text{replace}(\mathcal{T}_{cond_i}, \text{simplify}((\mathcal{T}_{cond_i}, \mathcal{T}_{cond_j}))) \quad (8.7)$$

Figure 8.4 shows an example of this simplifying procedure. As can be seen this simplification generalizes the policy by iteratively removing conditions that are not relevant for the application of the specific action. It is thus central for keeping the number of nodes low, as seen below in Fig. 8.8.

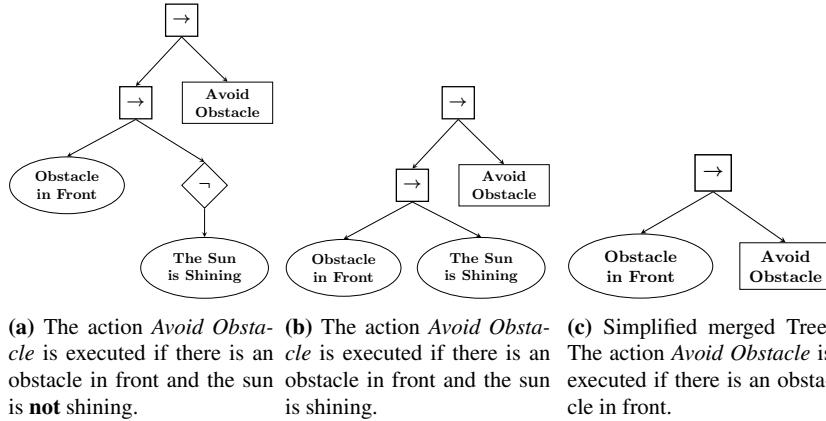


Fig. 8.4: Example of the simplifying procedure in (8.7). The two learned rules (a) and (b) are combined into (c): The important condition appears to be *Obstacle in Front*, and there is no reason to check the condition *The Sun is Shining*. These simplifications generalize the policies, and keep the BT sizes down.

Given these definitions, we can go through the steps listed in Algorithm 11. Note that the agent runs \mathcal{T}_i until a new situation is encountered which requires learning an expanded BT, or the goal is reached.

The BT \mathcal{T} is first initialized to be a single action, *Action Learn*, which will be used to trigger the learning algorithm. Running Algorithm 11 we execute the

Sequence composition of the safe subtree \mathcal{T}_{safe} (generated manually, or using a non-learning approach) with the current tree \mathcal{T} (Algorithm 11 Line 3). The execution of \mathcal{T}_{safe} overrides the execution of \mathcal{T} when needed to guarantee safety. If the action *Action Learn* is executed, it means that the current situation is not considered in neither \mathcal{T}_{safe} nor \mathcal{T} , hence a new action, or action composition, must be learned. The framework first starts with the greedy approach (Algorithm 11, Line 6) where it tries each action and stores the ones that increase the reward function, if no such actions are available (i.e. the reward value is a local maximum), then the framework starts learning the BT composition using the GP component (Algorithm 11, Line 8). Once the tree \mathcal{T}_{acts} is computed, by either the greedy or the GP component, the algorithm checks if \mathcal{T}_{acts} is already present in the BT as a response to another situation, and the new tree \mathcal{T} can be simplified using a generalization of the two situations (Algorithm 11, Line 11). Otherwise, the new tree \mathcal{T} is composed by the selector composition of the old \mathcal{T} with the new tree learned (Algorithm 11, Line 13). The algorithm runs until the goal is reached. The algorithm is guaranteed to lead the agent to the goal, under reasonable assumptions [14].

8.2.2 *The Algorithm Steps in Detail*

We now discuss Algorithm 11 in detail.

8.2.2.1 *GetSituation (Line 5)*

This function returns the tree \mathcal{T}_{cond} which represents the current situation. \mathcal{T}_{cond} is computed according to Equation (8.3).

8.2.2.2 *LearnSingleAction (Line 6)*

This function returns the tree \mathcal{T}_{acts} which represent the action to execute whenever the situation described by \mathcal{T}_{cond} holds. \mathcal{T}_{acts} is a Fallback composition with memory of all the actions that, if performed when \mathcal{T}_{cond} holds, increases the reward. The function *LearnSingleAction* runs the same episode N_a (number of actions) times executing a different action whenever \mathcal{T}_{cond} holds. When trying a new action, if the resulting reward increases, this action is stored. All the actions that lead to an increased reward are collected in a Fallback composition, ordered by the reward value. This Fallback composition, if any, is then returned to Algorithm 11.

8.2.2.3 LearnActionsUsingGP (Line 8)

If $LearnSingleAction$ has no children (Algorithm 11, Line 7) then there exists no single action that can increase the reward value when the situation described by \mathcal{T}_{cond} holds. In that case the algorithm learns a BT composition of actions and conditions that must be executed whenever \mathcal{T}_{cond} holds. This composition is derived as described in Section 8.1.

8.2.2.4 Simplify (Line 11)

If the resulting \mathcal{T}_{acts} is present in \mathcal{T} (Algorithm 11, Line 9) this means that there exist another situation \mathcal{S}_{exist} described by the BT $\mathcal{T}_{cond_{exist}}$, where the response in \mathcal{T}_{acts} is appropriate. To reduce the number of nodes in the updated tree, we create a new tree that captures both situations \mathcal{S} and \mathcal{S}_{exist} . This procedure removes from $\mathcal{T}_{cond_{exist}}$ a single condition c that is present in C_F for one situation (\mathcal{S} or \mathcal{S}_{exist}) and C_S for the other situation.

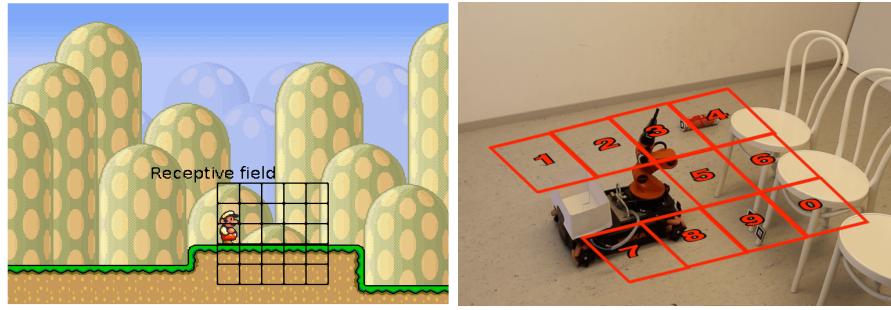
Remark 8.2. Note that the GP component is invoked exclusively whenever the greedy component fails to find a single action.

8.2.3 Pruning of Ineffective Subtrees

Once obtained the BT that satisfies the goal, we can search for ineffective subtrees, i.e. those action compositions that are superfluous for reaching the goal. To identify the redundant or unnecessary subtrees, we enumerate the subtrees with a Breadth-first enumeration. We run the BT without the first subtree and checking whether the reward function has a lower value or not. In the former case the subtree is kept, in the latter case the subtree is removed creating a new BT without the subtree mentioned. Then we run the same procedure on the new BT. The procedure stops when there are no ineffective subtree found. This procedure is optional.

8.2.4 Experimental Results

In this section we apply GP-BT to two problems. One is the *Mario AI benchmark* (Figure 8.5a) and one is a real robot, the *KUKA Youbot* (Figure 8.5b). The results on the Mario AI benchmark shows the applicability of the GP-BT on a highly complex dynamic environment and also allows us to compare the approach to the state-of-the-art. The results on the KUKA Youbot shows the applicability of GP-BT on a real robot.



(a) Mario AI benchmark.

(b) KUKA Youbot benchmark

Fig. 8.5: Benchmarks used to validate the framework.

8.2.4.1 Mario AI

The Mario AI benchmark [34] is an open-source software clone of Nintendo's Super Mario Bros used to test learning algorithms and game AI techniques. The task consists of moving the controlled character, Mario, through two-dimensional levels, which are viewed from the side. Mario can walk and run to the right and left, jump, and (depending on the mode, explained below) shoot fireballs. Gravity acts on Mario, making it necessary to jump over gaps to get past them. Mario can be in one of three modes: *Small*, *Big*, and *Fire* (can shoot fireballs). The main goal of each level is to get to the end of the level, which means traversing it from left to right. Auxiliary goals include collecting as many coins as possible, finishing the level as fast as possible, and collecting the highest score, which in part depends on the number of enemies killed. Gaps and moving enemies make the task more complex. If Mario falls down a gap, he loses a life. If he runs into an enemy, he gets hurt; this means losing a life if he is currently in the *Small* mode. Otherwise, his mode degrades from *Fire* to *Big* or from *Big* to *Small*.

Actions In the benchmark there are five actions available: *Walk Right*, *Walk Left*, *Crouch*, *Shoot*, and *Jump*.

Conditions In the benchmark there is a receptive field of observations as shown in Figure 8.5a. For each of the 25 cells in the grid there are 2 conditions available: *the box is occupied by an enemy* and *the box is occupied by an obstacle*. There are two other conditions: *Can Mario Shoot* and *Can Mario Jump*, creating a total of 52 conditions.

Reward Functions The reward function is given by a non linear function of the distance passed, enemies killed, number of times Mario is hurt, and time left when the end of the level is reached. The reward function is the same for every scenario.

Cross-Validation To evaluate the learned BT in an episode, we run a different episode of the same complexity (in terms of type of enemies; height of obstacles and length of gaps). In the Mario AI framework, this is possible by choosing different

so-called *seeds* for the learning episode and the validating episode. The result shown below are cross-validated in this way.

GP Parameters Whenever the GP part is invoked, it starts with 4 random BTs composed by one random control flow node and 2 random leaf nodes. The number of individuals in a generation is set to 25.

Scenarios We ran the algorithm in five different scenarios of increasing difficulty. The first scenario had no enemies and no gaps, thus only requiring motion to the right and jumping at the proper places. The resulting BT can be seen in Figure 8.7a where the action *Jump* is executed if an obstacle is in front of Mario and the action *Go Right* is executed otherwise. The second scenario has no obstacles but it has gaps. The resulting BT can be seen in Figure 8.7b where the action *Jump* is executed if Mario is close to a gap. The third scenario has high obstacles, gaps and walking enemies. The resulting BT can be seen in Figure 8.7c which is similar to a combination of the previous BTs with the addition of the action *Shoot* executed as soon as Mario sees an enemy (cell 14), and to *Jump* higher obstacles Mario cannot be too close. Note that to be able to show the BTs in a limited space, we used the *Pruning* procedure mentioned in Section 8.2.3. A video is available that shows the performance of the algorithm in all 5 scenarios.¹

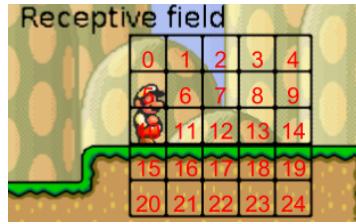


Fig. 8.6: Receptive field with cells' numbers.

We compared the performance of the GP-BT approach to a FSM-based algorithm of the type described in [21] and to a pure GP-based algorithm of the type described in [66].

For all three algorithms, we measure the performance, in terms of the reward function, and the complexity of the learnt solution in terms of the number of nodes in the BT/FSM respectively. The data for the simplest and most complex scenario, 1 and 5, can be found in Figure 8.8. As can be seen in Figure 8.8, the FSM approach generates a number of nodes that increases exponentially over time, while the growth rate in GP-BT tends to decrease over time. We believe that this is due to the simplification step, described in Equation (8.7), where two different situations requiring the same set of actions are generalized by a merge in the BT. The growth of the pure GP algorithm is very slow, as each iteration needs very long time to find a candidate improving the reward. This is due to the fact that the number of

¹ <https://youtu.be/Q00VtUYkoNQ>

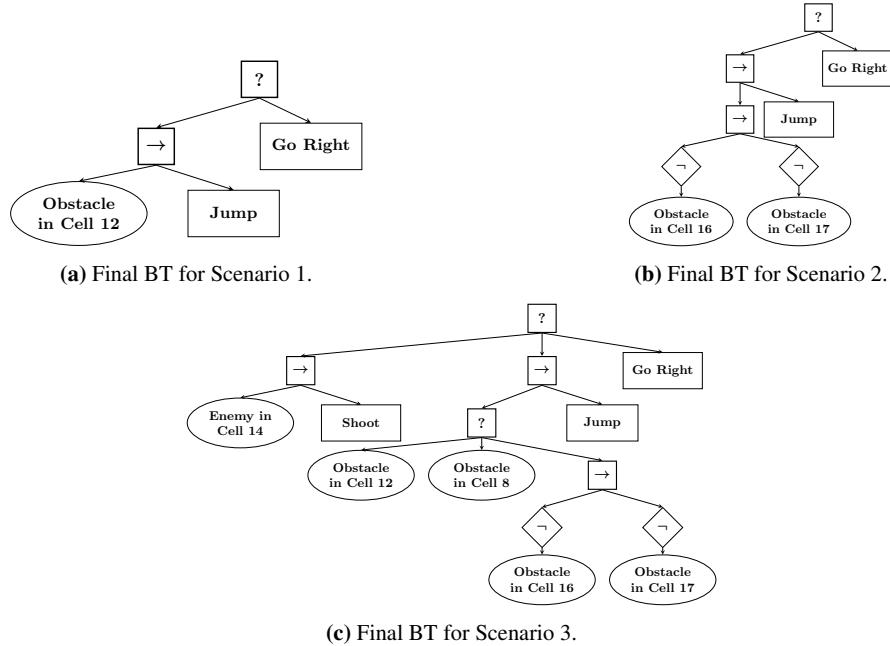


Fig. 8.7: Final BTs learned for Scenario 1-3.

conditions is larger than the number of actions, hence the pure GP approach often constructs BTs that check a large amount of conditions, while performing very few actions, or even none. Without a greedy component and the AND-OR-tree generalization with the conditions, a pure GP approach, like the one in [66], is having difficulties without any a-priory information. Looking at the performance in Figure 8.8 we see that GP-BT is the only one who reaches a reward of 1 (i.e. the task is completed) within the given execution time, not only for Scenario 5, but also for the less complex Scenario 1.

Remark 8.3. Note that we do not compare GP-BT with the ones of the Mario AI challenge, as we study problems with no a-priori information or model, whereas the challenge provided full information about the task and environment. When the GP-BT learning procedure starts, the agent (Mario) does not even know that the enemies should be killed or avoided.

The other scenarios have similar result. We chose to depict the simplest and the most complex ones.

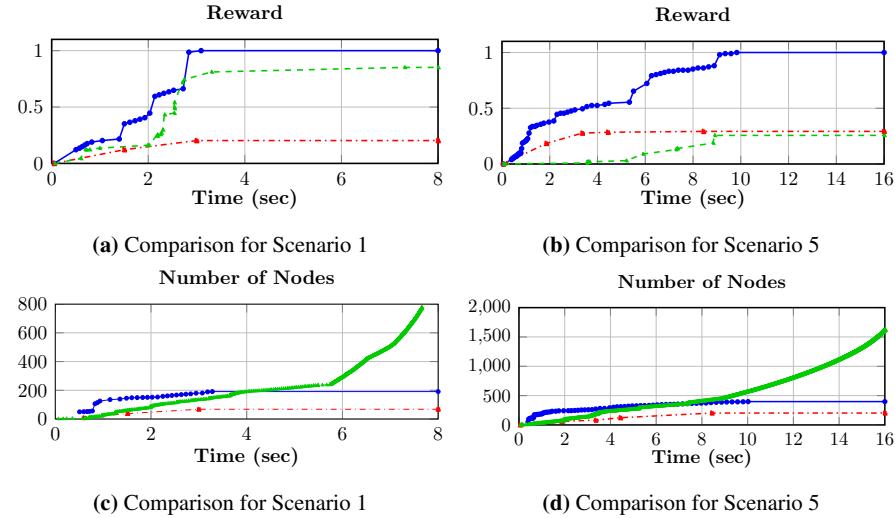


Fig. 8.8: Reward value comparison (a and b) and nodes number comparison (c and d). The blue solid line refers to GP-BT. The red dash-dotted line refers to the pure GP-based algorithm. The green dashed line refers to the FSM-based algorithm.

8.2.4.2 KUKA Youbot

As mentioned above, we use the KUKA Youbot to verify GP-BT on a real scenario. We consider three scenarios, one with a partially known environment and two with completely unknown environments.

Consider the Youbot in Fig. 8.5b, the conditions are given in terms of the 10 receptive fields and binary conditions regarding a number of different objects, e.g. larger or smaller obstacles. The corresponding actions are: go left/right/forward, push object, pick object up etc. Again, the problem is to learn a switching policy mapping conditions to actions.

Setup The robot is equipped with a wide range HD camera and uses markers to recognize the objects nearby. The recognized objects are mapped into the robot simulation environment V-REP [18]. The learning procedure is first tested on the simulation environment and then executed on the real robot.

Actions Move Forward, Move Left, Move Right, Fetch Object, Slide Object to the Side, Push Object.

Conditions Wall on the Left, Wall on the Right, Glass in Front, Glass on the Left, Glass on the Right, Cylinder in Front, Cylinder on the Left, Cylinder on the Right, Ball in Front, Ball on the Left, Ball on the Right, Big Object in Front, Big Object on the Left, Big Object on the Right.

Scenarios In the first scenario, the robot has to traverse a corridor dealing with different objects that are encountered on the way. The destination and the position of the walls is known a priori for simplicity. The other objects are recognized and

mapped once they enter the field of view of the camera. The second scenario illustrates the reason why GP-BT performs the learning procedure for each different situation. The same type of cylinder is dealt with differently in two different situations.

In the third scenario, a single action is not sufficient to increase the reward. The robot has to learn an action composition using GP to reach the goal.

A YouTube video is available that shows all three scenarios in detail².

8.2.5 Other Approaches using GP applied to BTs

There exists several other approaches using GP applied to BTs.

Grammatical Evolution (GE), a grammar-based form of GP, that specifies the syntax of possible solutions using a context-free grammar was used to synthesize a BT [58]. The root node consists of a Fallback node, with a variable number of subtrees called *BehaviourBlocks*. Each BehaviourBlock consists of a sequence of one or more conditions created through a GE algorithm, followed by a sequence of actions (possibly with some custom made Decorator) that are also created through a GE algorithm. The root node has as right most child a BehaviourBlocks called *DefaultSequence*: a sequence with memory with only actions, again created through a GE algorithm. The approach works as follows: When the BT is executed, the root node will route ticks to one BehaviourBlock; if none of those BehaviourBlocks execute actions, the ticks reach the DefaultSequence. The approach was used to compete at the 2010 Mario AI competition, reaching the fourth place of the gameplay track.

Another similar approach was used to synthesize sensory-motor coordinations of a micro air vehicle using a pure GP algorithm on a initial population of randomly generated BTs [66]. The mutation operation is implemented using two methods: *micro mutation* and *macro mutation*. Micro mutation affects only leaf nodes and it is used to modify the parameter of a node. Macro mutation was used to replace a randomly selected node with a randomly selected tree.

8.3 Reinforcement Learning applied to BTs

In this section we will describe an approach proposed in [57] for combining BTs with Reinforcement Learning (RL).

² <https://youtu.be/P9JRC9wTmIE> and <https://youtu.be/dLVQ01KSqGU>

8.3.1 Summary of *Q-Learning*

A general class of problems that can be addressed by RL is the Markov Decision Processes (MDPs). Let $s_t \in S$ be the system state at time step t , $a_t \in A_{s_t}$ the action executed at time t , where A_s is a finite set of admissible actions for the state s (i.e. the actions that can be performed by the agent while in s). The goal of the agent is to learn a policy $\pi : S \rightarrow A$ (i.e. a map from state to action), where $A = \bigcup_{s \in S} A_s$, that maximizes the expected long-term reward from each state s , [70].

One of the most used Reinforcement Learning techniques is *Q-Learning*, which can handle problems with stochastic transitions and rewards. It uses a function called *Q-function* $Q : S \times A \rightarrow \mathbb{R}$ to capture the expected future reward of each state-action combination (i.e. how good it is to perform a certain action when the agent is at a certain state). First, for each state-action pair, Q is initialized to an arbitrary fixed value. Then, at each time step of the algorithm, the agent selects an action and observes a reward and a new state. Q is updated according to a *simple value iteration* function [44], using the weighted average of the old value and a value based on the new information, as follows:

$$Q_{k+1}(s, a) = (1 - \alpha_k)Q_k(s, a) + \alpha_k \left[r + \gamma \max_{a' \in A_{s'}} Q_k(s', a') \right], \quad (8.8)$$

where k is the iteration index (increasing every time the agent receives an update), r is the reward, γ is the discount factor that trades off the influence of early versus late rewards, and α_k is the learning rate that trades off the influence of newly acquired information versus old information.

The algorithm converges to an optimal policy that maximizes the reward if all admissible state-action pairs are updated infinitely often [69, 3]. At each state, the optimal policy selects the action that maximizes the Q value.

Hierarchical Reinforcement Learning

The vast majority of RL algorithms are based upon the MDP model above. However, Hierarchical RL (HRL) algorithms have their basis in Semi MDPs (SMDPs) [3]. SMDPs enable a special treatment of the temporal aspects of the problem and, thanks to their hierarchical structure, reduce the impact of the curse of dimensionality by dividing the main problem into several subproblems.

Finally, the *option* framework [70] is a SMDP-based HRL approach used to efficiently compute the Q -function. In this approach, the options are a generalization of actions, that can call other options upon execution in a hierarchical fashion until a *primitive option* (an action executable in the current state) is found.

8.3.2 The RL-BT Approach

In this section, we outline the approach proposed in [57], that we choose to denote RL-BT. In order to combine the advantages of BTs and RL, the RL-BT starts from a

manually designed BT where a subset of nodes are replaced with so-called *Learning Nodes*, which can be both actions and flow control nodes.

In the *Learning Action node*, the node encapsulates a Q-learning algorithm, with states s , actions A , and reward r defined by the user. For example, imagine a robot with an Action node that need to “pick an object”. This learning Action node uses Q-learning to learn how to grasp an object in different positions. Then, the state is defined as the pose of the object with respect to the robot’s frame, the actions as the different grasp poses, and the reward function as the grasp quality.

In the *Learning Control Flow Node*, the node is a standard Fallback with flexible node ordering, quite similar to the ideas described in Chapter 4. Here, the designer chooses the representation for the state s , while the actions in A are the children of the Fallback node, which can be learning nodes or regular nodes. Hence, given a state s , the Learning Control Flow Node selects the order of its own children based on the reward. The reward function can be task-depended (i.e. the user finds a measure related to the specific task) or return value-dependent (i.e., it gives a positive reward for Success and negative reward for Failure). For example, consider a NPC that has three main goals: find resources, hide from stronger players, and attack weaker ones. The Learning Control Flow node has 3 subtrees, one for each goal. Hence the state s can be a vector collecting information about players’ position, weapons position, health level, etc. The reward function can be a combination of health level, score, etc.

In [57], a formal definition of the learning node is made, with analogies to the Options approach for HRL to compute the Q -function, and connections between BTs and the recursive nature of the approach.

8.3.3 Experimental Results

In this section, we briefly describe the results of two experiments from [57]. Both experiments execute the approach in 30 different episodes, each with 400 iterations. At the beginning of each experiment, the initial state is reset. The environment is composed of a set of rooms. In each room, the agent can perform 3 actions: *save victim*, *use extinguisher X*, and *change room*. Each room has a probability of 0.5 to have a victim; if there is a victim, the agent must save it first. Moreover, each room has a probability of 0.5 to have one of 3 types of fire (1, 2 and 3, with probability 1/3 each); if there is a fire, the agent must extinguish it before leaving the room. The agent is equipped with 3 extinguishers (types A , B and C), and each extinguisher can successfully extinguish only one type of fire, randomly chosen at the beginning of the episode and unknown to the agent.

Scenario 1

In this scenario, we evaluate the capability of *Use Extinguisher X* to learn the correct extinguisher to use for a specific type of fire.

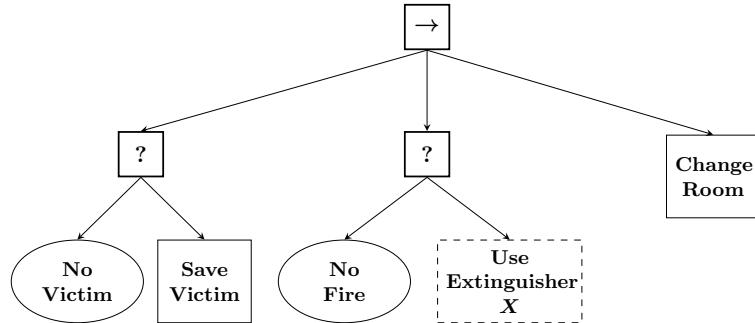


Fig. 8.9: The BT model for the first agent. Only a single Action node (*Use Extinguisher*) has learning capabilities.

Figure 8.9 depicts the BT modeling the behavior of the learning agent. In the learning Action node *Use Extinguisher X* the state is defined as $s = \langle \text{fire type} \rangle$, where $\text{fire type} = \{1, 2, 3\}$, and the available actions are as follows:

$A = \{\text{Use Extinguisher A}, \text{Use Extinguisher B}, \text{Use Extinguisher C}\}$. The reward is defined as 10 if the extinguisher can put out the fire and -10 otherwise. The results in [57] show that the accuracy converges to 100%.

Scenario 2

This scenario is a more complex version of the one above, where we consider the time spent to execute an action.

The actions *Save Victim* and *Use Extinguisher X* now take time to complete, depending on the fire intensity. Any given fire has an intensity $\text{fire intensity} \in \{1, 2, 3\}$, chosen randomly for each room. The fire intensity specifies the time steps needed to extinguish a fire. The fire is extinguished when its intensity is reduced to 0.

The fire intensity is reduced by 1 each time the agent uses the correct extinguisher. The action *change room* is still executed instantly and the use of the wrong extinguisher makes the agent lose the room.

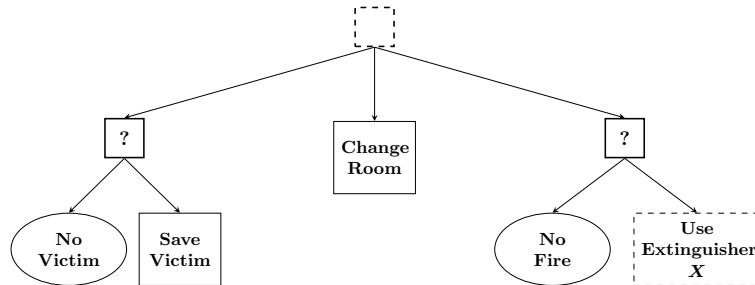


Fig. 8.10: The BT model for the second agent using two nested Learning Nodes.

Figure 8.10 shows the BT modeling the learning agent. It uses 2 learning nodes. The first, similar to the one used in Scenario 1, is a learning Action node using. In that node, the state is defined as $s = \langle \text{fire type} \rangle$, where $\text{fire type} = \{1, 2, 3\}$, the action set as $a = \{A, B, C\}$, and the reward as $\frac{10}{\text{fire intensity}}$ if the extinguisher can extinguish the fire and -10 otherwise.

The second learning node is a learning control flow node. It learns the behavior that must be executed given the state $s = \langle \text{has victim?}, \text{has fire?} \rangle$. The node's children are the actions $A = \{\text{save victim}, \text{use extinguisher } X, \text{change room}\}$. This learning node receives a cumulative reward, -10 if the node tries to save and there is no victim, -1 while saving the victim, and $+10$ when the victim is saved; -10 if trying to extinguish a non-existing fire, -1 while extinguishing it, and $+10$ if the fire is extinguished; $+10$ when the agent leaves the room at the right moment and -10 otherwise.

The results in [57] show that the accuracy converges to 97-99%. The deviation from 100% is due to the fact that the learning control flow node needs some steps of trial-and-error to learn the most effective action order.

8.4 Comparison between GP-BT and RL-BT

How do we choose between GP-BT and RL-BT for a given learning problem?

GP-BT and RL-BT operate on the same basic premise: they both try something, receive a reward depending on “how good” the result is, and then try and do something different to get a higher reward.

The key difference is that GP-BT operates on the BT itself, creating the BT from scratch and extending it when needed. RL-BT on the other hand starts with a fixed BT structure that is not changed by the learning. Instead, the behaviors of a set of designated nodes are improved. If RL-BT was started from scratch with a BT consisting of a single action, there would be no difference between RL-BT and standard RL. Instead, the point of combining BTs with RL is to address the curse of dimensionality, and do RL on smaller state spaces, while the BT connects these subproblems in a modular way, with the benefits described in this book. RL-BT thus needs user input regarding both the BT structure, and the actions, states and local reward functions to be considered in the subproblems addressed by RL. GP-BT on the other hand only needs user input regarding the single global reward function, and the conditions (sensing) and actions available to the agent.

8.5 Learning from Demonstration applied to BTs

Programming by demonstration has a straightforward application in both robotics and the development of the AI for NPCs in games.

In robotics, the Intera5 software for the Baxter and Sawyer robots provides learning by demonstration support³.

In computer games, one can imagine a game designer controlling the NPC during a training session in the game, demonstrating the expected behavior for that character in different situations. One such approach called *Trained BTs (TBTs)* was proposed in [64].

TBT applies the following approach: it first records traces of actions executed by the designer controlling the NPC to be programmed, then it processes the traces to generate a BT that can control the same NPC in the game. The resulting BT can then be further edited with a visual editor. The process starts with the creation of a minimal BT using the provided BT editor. This initial BT contains a special node called a Trainer Node (TN). Data for this node are collected in a game session where the designer simulates the intended behavior of the NPC. Then, the trainer node is replaced by a machine-generated sub-behavior that, when reached, selects the task that best fits the actual state of the game and the traces stored during the training session. The approach combines the advantages of programming by demonstration with the ability to fine-tune the learned BT.

Unfortunately, using learning from demonstration approaches the learned BT easily becomes very large, as each trace is directly mapped into a new sub-BT. Recent approaches address this problem by finding common patterns in the BT and generalizing them [62]. First, it creates a maximally specific BT from the given traces. Then iteratively reduces the BT in size by finding and combining common patterns of actions. The process stops when the BT has no common patters. Reducing the size of the BT also improves readability.

³ http://mfg.rethinkrobotics.com/intera/Building_a_Behavior_Tree_Using_the_Robot_Screen

Chapter 9

Stochastic Behavior Trees

In this chapter, we study the reliability of reactive plan executions, in terms of execution times and success probabilities. To clarify what we mean by these concepts, we consider the following minimalistic example: a robot is searching for an object, and can choose between the two subtasks *searching on the table*, and *opening/searching the drawer*. One possible plan is depicted in Figure 9.1. Here, the robot first searches the table and then, if the object was not found on the table, opens the drawer and searches it. In the figure, each task has an execution time and a success probability. For example, searching the table has a success probability of 0.1 and an execution time of 5s. Given a plan like this, it is fairly straightforward to compute the reliability of the entire plan, in terms of execution time distribution and success probability. In this chapter, we show how to compute such performance metrics for arbitrary complex plans encoded using BTs. In particular, we will define Stochastic BTs in Section 9.1, transform them into Discrete Time Markov Chains (DTMCs) in Section 9.2, compute reliabilities in Section 9.3 and describe examples Section 9.4. Some of the results of this chapter were previously published in the paper [11].

Before motivating our study of BTs we will make a few more observations regarding the example above. The ordering of the children of Fallback nodes (*searching on the table* and *opening/searching the drawer*) can in general be changed, whereas the ordering of the children of Sequence nodes (*opening the drawer* and *searching the drawer*) cannot. Note also that adding subtasks to a Sequence generally decreases overall success probabilities, whereas adding Fallbacks generally increases overall success probabilities, as described in Section 4.2.

9.1 Stochastic BTs

In this section we will show how some probabilistic measures, such as Mean Time to Succeed (MTTS), Mean Time to Fail (MTTF), and probabilities over time carry across modular compositions of BTs. The advantage of using BTs lie in their modularity and hierarchical structure, which provides good scalability, as explained

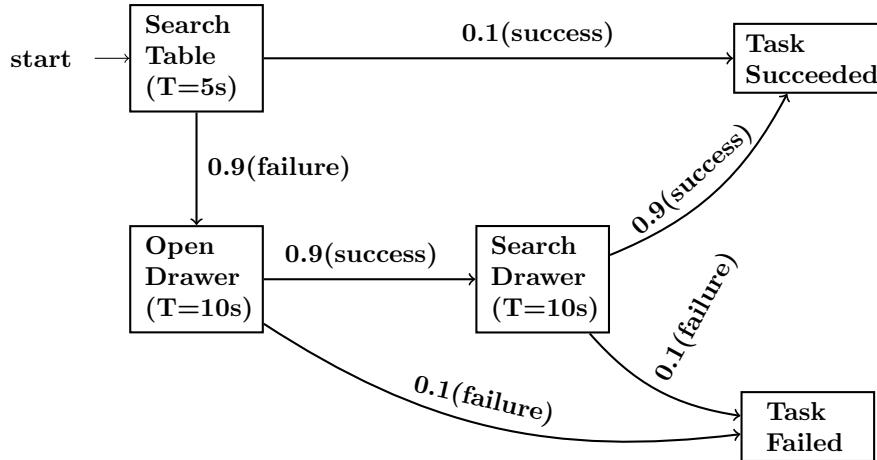


Fig. 9.1: A simple plan for a search task, modelled by a Markov Chain.

above. To address the questions above, we need to introduce some concepts from Markov theory.

9.1.1 Markov Chains and Markov Processes

Markov theory [52] deals with memoryless processes. If a process is given by a sequence of actions that changes the system's state disregarding its history, a DTMC is suitable to model the plan execution. Whereas if a process is given by a transition rates between states, a Continuous Time Markov Chain (CTMC) it then suitable to model such plan execution. A DTMC is given by a collection of states $\mathcal{S} = \{s_1, s_2, \dots, s_d\}$ and the transitions probabilities p_{ij} between states s_i and s_j . A CTMC is given by a collection of states \mathcal{S} and the transition rates q_{ij}^{-1} between states s_i and s_j .

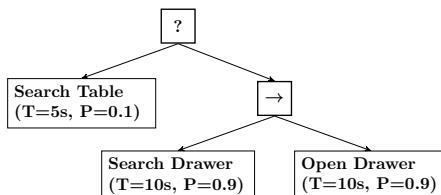


Fig. 9.2: The BT equivalent of the Markov chain in Figure 9.1. The atomic actions are the leaves of the tree, while the interior nodes correspond to *Sequence* compositions (arrow) and *Fallbacks* compositions (question mark).

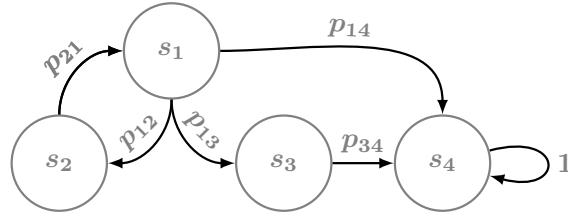


Fig. 9.3: Example of a DTMC with 4 states and 6 transitions.

Definition 9.1. The stochastic sequence $\{X_n, n = 0, 1, 2, \dots\}$ is a DTMC provided that:

$$\begin{aligned} P\{X_{n+1} = s_{n+1} | X_n = s_n, X_{n-1} = s_{n-1}, \dots, X_0 = s_0\} &= \\ &= P\{X_{n+1} = s_{n+1} | X_n = s_n\} \end{aligned} \quad (9.1)$$

$\forall n \in \mathbb{N}$, and $\forall s \in \mathcal{S}$

The expression on the right hand side of (9.1) is the so-called *one step transition probability* of the chain and denotes the probability that the process goes from state s_n to state s_{n+1} . We use the following notation:

$$p_{ij} = P\{X_{n+1} = s_j | X_n = s_i\} \quad (9.2)$$

to denote the probability to jump from a state s_i to a state s_j . Since we only consider homogeneous DTMC, the above probabilities do not change in time.

Definition 9.2. The *one-step transition matrix* P is a $|\mathcal{S}| \times |\mathcal{S}|$ matrix in which the entries are the transition probabilities p_{ij} .

Let $\pi(k) = [\pi_1(k), \dots, \pi_{|\mathcal{S}|}(k)]^\top$, where π_i is the probability of being in state i , then the Markov process can be described as a discrete time system with the following time evolution:

$$\begin{cases} \pi(k+1) = P^\top \pi(k) \\ \pi(0) = \pi_0. \end{cases} \quad (9.3)$$

where π_0 is assumed to be known a priori.

Definition 9.3. The stochastic sequence $\{X(t), t \geq 0\}$ is a CTMC provided that:

$$\begin{aligned} P\{X(t_{n+1}) = s_{n+1} | X(t_n) = s_n, X(t_{n-1}) = s_{n-1}, \dots, \\ , X(t_0) = s_0\} = P\{X(t_{n+1}) = s_{n+1} | X(t_n) = s_n\} \end{aligned} \quad (9.4)$$

$\forall n \in \mathbb{N}$, $\forall s \in \mathcal{S}$, and all sequences $\{t_0, t_1, \dots, t_{n+1}\}$ such that $t_0 < t_1 < \dots < t_n < t_{n+1}$. We use the following notation:

$$p_{ij}(\tau) = P\{X(t + \tau) = s_j | X(\tau) = s_i\} \quad (9.5)$$

to denote the probability to be in a state s_j after a time interval of length τ given that at present time is into a state s_i . Since we only consider homogeneous CTMC, the above probabilities only depend on the time length τ .

To study the continuous time behavior of a Markov process we define the so-called *infinitesimal generator matrix* Q .

Definition 9.4. The infinitesimal generator of the transition probability matrix $P(t)$ is given by:

$$Q = [q_{ij}] \quad (9.6)$$

where

$$q_{ij} = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(\Delta t)}{\Delta t} & \text{if } i \neq j \\ -\sum_{k \neq i} q_{kj} & \text{otherwise.} \end{cases} \quad (9.7)$$

Then, the continuous time behavior of the Markov process is described by the following ordinary differential equation, known as the Cauchy problem:

$$\begin{cases} \dot{\pi}(t) = Q^\top \pi(t) \\ \pi(0) = \pi_0 \end{cases} \quad (9.8)$$

where the initial probability vector π_0 is assumed to be known a priori.

Definition 9.5. The average sojourn time SJ_i of a state s_i in a CTMC is the average time spent in that state. It is given by [68]:

$$SJ_i = -\frac{1}{q_{ii}} \quad (9.9)$$

Definition 9.6. Considering the CTMC $\{X(t), t \geq 0\}$, the stochastic sequence $\{Y_n, n = 0, 1, 2, \dots\}$ is a DTMC and it is called Embedded MC (EMC) of the process $X(t)$ [68].

The transition probabilities of the EMC r_{ij} are defined as:

$$r_{ij} = P\{Y_{n+1} = s_j | Y_n = s_i\} \quad (9.10)$$

and they can be easily obtained as a function of the transition rates q_{ij} :

$$r_{ij} = \begin{cases} -\frac{q_{ij}}{q_{ii}} & \text{if } i \neq j \\ 1 - \sum_{k \neq i} r_{kj} & \text{otherwise.} \end{cases} \quad (9.11)$$

On the other hand, the infinitesimal generator matrix Q can be reconstructed from the EMC as follows

$$q_{ij} = \begin{cases} \frac{1}{Sj} r_{ij} & \text{if } i \neq j \\ -\sum_{k \neq i} r_{kj} & \text{otherwise} \end{cases}. \quad (9.12)$$

9.1.2 Formulation

We are now ready to make some definitions and assumptions, needed to compute the performance estimates.

Definition 9.7. An action \mathcal{A} in a BT, is called *stochastic* if the following holds:

- It first returns Running, for an amount of time that might be zero or non-zero, then consistently returns either Success or Failure for the rest of the execution of its parent node.¹
- The probability to succeed p_s and the probability to fail p_f are known a priori.
- The probability to succeed $p_s(t)$ and the probability to fail $p_f(t)$ are exponentially distributed with the following Probability Density Functions (PDFs):

$$\hat{p}_s(t) = p_s \mu e^{-\mu t} \quad (9.13)$$

$$\hat{p}_f(t) = p_f \nu e^{-\nu t} \quad (9.14)$$

from which we can calculate the Cumulative Distribution Functions (CDFs)

$$\bar{p}_s(t) = p_s(1 - e^{-\mu t}) \quad (9.15)$$

$$\bar{p}_f(t) = p_f(1 - e^{-\nu t}) \quad (9.16)$$

Definition 9.8. An action \mathcal{A} in a BT, is called *deterministic* (in terms of execution time, not outcome) if the following holds:

- It first returns Running, for an amount of time that might be zero or non-zero, then consistently returns either Success or Failure for the rest of the execution of its parent node.
- The probability to succeed p_s and the probability to fail p_f are known a priori.
- The time to *succeed* and the time to *fail* are deterministic variables τ_s and τ_f known a priori.
- The probability to succeed $p_s(t)$ and the probability to fail $p_f(t)$ have the following PDFs:

$$\hat{p}_s(t) = p_s \delta(t - \tau_s) \quad (9.17)$$

$$\hat{p}_f(t) = p_f \delta(t - \tau_f) \quad (9.18)$$

¹ The execution of the parent node starts when it receives a tick and finishes when it returns either Success/Failure to its parent.

where $\delta(\cdot)$ is the Dirac's delta function. From the PDFs we can calculate the CDFs:

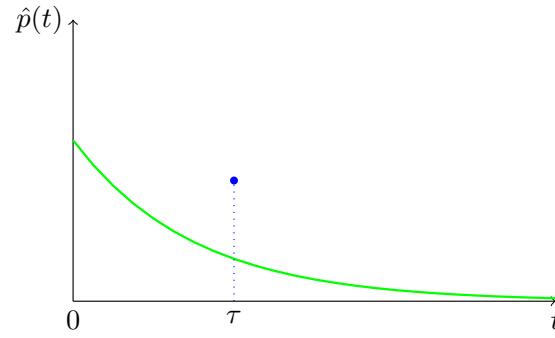
$$\bar{p}_s(t) = p_s H(t - \tau_s) \quad (9.19)$$

$$\bar{p}_f(t) = p_f H(t - \tau_f) \quad (9.20)$$

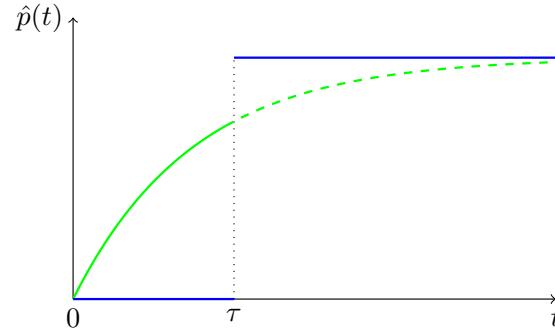
where $H(\cdot)$ is the step function.

Remark 9.1. Note that it makes sense to sometimes have $\tau_s \neq \tau_f$. Imagine a door opening task which takes 10s to complete successfully but fails 30% of the time after 5s when the critical grasp phase fails.

Example 9.1. For comparison, given a deterministic action with τ_s , we let the rates of a stochastic action have $\mu = \tau_s^{-1}$. Then the PDFs and CDFs are as seen in Figure 9.4.



(a) PDFs.



(b) CDFs.

Fig. 9.4: Cumulative and probability density distribution function for a deterministic (dark straight lines) and stochastic action (bright curves).

As we want to analyze the BT composition of actions, we must also define actions that include both stochastic and deterministic parts.

Definition 9.9. An action \mathcal{A} in a BT, is called *hybrid* if one of $p_s(t)$ and $p_f(t)$ is a random variable with exponential distribution, and the other one is deterministic.

Thus hybrid actions come in two different variations: **Deterministic Success Time** For this type of hybrid action, the following holds:

- It first returns Running, for an amount of time that might be zero or non-zero, then consistently returns either Success or Failure for the rest of the execution of its parent node.
- The probability to succeed p_s is known a priori.
- The time to *succeed* is a deterministic variable τ_s known a priori.
- The probability to fail has the following PDF:

$$\hat{p}_f(t) = \begin{cases} p_f(1 - e^{-vt}) & \text{if } t < \tau_s \\ 1 - p_s & \text{if } t = \tau_s \\ 0 & \text{otherwise} \end{cases} \quad (9.21)$$

In this case the CDF and the PDF of the probability to succeed are discontinuous. In fact this hybrid action will return Failure if, after the success time τ_s , it does not return Success. Then, to have an analogy with stochastic actions we derive the PDF of the probability to succeed:

$$\hat{p}_s(t) = p_s \delta(t - \tau_s) \quad (9.22)$$

and the CDFs as follows:

$$\bar{p}_s(t) = p_s H(t - \tau_s) \quad (9.23)$$

$$\bar{p}_f(t) = \begin{cases} p_f(1 - e^{-vt}) & \text{if } t < \tau_s \\ 1 - \bar{p}_s(t) & \text{otherwise} \end{cases} \quad (9.24)$$

Thus, the probability of Running is zero after τ_s i.e. after τ_s it either fails or succeeds. Moreover, the success rate is set to $\mu = \tau_s^{-1}$.

Deterministic Failure Time For this type of hybrid action, the following holds:

- It first returns Running, for an amount of time that might be zero or non-zero, then consistently returns either Success or Failure for the rest of the execution of its parent node.
- The probability to fail p_f is known a priori.
- The time to *succeed* is a random variables with exponential distribution with rate μ known a priori.
- The probability to succeed has the following PDF:

$$\hat{p}_s(t) = \begin{cases} p_s(1 - e^{-\mu t}) & \text{if } t < \tau_f \\ 1 - p_f & \text{if } t = \tau_f \\ 0 & \text{otherwise} \end{cases}. \quad (9.25)$$

To have an analogy with stochastic actions we derive the PDF of the probability to fail:

$$\hat{p}_f(t) = p_f \delta(t - \tau_f) \quad (9.26)$$

and the CDFs as follows:

$$\bar{p}_f(t) = p_f H(t - \tau_f) \quad (9.27)$$

$$\bar{p}_s(t) = \begin{cases} p_s(1 - e^{-\mu t}) & \text{if } t < \tau_f \\ 1 - \bar{p}_f(t) & \text{otherwise} \end{cases}. \quad (9.28)$$

Moreover, the failure rate is set to $\nu = \tau_f^{-1}$

Remark 9.2. Note that the addition of deterministic execution times makes (9.8) discontinuous on the right hand side, but it still has a unique solution in the Carathéodory sense [16].

We will now give an example of how these concepts transfer over BT compositions.

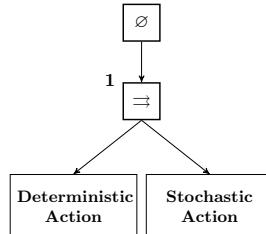


Fig. 9.5: Parallel node of Example 9.2.

Example 9.2. Consider the BT in Figure 9.5. The Parallel node is set to returns Success as soon as one child returns Success, and the two children are of different kinds, one deterministic and the other stochastic. Note that the MTTS and MTTF of this BT has to account for the heterogeneity of its children. The deterministic child can succeed only at time τ_s . The CDF of the Parallel node is given by the sum of the CDFs of its children. The PDF has a jump at time τ_s accounting for the fact that the Parallel node is more likely to return Success after that time. Thus, the PDF and the CDF of a Success return status are shown in Figure 9.6.

Definition 9.10. A BT \mathcal{T}_1 and a BT \mathcal{T}_2 are said *equivalent* if and only if \mathcal{T}_1 can be created from \mathcal{T}_2 by permutations of the children of Fallbacks and Parallel compositions.

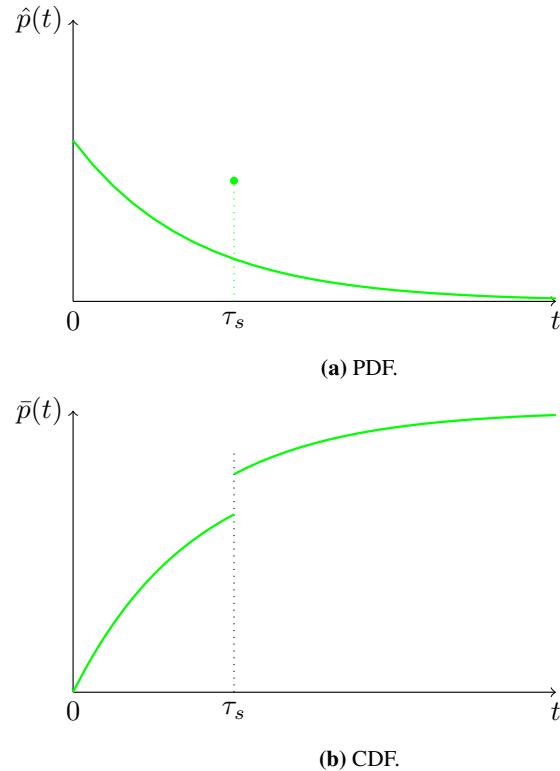


Fig. 9.6: Cumulative and probability density distribution function of Success of the Parallel node in Figure 9.5.

An example of two equivalent BTs is shown in Figure 9.7.

Assumption 9.1 *For each action \mathcal{A} in the BT, one of the following holds*

- *The action \mathcal{A} is a stochastic action.*
- *The action \mathcal{A} is a deterministic action.*
- *The action \mathcal{A} is a hybrid action.*

Assumption 9.2 *For each condition \mathcal{C} in the BT, the following holds*

- *It consistently returns the same value (Success or Failure) throughout the execution of its parent node.*
- *The probability to succeed at any given time $p_s(t)$ and the probability to fail at any given time $p_f(t)$ are known a priori.*

We are now ready to define a Stochastic BT (SBT).

Definition 9.11. A SBT is a BT satisfying Assumptions 9.1 and 9.2.

Given a SBT, we want to use the probabilistic descriptions of its actions and conditions, $p_s(t)$, $p_f(t)$, μ and ν , to recursively compute analogous descriptions for every subtrees and finally the whole tree.

To illustrate the investigated problems and SBTs we take a look at the following example.

Example 9.3. Imagine a robot that is to search for a set of keys on a table and in a drawer. The robot knows that the keys are often located in the drawer, so that location is more likely than the table. However, searching the table takes less time, since the drawer must be opened first. Two possible plans are conceivable: searching the table first, and then the drawer, as in Figure 9.7a, or the other way around as in Figure 9.7b. These two plans can be formulated as SBTs and analyzed through the scope of Problem 1 and 2, using the results of Section 9.1 below. Depending on the user requirements in terms of available time or desired reliability at a given time, the proper SBT can be chosen.

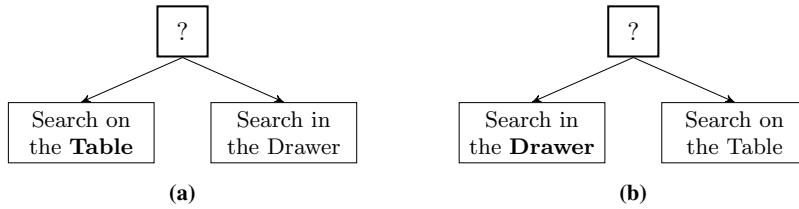


Fig. 9.7: BT modeling of two plan options. In (a), the robot searches on the table first, and in the drawer only if the table search fails. In (b), the table is searched only if nothing is found in the drawer.

Remark 9.3. Note that Assumption 9.1 corresponds to the return status of the search actions in Example 9.3 behaving in a reasonable way, e.g., not switching between Success and Failure.

9.2 Transforming a SBT into a DTMC

The first step is to define, for each control flow node in \mathcal{V} , a vector representation of the children's outcomes and a description of its execution policy, then we map the execution into a DTMC with a direct representation of the one-step transition matrix, and finally we compute the probability of success and failure over time for each node.

Note that the modularity of BTs comes from the recursive tree structure, any BT can be inserted as subtree in another BT. This modularity allows us to do the analysis

in a recursive fashion, beginning with the leaves of the BT, i.e. the actions and conditions which have known probabilistic parameters according to Assumptions 9.1 and 9.2, and then progressing upwards in a scalable fashion.

To keep track of the execution of a given flow control node, the children outcomes are collected in a vector state called the *marking* of the node, and the transitions between markings are defined according to the execution policy of the node. In detail, let $\mathbf{m}(k) = [m_1(k), m_2(k), \dots, m_N(k)]$ be a marking of a given BT node with N children at time step k with

$$m_i(k) = \begin{cases} -1 & \text{if child } i \text{ returns Failure at k} \\ 1 & \text{if child } i \text{ returns Success at k} \\ 0 & \text{otherwise} \end{cases} \quad (9.29)$$

Example 9.4. Consider the BT in Figure 9.7a. If the first child (Search Table) has failed, and the second (Search Drawer) is currently running, the marking would be $\mathbf{m}(k) = [-1, 0]$.

We define an *event* related to a BT node when one of its children returns either Success or Failure. Defining $\mathbf{e}_i(k)$ to be the vector associated to the event of the i -th running child, all zeros except the i -th entry which is equal to $e_i(k) \in \{-1, 1\}$:

$$\mathbf{e}_i(k) = \begin{cases} -1 & \text{if child } i \text{ has failed at k} \\ 1 & \text{if child } i \text{ has succeeded at k.} \end{cases} \quad (9.30)$$

We would like to describe the time evolution of the node marking due to an event associated with the child i as follows:

$$\mathbf{m}(k+1) = \mathbf{m}(k) + \mathbf{e}_i(k) \quad (9.31)$$

with the event $\mathbf{e}_i(k)$ restricted to the feasible set of events at $\mathbf{m}(k)$, i.e.

$$\mathbf{e}_i(k) \in \mathcal{F}(\mathbf{m}(k)).$$

In general, $\mathcal{F}(\mathbf{m}(k)) \subset \mathcal{F}_0$, with

$$\mathcal{F}_0 = \{\mathbf{e}_i : \mathbf{e}_i \in \{-1, 0, 1\}^N, \|\mathbf{e}_i\|_2 = 1\}, \quad (9.32)$$

i.e. events having only one nonzero element, with value -1 or 1 . We will now describe the set $\mathcal{F}(\mathbf{m}(k))$ for the three different node types.

Feasibility condition in the Fallback node

$$\begin{aligned}\mathcal{F}_{FB}(\mathbf{m}(k)) = \{\mathbf{e}_i \in \mathcal{F}_0 : \exists i : m_i(k) = 0, e_i \neq 0, \\ m_j(k) = -1, \forall j, 0 < j < i\},\end{aligned}\tag{9.33}$$

i.e. the event of a child returning Success or Failure is only allowed if it was ticked, which only happens if it is the first child, or if all children before it have returned Failure.

Feasibility condition in the Sequence node

$$\begin{aligned}\mathcal{F}_S(\mathbf{m}(k)) = \{\mathbf{e}_i \in \mathcal{F}_0 : \exists i : m_i(k) = 0, e_i \neq 0, \\ m_j(k) = 1, \forall j, 0 < j < i\},\end{aligned}\tag{9.34}$$

i.e. the event of a child returning Success or Failure is only allowed if it was ticked, which only happens if it is the first child, or if all children before it have returned Success.

Feasibility condition in the Parallel node

$$\begin{aligned}\mathcal{F}_P(\mathbf{m}(k)) = \{\mathbf{e}_i \in \mathcal{F}_0 : \exists i : m_i(k) = 0, e_i \neq 0, \\ \sum_{j:m_j(k)>0} m_j(k) < M \\ \sum_{j:m_j(k)<0} m_j(k) < N - M + 1\},\end{aligned}\tag{9.35}$$

i.e. the event of a child returning Success or Failure is only allowed if it has not returned yet, and the conditions for Success ($< M$ successful children) and Failure ($< N - M + 1$ failed children) of the Parallel node are not yet fulfilled.

Example 9.5. Continuing Example 9.4 above, $\mathcal{F}(\mathbf{m}(k)) = \mathcal{F}_{FB}([-1, 0]) = \{(0, 1), (0, -1)\}$, i.e. the second child returning Success or Failure. Note that if the first child would have returned Success, the feasible set would be empty $\mathcal{F}_{FB}([1, 0]) = \emptyset$.

The *Marking Reachability Graph* (MRG), see Figure 9.8, of a BT node can now be computed starting from the initial marking $\mathbf{m}(0) = \mathbf{m}_0 = \mathbf{0}^\top$, taking into account all the possible event combinations that satisfy the feasibility condition.

Definition 9.12. A marking \mathbf{m}_i is reachable from a marking \mathbf{m}_j if there exists a sequence of feasible events $\sigma = [\sigma_1, \sigma_2, \dots, \sigma_g]$ such that $\mathbf{m}_i = \mathbf{m}_j + \sum_{h=1}^g \sigma_h$.

Remark 9.4. Note that $\mathbf{m}(k) = \mathbf{m}_i$ when \mathbf{m}_i is the marking at time k .

9.2.1 Computing Transition Properties of the DTMC

The MRG of a BT node comprises all the reachable markings, the transitions between them describe events which have a certain success/failure probability. We

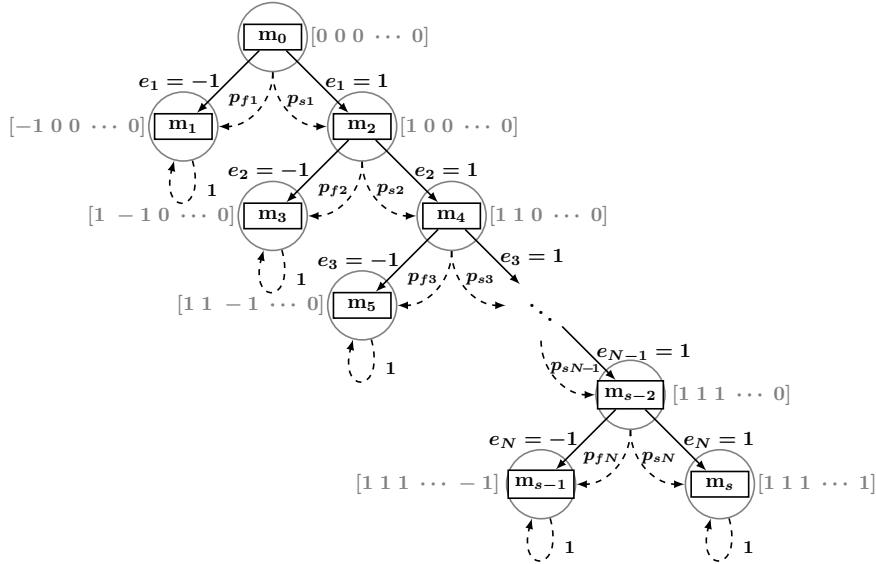


Fig. 9.8: MRG of the Sequence node (rectangles) with N children and its DTMC representation (circles).

can then map the node execution to a DTMC where the states are the markings in the MRG and the one-step transition matrix P is given by the probability of jump between markings, with off diagonal entries defined as follows:

$$p_{ij} = \begin{cases} \tilde{p}_{sh} & \text{if } \mathbf{m}_j - \mathbf{m}_i \in \mathcal{F}(\mathbf{m}_i) \wedge e_h \mathbf{e}_h^T (\mathbf{m}_j - \mathbf{m}_i) > 0 \\ \tilde{p}_{fh} & \text{if } \mathbf{m}_j - \mathbf{m}_i \in \mathcal{F}(\mathbf{m}_i) \wedge e_h \mathbf{e}_h^T (\mathbf{m}_j - \mathbf{m}_i) < 0 \\ 0 & \text{otherwise} \end{cases} \quad (9.36)$$

and diagonal entries defined as:

$$p_{ii} = 1 - \sum_j p_{ij}. \quad (9.37)$$

with:

$$\tilde{p}_{sh} = \frac{p_{sh} \mu_h v_h}{p_{fh} \mu_h + p_{sh} v_h} \cdot \left(\sum_{j: \mathbf{e}_j \in \mathcal{F}(\mathbf{m}_i)} \frac{\mu_j v_j}{p_{fj} \mu_j + p_{sj} v_j} \right)^{-1} \quad (9.38)$$

and

$$\tilde{p}_{fh} = \frac{p_{fh} \mu_h v_h}{p_{fh} \mu_h + p_{sh} v_h} \cdot \left(\sum_{j: \mathbf{e}_j \in \mathcal{F}(\mathbf{m}_i)} \frac{\mu_j v_j}{p_{fj} \mu_j + p_{sj} v_j} \right)^{-1} \quad (9.39)$$

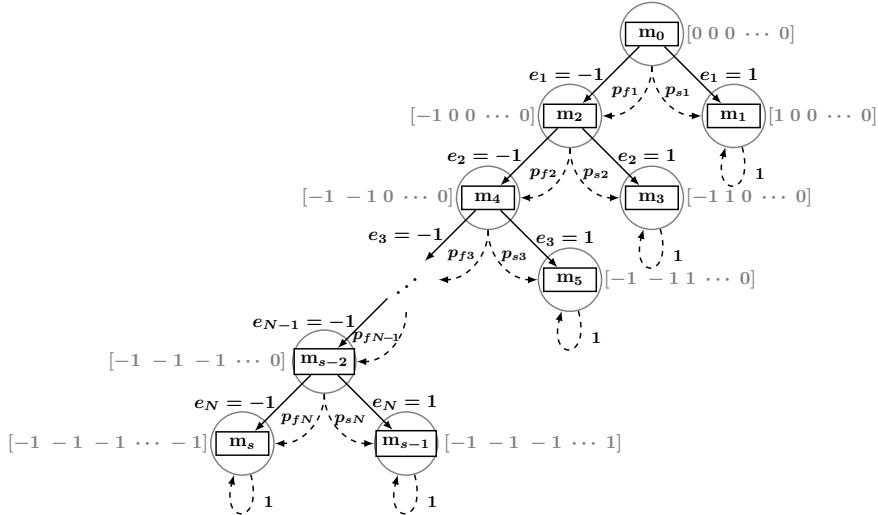


Fig. 9.9: MRG of the Fallback node (rectangles) with N children and its DTMC representation (circles).

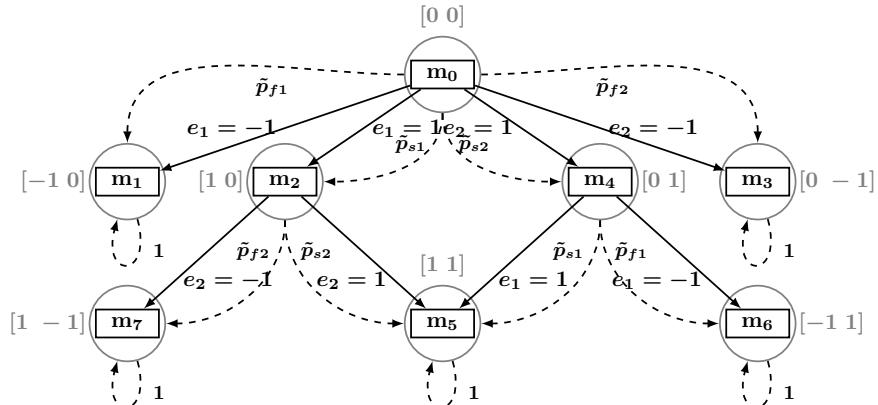


Fig. 9.10: MRG of the Parallel node (rectangles) with 2 children and its DTMC representation (circles).

where p_{sj} and p_{fj} is the p_s and p_f of child j .

Remark 9.5. For Sequence and Fallback nodes the following holds: $\tilde{p}_{sh} = p_{sh}$ and $\tilde{p}_{fh} = p_{fh}$.

In Figures. 9.8 and 9.9 the mapping from MRG to a DTCM related to a Sequence node and a Fallback node are shown. In Figure 9.10 the mapping for a Parallel node with two children and $M = 2$ is shown. We choose not to depict the mapping of

a general Parallel node, due to its large amount of states and possible transition between them.

To obtain the continuous time probability vector $\pi(t)$ we need to compute the infinitesimal generator matrix Q associated with the BT node. For doing so we construct a CTMC for which the EMC is the DTMC of the BT node above computed. According to (9.7) the map from the EMC and the related CTMC is direct, given the average sojourn times SJ_i .

9.3 Reliability of a SBT

9.3.1 Average sojourn time

We now compute the average sojourn time of each marking \mathbf{m}_i of a BT node.

Lemma 9.1. *For a BT node with $p_{si}, p_{fi}, \mu_i, v_i$ given for each child, the average sojourn time of in a marking \mathbf{m}_i is:*

$$SJ_i = \left(\sum_{h: \mathbf{e}_h \in \mathcal{F}(\mathbf{m}_i)} \left(\frac{p_{sh}}{\mu_h} + \frac{p_{fh}}{v_h} \right)^{-1} \right)^{-1} \quad (9.40)$$

with $h : \mathbf{e}_h \in \mathcal{F}(\mathbf{m}_i)$.

Proof. In each marking one of the following occur: the running child h fails or succeeds. To take into account both probabilities and time rates, that influence the average sojourn time, we describe the child execution using an additional CTMC, depicted in Figure 9.11

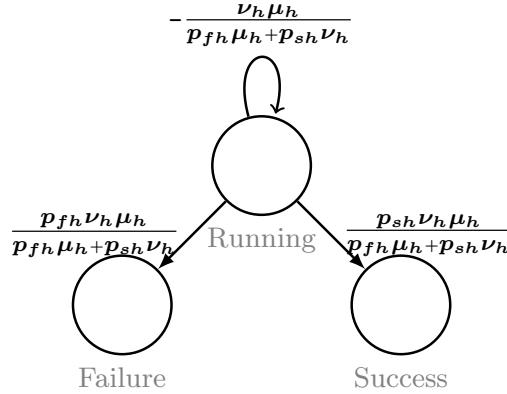
According to (9.9) the average sojourn time is:

$$\tau_i = \frac{p_{fh}\mu_h + p_{sh}v_h}{v_h\mu_h} = \frac{p_{sh}}{\mu_h} + \frac{p_{fh}}{v_h} \quad (9.41)$$

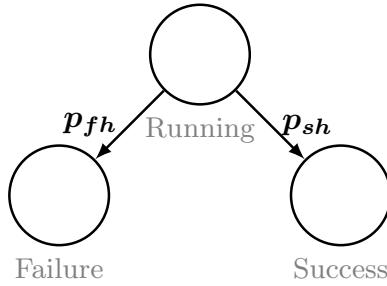
and the rate of leaving that state is τ_i^{-1} . Now to account all the possible running children outcome, e.g. in a Parallel node, we consider all the rates associate to the running children. The rate of such node is the sum of all the rates associated to the running children τ_i^{-1} . Finally, the average sojourn time of a marking \mathbf{m}_i is given by the inverse of the combined rate:

$$\frac{1}{SJ_i} = \sum_{h: \mathbf{e}_h \in \mathcal{F}(\mathbf{m}_h)} \frac{1}{\frac{p_{sh}}{\mu_h} + \frac{p_{fh}}{v_h}} \quad (9.42)$$

from which we obtain (9.40).

**Fig. 9.11:** CTMC of a child's execution.

Remark 9.6. The EMC associated with the CTMC in Figure 9.11 is depicted in Figure 9.12. It describes the child's execution as a DTMC.

**Fig. 9.12:** DTMC of a child's execution.

9.3.2 Mean Time To Fail and Mean Time To Succeed

To derive a closed form of the mean time to fail MTTF/MTTS of a BT node, we take the probability to reach a success/failure state from the DTCM and the average time spent in each state visited before reaching this state obtained from (9.40). We rearrange the state space of the DTMC so that the initial state is first, the other transient states are second, the failure states are second last and the success states are last:

$$P_c^\top = \begin{bmatrix} T & 0 & 0 \\ R_F & \mathbb{I} & 0 \\ R_S & 0 & \mathbb{I} \end{bmatrix} \quad (9.43)$$

where T is the matrix describing the one-step transition from a transit state to another one, R_F is a the matrix describing the one-step transition from a transit state to a failure state, and R_S is the matrix describing the one-step transition from a transit state to a success state. We call this rearrangement the *canonization* of the state space.

Lemma 9.2. *Let A be a matrix with the ij -th entry defined as $\exp(t_{ij})$ where t_{ij} is the time needed to transit from a state j to a state i if j, i are neighbors in the MRG, 0 otherwise. The MTTF and MTTS of the BT node can be computed as follows*

$$\text{MTTF} = \frac{\sum_{i=1}^{|S_F|} u_{i1}^F \log(h_{i1}^F)}{\sum_{i=1}^{|S_F|} u_{i1}^F} \quad (9.44)$$

where:

$$H^F \triangleq A_F \sum_{i=0}^{\infty} A_T^i. \quad (9.45)$$

and

$$\text{MTTS} = \frac{\sum_{i=1}^{|S_S|} u_{i1}^S \log(h_{i1}^S)}{\sum_{i=1}^{|S_S|} u_{i1}^S} \quad (9.46)$$

where:

$$H^S \triangleq A_S \sum_{i=0}^{\infty} A_T^i \quad (9.47)$$

where A_T , A_F , and A_S are the submatrices of A corresponding to the canonization described in (9.43), for which the following holds:

$$A = \begin{bmatrix} A_T & 0 & 0 \\ A_F & 0 & 0 \\ A_S & 0 & 0 \end{bmatrix}. \quad (9.48)$$

Proof. Failure and success states are absorbing, hence we focus our attention on the probability of leaving a transient state, described by the matrix U , defined below:

$$U = \sum_{k=0}^{\infty} T^k, \quad (9.49)$$

Thus, considering i as the initial transient state, the entries u_{ij} is the mean number of visits of j starting from i before being absorbed, we have to distinguish the case in which the absorbing state is a failure state from the case in which it is a success state:

$$U^F \triangleq R_F U \quad (9.50)$$

$$U^S \triangleq R_S U. \quad (9.51)$$

Equations (9.50) and (9.51) represent the mean number of visits before being absorbed in a failure or success state respectively.

To derive MTTF/MTTS we take into account the mean time needed to reach every single failure/success state with its probability, normalized over the probability of reaching any failure (success) state, starting from the initial state. Hence we sum the probabilities of reaching a state starting from the initial one, taking into account only the first column of the matrices obtaining (9.44) and (9.46).

Remark 9.7. Since there are no self loops in the transient state of the DTMC above, the matrix T is nilpotent. Hence u_{ij} is finite $\forall i, j$.

9.3.3 Probabilities Over Time

Since all the marking of a BT node have a non null corresponding average sojourn time, the corresponding DTMC is a EMC of a CTMC with infinitesimal generator matrix $Q(t)$ as defined in (9.7). Hence, we can compute the probability distribution over time of the node according to (9.8) with the initial condition $\pi_0 = [1 \mathbf{0}]^\top$ that represents the state in which none of the children have returned Success/Failure yet.

9.3.4 Stochastic Execution Times

Proposition 9.1. *Given a SBT, with known probabilistic parameters for actions and conditions, we can compute probabilistic measures for the rest of the tree as follows: For each node whose children have known probabilistic measures we compute the related DTMC. Now the probability of a node to return Success $p_s(t)$ (Failure $p_f(t)$) is given by the sum of the probabilities of the DTMC of being in a success (failure) state. Let $\mathcal{S}_S \subset \mathcal{S}_A$, and $\mathcal{S}_F \subset \mathcal{S}_A$ be the set of the success and failure states respectively of a DTMC related to a node, i.e. those states representing a marking in which the node returns Success or Failure, with $\mathcal{S}_F \cup \mathcal{S}_S = \mathcal{S}_A$ and $\mathcal{S}_F \cap \mathcal{S}_S = \emptyset$.*

Then we have

$$\bar{p}_s(t) = \sum_{i: s_i \in \mathcal{S}_S} \pi_i(t) \quad (9.52)$$

$$\bar{p}_f(t) = \sum_{i: s_i \in \mathcal{S}_F} \pi_i(t) \quad (9.53)$$

where $\pi(t)$ is the probability vector of the DTMC related to the node (i.e. the solution of (9.8)). The time to succeed (fail) for a node is given by a random variable with

exponential distribution and rate given by the inverse of the MTTS (MTTF) since for such random variables the mean time is given by the inverse of the rate.

$$\mu = \text{MTTS}^{-1} \quad (9.54)$$

$$\nu = \text{MTTF}^{-1} \quad (9.55)$$

Remark 9.8. Proposition 9.55 holds also for deterministic and hybrid BTs, as (9.8) has a unique solution in the Carathéodory sense [16].

9.3.5 Deterministic Execution Times

As the formulation of the deterministic case involves Dirac delta functions, see Equation (9.17)-(9.18), the approach described above might lead to computational difficulties. As an alternative, we can take advantage of the fact that we know the exact time of possible transitions. Thus, the success and failure probabilities of a deterministic node are unchanged in the intervals between the *MTTF* and *MTTS* of its children.

Example 9.6. Consider the BT

$$\mathcal{T} = \text{Fallback}(\mathcal{A}_1, \mathcal{A}_2) \quad (9.56)$$

depicted in Figure 9.13 and let τ_{Fi} (τ_{Si}) be the *MTTF* (*MTTS*) of action i and p_{fi} (p_{si}) its probability to fail (succeed). The success/failure probability over time of the tree \mathcal{T} is a discontinuous function depicted in Figure 9.14.

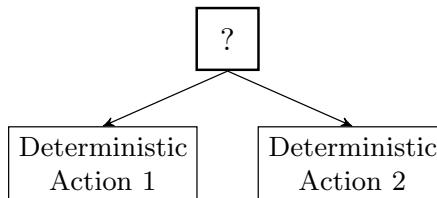


Fig. 9.13: Example of a Fallback node with two deterministic actions/subtrees.

Hence the success and failure probability have discrete jumps over time. These piece-wise continuous functions can be described by the discrete time system (9.3) introducing the information of the time when the transitions take place, which is more tractable than directly solving (9.8). Then, the calculation of $\pi(t)$ is given by a zero order hold of the discrete solution.

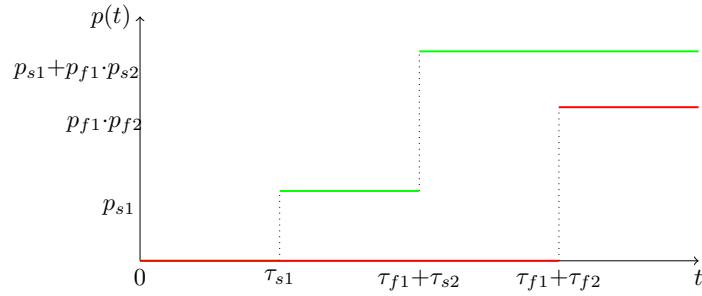


Fig. 9.14: Failure (red, lower) and success (green, upper) probability of the deterministic node of example. The running probability is the complement of the other two (not shown).

Proposition 9.2. Let P be the one-step transition matrix given in Definition 9.2 and let τ_{Fi} (τ_{Si}) be the time to fail (succeed) of action i and p_{fi} (p_{si}) its probability to fail (succeed). Let $\tilde{\pi}(\tau) = [\tilde{\pi}_1(\tau), \dots, \tilde{\pi}_{|\mathcal{S}|}(\tau)]^\top$, where $\tilde{\pi}_i(\tau)$ is the probability of being in a marking \mathbf{m}_i at time τ of a MRG representing a deterministic node with N children, let $\tilde{P}(\tau)$ be a matrix which entries $\tilde{p}_{ij}(\tau)$ are defined as:

$$\tilde{p}_{ij}(\tau) = \begin{cases} p_{ij} \cdot \delta(\tau - (\log(\tilde{a}_{j1}))) & \text{if } i \neq j \\ 1 - \sum_{k \neq i} \tilde{p}_{ik} & \text{otherwise} \end{cases} \quad (9.57)$$

with \tilde{a}_{ij} the ij -th entry of the matrix \tilde{A} defined as:

$$\tilde{A} \triangleq \sum_{i=0}^{\infty} A^i \quad (9.58)$$

with A as defined in (9.48).

Then the evolution of $\tilde{\pi}(k)$ process can be described as a discrete time system with the following time evolution:

$$\tilde{\pi}(\tau + \Delta\tau) = \tilde{P}(\tau)^\top \tilde{\pi}(\tau) \quad (9.59)$$

where $\Delta\tau$ is the common factor of $\{\tau_{F1}, \tau_{S1}, \tau_{F2}, \tau_{S2}, \dots, \tau_{FN}, \tau_{SN}\}$. Then for, deterministic nodes, given $\tilde{\pi}(\tau)$ the probability over time is given by:

$$\pi(t) = ZOH(\tilde{\pi}(\tau)) \quad (9.60)$$

where ZOH is the zero order hold function.

Proof. The proof is trivial considering that (9.59) is a piece-wise constant function and $\Delta\tau$ is the common fraction of all the step instants.

9.4 Examples

In this section, we present three examples. The first example is the BT in Figure 9.15a, which is fairly small and allows us to show the details of each step. The second example is the deterministic time version of the same BT, illustrating the differences between the two cases. The third example involves a more complex BT, shown in Figure 9.17. This example will be used to verify the approach numerically, by performing Monte Carlo simulations and comparing the numeric results to the analytical ones, see Table 9.2 and Figure 9.20. It is also used to illustrate the difference in performance metrics, between two equivalent BTs, see Figure 9.22.

We will now carry out the computation of probabilistic parameters for an example SBT.

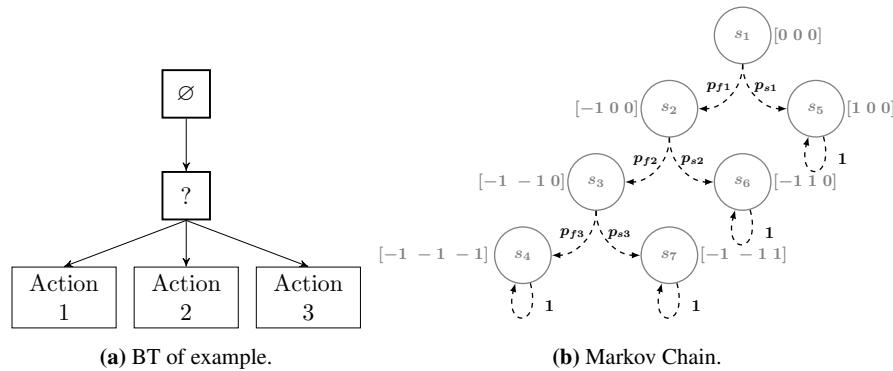


Fig. 9.15: BT and related DTMC modeling the plan of Example 9.4.

Example 9.7. Given the tree shown in Figure 9.15a, its probabilistic parameters are given by evaluating the Fallback node, since it is the child of the root node. The given PDF of the i -th action are:

$$\hat{p}_s(t) = p_{s_i} \mu e^{-\mu_i t} \quad (9.61)$$

$$\hat{p}_f(t) = p_{f_i} \nu e^{-\nu_i t} \quad (9.62)$$

where:

- p_{f_i} probability of failure
- p_{s_i} probability of success
- ν_i failure rate
- μ_i success rate

The DTMC related as shown in Figure 9.15b has $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$, $\mathcal{S}_F = \{s_4\}$ and $\mathcal{S}_S = \{s_5, s_6, s_7\}$.

According to the canonization in (9.43), the one-step transition matrix is:

$$P_c^\top = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_{f_1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & p_{f_2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & p_{f_3} & 1 & 0 & 0 & 0 \\ p_{s_1} & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & p_{s_2} & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & p_{s_3} & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.63)$$

According to (9.40) the average sojourn times are collected in the following vector

$$SJ = \left[\frac{p_{s_1}}{\mu_1} + \frac{p_{f_1}}{v_1}, \frac{p_{s_2}}{\mu_2} + \frac{p_{f_2}}{v_2}, \frac{p_{s_3}}{\mu_3} + \frac{p_{f_3}}{v_3} \right] \quad (9.64)$$

The infinitesimal generator matrix is defined, according to (9.7), as follows:

$$Q = \begin{bmatrix} \frac{-\mu_1 v_1}{p_{s_1} v_1 + p_{f_1} \mu_1} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\mu_1 v_1 p_{f_1}}{p_{s_1} v_1 + p_{f_1} \mu_1} & \frac{-\mu_2 v_2}{p_{s_2} v_2 + p_{f_2} \mu_2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{\mu_2 v_2 p_{f_2}}{p_{s_2} v_2 + p_{f_2} \mu_2} & \frac{-\mu_3 v_3}{p_{s_3} v_3 + p_{f_3} \mu_3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\mu_3 v_3 p_{f_3}}{p_{s_3} v_3 + p_{f_3} \mu_3} & 0 & 0 & 0 & 0 \\ \frac{\mu_1 v_1 p_{s_1}}{p_{s_1} v_1 + p_{f_1} \mu_1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{\mu_2 v_2 p_{s_2}}{p_{s_2} v_2 + p_{f_2} \mu_2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\mu_3 v_3 p_{s_3}}{p_{s_3} v_3 + p_{f_3} \mu_3} & 0 & 0 & 0 & 0 \end{bmatrix}. \quad (9.65)$$

The probability vector, according to (9.8), is given by:

$$\pi(t) = [\pi_1(t) \pi_2(t) \pi_3(t) \pi_4(t) \pi_5(t) \pi_6(t) \pi_7(t)]^\top \quad (9.66)$$

We can now derive closed form expression for MTTS and MTTF. Using the decomposition in (9.43), the matrices computed according (9.51) and (9.50) are:

$$U^S = \begin{bmatrix} p_{s_1} & 0 & 0 \\ p_{f_1} p_{s_2} & p_{s_2} & 0 \\ p_{f_1} p_{f_2} p_{s_3} & p_{f_2} p_{s_3} & p_{s_3} \end{bmatrix} \quad (9.67)$$

$$U^F = [p_{f_1} p_{f_2} p_{f_3} \ p_{f_2} p_{f_3} \ p_{f_3}] \quad (9.68)$$

Note that U^S is a 3×3 matrix and U^F is a 1×3 matrix since there are 3 transient states, 3 success state and 1 failure state. For action i we define $t_{f_i} = v_i^{-1}$ the time to fail and $t_{s_i} = \mu_i^{-1}$ the time to succeed. The non-zero entries of the matrix given by (9.48) are:

$$\begin{aligned} a_{2,1} &= e^{t_{f_1}} & a_{3,2} &= e^{t_{f_2}} & a_{4,3} &= e^{t_{f_3}} \\ a_{5,1} &= e^{t_{s_1}} & a_{6,2} &= e^{t_{s_2}} & a_{7,3} &= e^{t_{s_3}} \end{aligned} \quad (9.69)$$

from which we derive (9.45) and (9.47) as:

$$H^S = \begin{bmatrix} e^{t_{s_1}} & 0 & 0 \\ e^{t_{f_1}} e^{t_{s_2}} & e^{t_{s_2}} & 0 \\ e^{t_{f_1}} e^{t_{f_2}} e^{t_{s_3}} & e^{t_{f_2}} e^{t_{s_3}} & e^{t_{s_3}} \end{bmatrix} \quad (9.70)$$

$$H^F = [e^{t_{f_1}} e^{t_{f_2}} e^{t_{f_3}} \ e^{t_{f_2}} e^{t_{f_3}} \ e^{t_{f_3}}] \quad (9.71)$$

Using (9.44) and (9.46) we obtain the MTTS and MTTF. Finally, the probabilistic parameters of the tree are expressed in a closed form according to (9.52)-(9.55):

$$\bar{p}_s(t) = \pi_5(t) + \pi_6(t) + \pi_7(t) \quad (9.72)$$

$$\bar{p}_f(t) = \pi_4(t) \quad (9.73)$$

$$\mu = \frac{p_{s_1} + p_{f_1} p_{s_2} + p_{f_1} p_{f_2} p_{s_3}}{p_{s_1} t_{s_1} + p_{f_1} p_{s_2} (t_{f_1} + t_{s_2}) + p_{f_1} p_{f_2} p_{s_3} (t_{f_1} + t_{f_2} + t_{s_3})} \quad (9.74)$$

$$v = \frac{1}{t_{f_1} + t_{f_2} + t_{f_3}} \quad (9.75)$$

Example 9.8. Consider the BT given in Example 9.4, we now compute the performances in case when the actions are all deterministic.

The computation of MTTF and MTTS follows from Example 9.4, whereas the computation of $\pi(t)$ can be made according to Proposition 9.2.

According to (9.58) the matrix \tilde{A} takes the form below

$$\tilde{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ e^{t_{f_1}} & 0 & 0 & 0 & 0 & 0 & 0 \\ e^{t_{f_1}} e^{t_{f_2}} & e^{t_{f_2}} & 0 & 0 & 0 & 0 & 0 \\ e^{t_{f_1}} e^{t_{f_2}} e^{t_{f_3}} & e^{t_{f_2}} e^{t_{f_3}} & e^{t_{f_3}} & 0 & 0 & 0 & 0 \\ e^{t_{s_1}} & 0 & 0 & 0 & 0 & 0 & 0 \\ e^{t_{f_1}} e^{t_{s_2}} & e^{t_{s_2}} & 0 & 0 & 0 & 0 & 0 \\ e^{t_{f_1}} e^{t_{f_2}} e^{t_{s_3}} & e^{t_{f_2}} e^{t_{s_3}} & e^{t_{s_3}} & 0 & 0 & 0 & 0 \end{bmatrix} \quad (9.76)$$

thereby, according to (9.57), the modified one step transition matrix takes the form of Figure 9.16,

$$\begin{bmatrix} 1 - (p_{f_1} \delta(t - t_{f_1}) + p_{s_1} \delta(t - t_{s_1})) & 0 & 0 & 0 \\ p_{f_1} \delta(t - t_{f_1}) & 1 - (p_{f_2} \delta(t - (t_{f_1} + t_{f_2})) + p_{s_2} \delta(t - (t_{f_1} + t_{s_2}))) & 0 & 0 \\ 0 & p_{f_2} \delta(t - (t_{f_1} + t_{f_2})) & 1 - (p_{f_3} \delta(t - (t_{f_1} + t_{f_2} + t_{f_3})) + p_{s_3} \delta(t - (t_{f_1} + t_{f_2} + t_{s_3}))) & 0 \\ 0 & 0 & p_{f_3} \delta(t - (t_{f_1} + t_{f_2} + t_{f_3})) & 0 \\ p_{s_1} \delta(t - t_{s_1}) & 0 & 0 & 1 \\ 0 & p_{s_2} \delta(t - (t_{f_1} + t_{s_2})) & 0 & 0 \\ 0 & 0 & p_{s_3} \delta(t - (t_{f_1} + t_{f_2} + t_{s_3})) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 9.16: Modified one step transition matrix \tilde{P}^\top .

and the probability vector $\pi(t)$ is given by (9.60).

Below we present a more complex example, extending Example 9.4 above. We use this example for two purposes, first, to verify the correctness of the proposed approach using Monte Carlo simulations, and second, to illustrate how changes in the SBT lead to different performance metrics.

Example 9.9. The task given to a two armed robot is to find and collect objects which can be found either on the floor, in the drawers or in the closet. The time needed to search for a desired object on the floor is less than the time needed to search for it in the drawers, since the latter has to be reached and opened first. On the other hand, the object is more likely to be in the drawers than on the floor, or in the closet. Moreover, the available policies for picking up objects are the one-hand and the two-hands grasps. The one-hand grasp most likely fails, but it takes less time to check if it has failed or not. Given these options, the task can be achieved in different ways, each of them corresponding to a different performance measure. The plan chosen for this example is modeled by the SBT shown in Figure 9.17.

The performance estimates given by the proposed approach for the whole BT, as well as for two sub trees can be seen in Figures 9.18-9.19 .

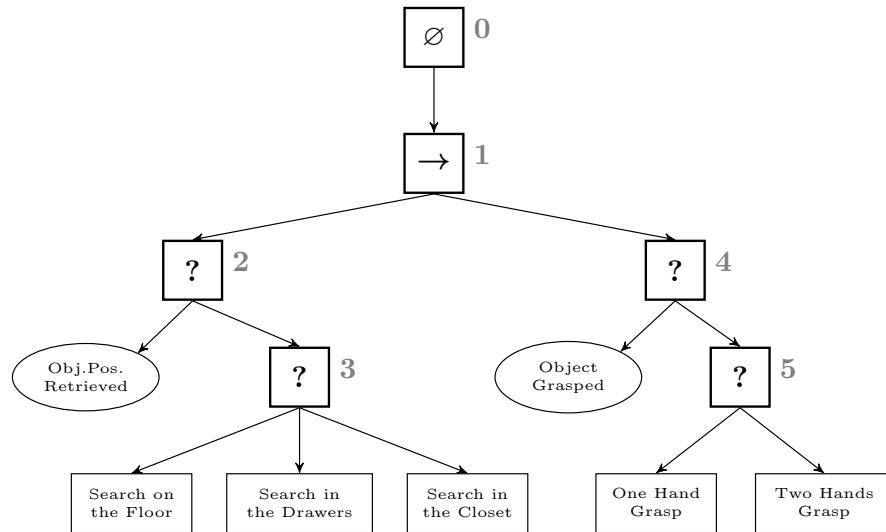


Fig. 9.17: BT modeling the search and grasp plan. The leaf nodes are labeled with a text, and the control flow nodes are labeled with a number, for easy reference.

We also use the example above to verify the correctness of the analytical estimates, and the results can be seen in Table 9.2. We compared the analytical solution derived using our approach with numerical results given by a massive Monte Carlo

simulation carried out using a BT implementation in the Robot Operative System (ROS) [38] where actions and conditions are performed using ROS nodes with outcomes computed using the C++ random number generator with exponential distribution. The BT implementation in ROS was run approximately 80000 times to have enough samples to get numerical averages close to the true values. For each run we stored if the tree (and some subtrees) succeeded or failed and how long it took, allowing us to estimate μ , v , $p_s(t)$, $p_f(t)$ experimentally. The match is reported in Figures 9.18-9.19 and in Table 9.1. As can be seen, all estimates are within 0.18 % of the analytical results.

Measure	Analytical	Numerical	Relative Error
μ_0	5.9039×10^{-3}	5.8958×10^{-3}	0.0012
v_0	4.4832×10^{-3}	4.4908×10^{-3}	0.0017
μ_3	6.2905×10^{-3}	6.2998×10^{-3}	0.0014
v_3	2.6415×10^{-3}	2.6460×10^{-3}	0.0017
μ_5	9.6060×10^{-2}	9.5891×10^{-2}	0.0018
v_5	4.8780×10^{-2}	4.8701×10^{-2}	0.0016

Table 9.1: Table comparing numerical and experimental results of MTTF and MTTS. The labels of the subscripts are given in Figure 9.17

Label	μ	v	$p_s(t)$	$p_f(t)$
Obj. Pos. Retrieved	—	—	$p_{s5}(t)$	$p_{f5}(t)$
Object Grasped	—	—	$p_{s4}(t)$	$p_{f4}(t)$
Search on the Floor	0.01	0.0167	0.3	0.7
Search in the Drawer	0.01	0.01	0.8	0.2
Search in the Closet	0.005	0.0056	0.2	0.8
One Hand Grasp	0.1	20	0.1	0.9
Two Hands Grasp	0.1	0.05	0.5	0.5

Table 9.2: Table collecting given parameters, the labels of the control flow nodes are given in Figure 9.17.

To further illustrate the difference between modeling the actions as deterministic and stochastic, we again use the BT in Figure 9.17 and compute the accumulated Success/Failure/Running probabilities for the two cases. Defining the time to succeed and fail as the inverse of the given rates and computing the probabilities as described in Section 9.3.5 we get the results depicted in Figures 9.20 and 9.21. As can be seen, the largest deviation is found in the Failure probabilities. In the stochastic case the CDF rises instantly, whereas in the deterministic case it becomes non-zero only after all the Fallbacks in at least one of the two subtrees have failed.

In Figure 9.22 the results of swapping the order of “Search on the Floor” and “Search in the Drawers” are shown in. As can be seen, the success probability after 100s is about 30% when starting with the drawers, and about 20% when starting with

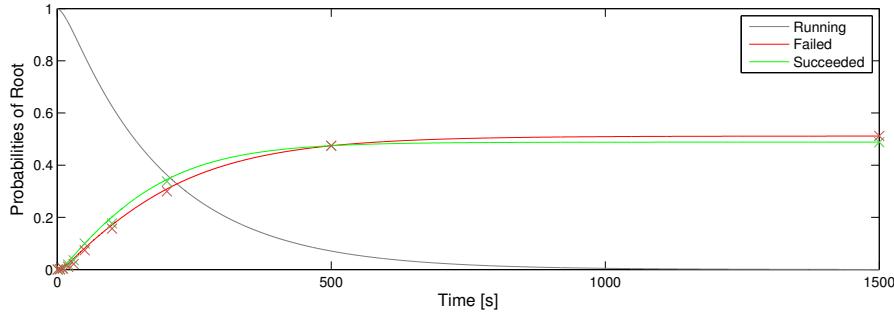
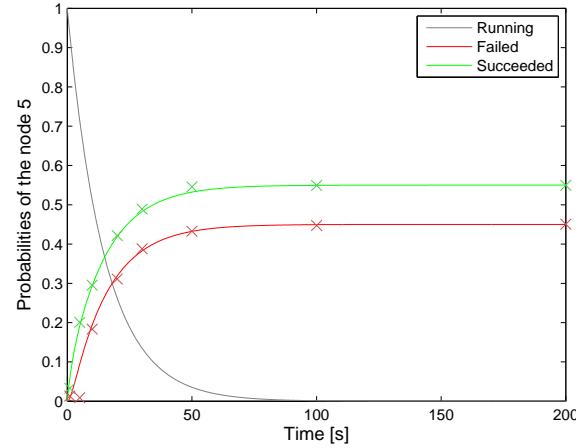
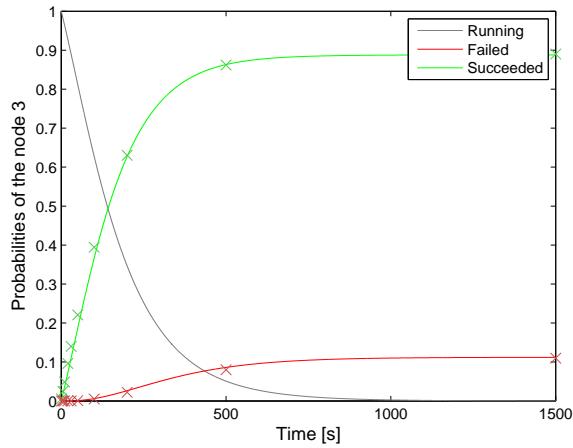


Fig. 9.18: Probability distribution over time for the Root node of the larger BT in Figure 9.17. Numerical results are marked with an 'x' and analytical results are drawn using solid lines. Note how the failure probability is initially lower, but then becomes higher than the success probability after $t = 500$.

the floor. Thus the optimal solution is a new BT, with the drawer search as the first option. Note that the asymptotic probabilities are always the same for equivalent BT, see Definition 9.10, as the changes considered are only permutations of Fallbacks.



(a) Node 5



(b) Node 3

Fig. 9.19: Comparison of probability distribution over time related to Node 5 (a) and Node 3 (b). Numerical results are marked with an 'x' and analytical results are drawn using solid lines. The failure probabilities are lower in both plots.

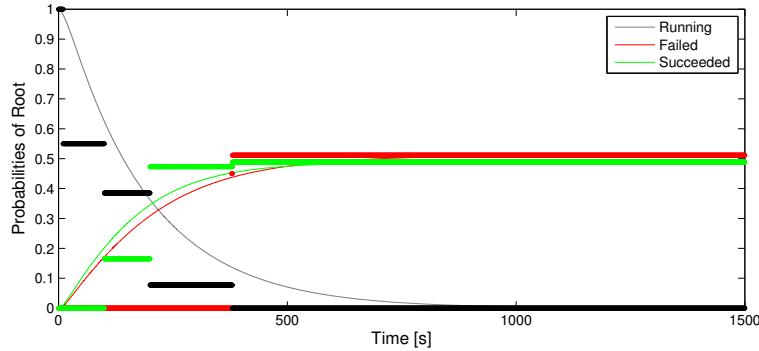
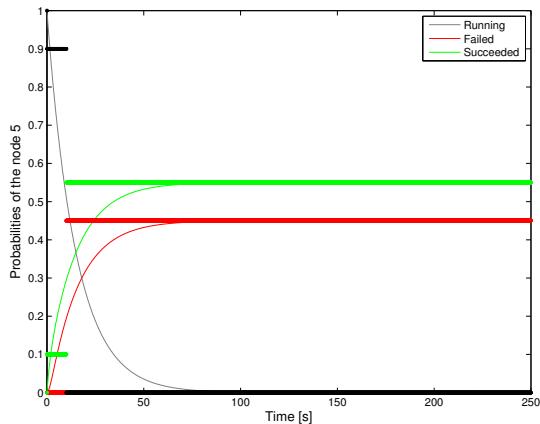
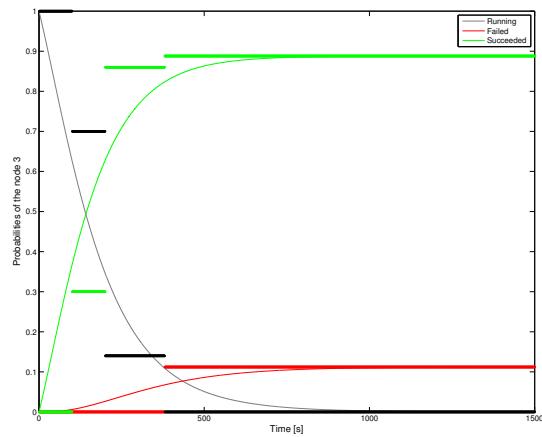


Fig. 9.20: Comparison of Success/Failure/Running probabilities of the root node in the case of deterministic times (thick) and stochastic times (thin).



(a) Node 5



(b) Node 3

Fig. 9.21: Comparison of Success/Failure/Running probabilities of the node 5 (a) and node 3 (b) in the case of deterministic times (thick) and stochastic times (thin).

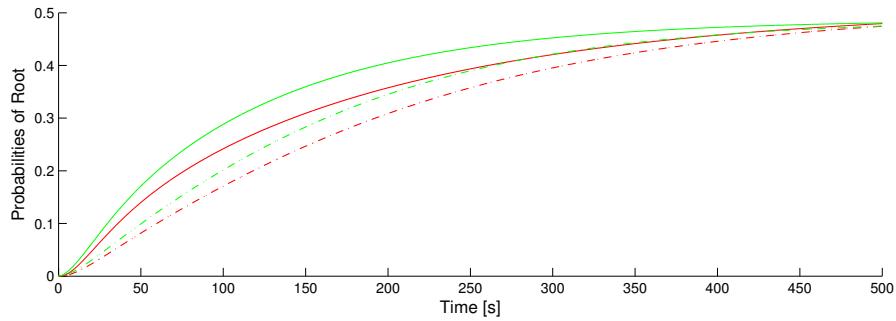


Fig. 9.22: Success/Failure probabilities in the case of searching on the floor first (dashed) and searching in the drawer first (solid). Failure probabilities are lower in both cases.

Chapter 10

Concluding Remarks

In this book, we have tried to present a broad, unified picture of BTs. We have covered the classical formulation of BTs, its extensions and its relation to other approaches. We have provided theoretical results on efficiency, safety and robustness, using a new state space formalism, as well as estimates on execution time and success probabilities using a stochastic framework. We have described a number of practical design principles as well as connections between BTs and the important areas of planning and learning.

We believe that modularity is the main reason behind the huge success of BTs in the computer game AI community, and the growing popularity of BTs in robotics. It is well known that modularity is a key enabler when designing complex, maintainable and reusable systems. Clear interfaces reduce dependencies between components and makes development, testing, and reuse much simpler. BTs have such interfaces, as each level of the tree has the same interface as a single action, and the internal nodes of the tree makes the implementation of an action independent of the context and order in which the action is to be used. Finally, these simple interfaces provide structures that are equally beneficial for both humans and machines. In fact, they are vital to the ideas of all chapters, from state-space formalism and planning to design principles and machine learning.

Thus, BTs represent a promising control architecture in both computer game AI and robotics. However, the parallel development in the field has given rise to a set of different formulations and variations on the theme. This book is an attempt to provide a unified view of a breadth of ideas, algorithms and applications. There is still lots of work to be done, and we hope the reader has found this book helpful, and perhaps inspiring, when continuing on the journey towards building better virtual agents and robots.

References

1. David Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-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*, 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 Marzinootto, 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. *arXiv preprint arXiv:1504.05811*, 2015.
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, Belén 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, <https://code.google.com/p/bart/>. 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. JK Gershenson, GJ 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 Weighofer, 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, 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 Klöckner. 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 Marzlinotto, 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 Etinger, 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. S 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. *arXiv preprint arXiv:1611.06145*, 2016.
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-Verlag.
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. *Journal of Field Robotics*, 25(8):425–466, 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. *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.

