

A principled analysis of Behavior Trees and their generalisations

Oliver Biggar, Mohammad Zamani and Iman Shames

Abstract

As complex autonomous robotic systems become more widespread, the goals of transparent and reusable Artificial Intelligence (AI) become more important. In this paper we analyse how the principles behind Behavior Trees (BTs), an increasingly popular tree-structured control architecture, are applicable to these goals. Using structured programming as a guide, we analyse the BT principles of *reactiveness* and *modularity* in a formal framework of action selection. Proceeding from these principles, we review a number of challenging use-cases of BTs in the literature, and show that reasoning via these principles leads to compatible solutions. Extending these arguments, we introduce a new class of control architectures we call *generalised BTs* or *k-BTs* and show how they can extend the applicability of BTs to some of the aforementioned challenging BT use-cases while preserving the BT principles. We compare BTs to a number of other control architectures within this framework, and show which forms of decision-making can and cannot be equivalently represented by BTs. This allows us to construct a hierarchy of architectures and to show how BTs fit into such a hierarchy.

1 Introduction

In the late 1960s and 1970s, software engineering underwent a transformation centred around transparency and code reuse. This transformation, known as the *structured programming* movement, has fundamentally changed the way people think about writing programs. As a result, today's programs can be orders of magnitude larger in size and complexity.

The benefits of these concepts have been noted for some time in Artificial Intelligence (AI) for Robotics [1, 2], but in the main they remain fringe goals. Robotic control architectures have historically been too simple to merit such abstractions. Finite State Machines (FSMs) have been dominant, despite the fact they increase in fragility with size [3, 4]. Recently though, as large-scale autonomous systems become widespread, many of the developments which were central to the structured programming movement are becoming important in robotic AI. There is growing recognition of the value of reusable and transparent behaviors which can be stored, extended and recombined to build agent behavior of great complexity [5–8].

One tool which has been growing in popularity in robotics and AI is the Behavior Tree (BT). These tree-structured control architectures have been presented as a replacement for FSMs, with proponents citing Dijkstra's pivotal letter "go to statement considered harmful" [9] and arguing FSMs resemble unconstrained use of the goto statement [3, 5, 10]. Unlike FSMs, BTs are *modular* in that every subtree is itself a BT with cohesive behavior, and *reactive*, in that they respond immediately to changes in the environment. The modularity allows subtrees to be easily reused in other trees and the reactiveness make their behavior transparent and predictable. These two principles have been suggested [5, 6, 10, 11] as the reason why BTs are reportedly easy to use [7, 12].

Nowadays, the principles of structured programming are ubiquitous (and form the basis for more complex programming paradigms, such as object-oriented programming), and yet few programming languages are ‘purely’ structured, in the sense of [13]. Most instead contain some exceptions to the rules, usually in the form of early exit from loops and functions by **break** and **return** statements respectively. In 1974, Knuth [14] essentially predicted this, stating “purists will point the way to clean constructions, and others will find ways to purify their use of floating-point arithmetic, pointer variables, assignments, etc., so that these classical tools can be used with comparative safety”. In that same paper, Knuth points out that while reducing the use of **goto** statements was in many cases desirable, certain constructions still required it—‘structuring’ them led to more obscure and opaque code. The benefits of structured programming still remain—the challenge is to identify precisely when those benefits can be exploited. If structured approaches fail, one must determine the next-best solution.

In this paper, we aim to study Behavior Trees in robotic AI in a similar way as Knuth, Dijkstra and others analysed structured programming. That is, we analyse the fundamental principles of BTs, *reactiveness* and *modularity*, and we look at how these principles allow for reusable and transparent behavior. Then, we provide a number of examples from the robotics literature and analyse where and how BTs should be applied so as to maximise their benefits. We show explicitly that not all FSM controllers can be equivalently reconstructed as BTs, and provide examples of how the use of FSMs in such cases can be done most transparently.

We provide answers to a number of open questions in the Behavior Tree literature, such as when a BT can be used in place of an FSM and at what level of abstraction BTs should operate. In doing so, we provide arguments for how actions and return values in BTs should be modelled and how BT extensions such as control flow nodes with memory influence their reactiveness and expressiveness. We show how BTs can be generalised to what we call the k -BTs, which preserve the reactiveness and readability of BTs while increasing their modularity. We provide a formal framework which describes how BTs structure action selection, and use this to compare BTs to other architectures. Finally, we enumerate the BTs up to ‘structural equivalence’, providing a numerical measure of the behaviors are which they can construct. We see these results as a toolkit for BT practitioners, to aid in determining when a BT could be an important part of a robotic architecture.

This work is a starting point for thinking about how modularity and transparency can be introduced to robotic AI using structures like BTs. Just as there are a number of proposed constructs in structured programming, so to there are many architectures which aim to achieve these same principles within robotics. Other architectures such as Decision Trees and Teleo-reactive programs are also discussed here, and much of our discussion will also be applicable to these, but our focus will be on BTs.

The rest of the paper is organised as follows. Section 2 introduces a number of control architectures commonly used in AI and robotics. These include BTs, FSMs, Decision trees (DTs) and Teleo-Reactive programs (TRs). Section 3 introduces a common framework for representing these control architectures, and formalises our definitions and assumptions. In Section 4 we discuss the two main principles of BT use: reactiveness and modularity. Section 5 then shows how these principles can be applied in practice. We review a number of common use cases for BTs and provide solutions motivated by the aforementioned principles. Section 6 introduces k -BTs and their application towards some common problems encountered in BTs. We formally study the expressiveness of BTs and other related architectures in Section 7.

2 Finite State Machines, Decision Trees, Teleo-reactive programs and Behavior Trees

These definitions are in general the same as those in [15].

2.1 Behavior Trees

BTs are control architectures which take the form of ordered directed trees. The execution of a BT occurs through signals called ‘ticks’, which are generated by the root node and sent to its children. A node is executed when it receives ticks. Internal nodes tick their children when ticked, and are called *control flow nodes* and leaf nodes are called *execution nodes*. When ticked, each node can return one of three possible return values; ‘Success’ if it has achieved its goal, ‘Failure’ if it cannot operate properly and ‘Running’ otherwise, indicating its execution is underway. Typically, there are four types of control flow nodes (Sequence, Fallback, Parallel, and Decorator) and two types of execution node (Action and Condition) [5]. Note that Fallback is sometimes called Selector. A Condition (drawn as an ellipse) checks some value, returning Success if true and Failure otherwise. An Action node (drawn as a rectangle) represents an action taken by the agent. Sequence nodes (drawn as a \rightarrow symbol) tick their children from left to right. If any children return Failure or Running that value is immediately returned by the Sequence node, and it returns Success only if every child returns Success. Fallback (drawn as a $?$) is analogous to Sequence, except that it returns Failure only if every child returns Failure, and so on. The Parallel node (symbol \Rightarrow) has a success threshold M , and ticks all of its N children simultaneously, returning Success if M of its children return Success, Failure if $N - M + 1$ return Failure and Running otherwise. The Decorator node returns a value based on some user-defined policy regarding the return values of its children. For a more detailed discussion, we refer the reader to [5]. Following [16], we will often write BTs succinctly in infix notation, such as $A \rightarrow (B ? C) \rightarrow D$ where A, B, C, D are interpreted as leaf nodes and the control flow nodes \rightarrow and $?$ are interpreted as associative operators over the leaf nodes.

2.2 Finite State Machines

A Finite State Machine (FSM) is a control architecture structured as a labelled directed graph, where nodes represent *states*. The machine is in exactly one of these states at a given time. Arcs, called *transitions*, link states from one to another. A transition from one state to another is undertaken in response to input which *triggers* the transition. There is a single state called the initial state. States are labelled by actions. FSMs execute by beginning in the initial state, and whenever input is received, taking any transitions which are triggered by that input, until reaching a state where no transitions are triggered. The FSM then executes the action labelling the state until new input is received. Formally, an FSM is a six-tuple $(Q, q_0, \Sigma, \mathcal{A}, \delta, \ell)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite set called the input alphabet, \mathcal{A} is a finite set of actions, $\delta : \Sigma \times Q \rightarrow Q$ is the transition function dictating when transitions are triggered and $\ell : Q \rightarrow \mathcal{A}$ is the output function assigning actions to states. This particular formal interpretation of a Finite State Machine is called a *Moore machine* [17].

2.3 Teleo-reactive programs

Teleo-reactive programs (TRs) [2] are lists of condition-action rules.

$$\begin{aligned} k_1 &\rightarrow a_1 \\ k_2 &\rightarrow a_2 \\ &\vdots \\ k_n &\rightarrow a_n \end{aligned}$$

They are executed by continuously scanning the list of conditions k_i in order, and executing the action a_i associated with the first satisfied condition. As these conditions become true and false, the selected action changes immediately to the action corresponding to the first satisfied condition. The *teleo* indicates that such lists are goal-oriented while *reactive* is intended to describe how they react constantly to changes in the environment. Reactiveness in this sense is a central principle of this paper.

2.4 Decision Trees

Decision Trees (DTs) [5] are decision-making tools which resemble the structure of if-then statements. Specifically, DTs are binary trees where leaves represent actions and internal nodes represent predicates. The two arcs out of each predicate are labelled by ‘True’ and ‘False’. Execution of DTs occurs by beginning at the root and evaluating each predicate on the current input state until a leaf is reached, at which point that action is evaluated. At a predicate node, if it is true in the current input state the execution proceeds down the ‘True’ arc, and otherwise down the ‘False’ arc. Like TRs, DTs are executed by continuously checking the predicates against the current state of the world.

3 Assumptions and Definitions

3.1 Layers of architectures

In AI and software a common abstraction is to think of decision-making in layers, to help clearly separate concerns. From an AI perspective, we assume that higher layers act upon the objects in the layers beneath, but lower layers cannot affect higher layers. These ideas extends to BTs, because in general BTs make up only a part of a complex robotic architecture, which we will call the *BT layer* (see Fig 1). The layers below the BT correspond to controllers for the individual actions from which the BT selects. These could be continuous controllers operating on actuators or possibly themselves complex action selection mechanisms operating on yet lower layers—these too could be implemented as BTs¹. Likewise, there may be layers above the BT, which select this BT under various circumstances, and these again could also be BTs. BTs are also described in this way in [18]. We can thus rephrase describe the goal of our paper as finding what aspects of the decision-making are best structured within the BT layer².

¹The BT layer need not be truly a single layer. If BT implementations of layers appear distinctly in a robotic architecture then the same principles will apply to both and we will refer to either as ‘the’ BT layer. Note that if a BT selects a BT in an immediately lower layer these BTs can be joined to a single larger BT, by modularity.

²Of course, there may be no BT layer in general, but we can always refer to ‘the BT layer’ allowing for the fact that it may be empty.

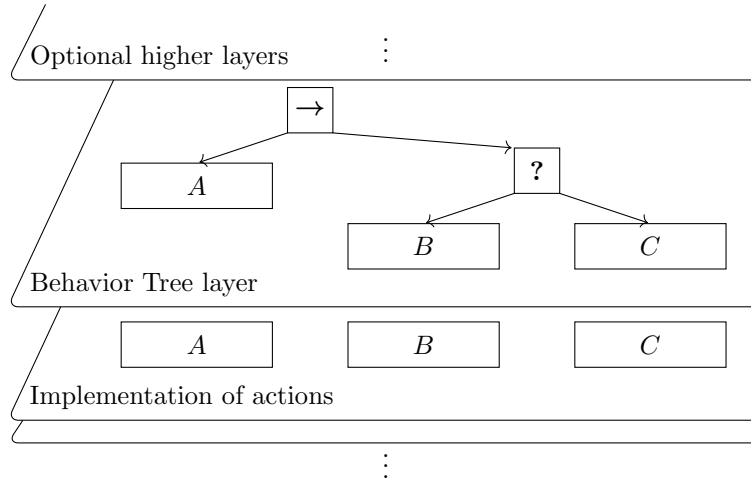


Figure 1: A layered robotic control architecture

3.2 Action selection mechanisms

In this paper we will wish to discuss how BTs compare to other forms of discrete task-switching. To formalise this comparison, we need a definition of action selection which is general enough capture the operation of a number of architectures, including BTs and FSMs. For this purpose we define an *action selection mechanism*.

Definition 3.1. Let \mathcal{A} be a finite set of *actions* and Σ a finite set of *inputs*. An *action selection mechanism* (ASM) is a map $f : \Sigma^* \rightarrow \mathcal{A}$, where Σ^* is the set of all finite sequences of elements of Σ . In other words, an ASM takes a finite sequence of inputs and produces an action.

By this definition, ASMs are deterministic. The input sequence to an ASM can be considered as the history of the input to the system, with the last element of this sequence as the current input. In control architectures, we tend to think of all only the current input being applied at any time step, rather than the entire history of the system. However, this abstraction represents the fact that ASMs may have access to *memory*, so that despite only one input being physically provided at any time in concrete implementations, the information of the entire history is available for decision-making. The ‘actions’ in the set \mathcal{A} represent units of computation on the next layer, which could be of arbitrary complexity. In the BT case ‘actions’ in this sense will apply to either Action or Condition nodes, where we view Condition nodes as a special case of an Action where Success or Failure is always returned. Since our focus is on how structures switch between actions, the actual content of actions will in general not matter, so we will not focus on \mathcal{A} . We assume the inputs Σ describe the state of the ‘world’, perhaps given by discrete sensors, and again not concern ourselves with the actual content of Σ .

However, our intention is to use these to model control architectures repeatedly selecting an action in response to periodic input. In other words, our ASMs should map sequences (possibly infinite) of input to sequences of actions. We can easily interpret the above in this sense, in the following obvious way:

$$\text{Input: } x_1, x_2, x_3 \dots \in \Sigma \quad \text{Output: } f(x_1), f(x_1, x_2), f(x_1, x_2, x_3), \dots \in \mathcal{A}$$

This sequence represents the intuition of how concrete ASMs act. At each step i the additional input x_i is provided, but the ASM can make use of all x_1, \dots, x_i inputs potentially stored in

memory. Observe that if two ASMs produce different output on the same input sequence then they are distinct. We will use this to compare ASMs.

Before explaining how concrete control architectures are interpreted as ASMs, we must make some assumptions. This model of action selection assumes there is some outward universe to which the action selection mechanism provides constant feedback. That feedback is given by the currently selected action. At a sequence of discrete time points, the universe updates its state, and this generates inputs to the ASM carrying this universe state information, which the ASM can use to update its selection. We assume the process of computing the selection takes negligible time on the universe time-scale. As an example, this assumption means that the time between when a tick is generated by the root of a BT and when that root returns a value is negligible with respect to the outside world. This assumption is reasonable in control architectures where computation is fast in comparison with real-world changes.

Example 3.2. Here we explain how FSMs, BTs and DTs are interpreted as ASMs. Firstly, we assume that for any input $x \in \Sigma$, we can determine whether an action returns any given value in x (such as Success or Failure) or if a transition in an FSM is triggered in x . Let $x_1, \dots, x_n \in \Sigma^*$ be a sequence of input.

FSM: Begin at the start node. For each x_i , if a transition is triggered in the current state, then update the current state follow each transition which is triggered in x_i until we reach a state q from which x_i triggers no transitions. If there is no more input ($i = n$), return the action α labelling q , otherwise process the next input x_{i+1} , starting at q .

BT: Process the input x_i from the root, traversing the tree on the basis of the return values of the subtrees in x_i . If x_i is the last input, return the action labelling the leaf node that was ticked last in the traversal, otherwise start again from the root with x_{i+1} .

DT: Process the input x_i from the root, traversing the tree on the basis of which conditions are true in x_i until a leaf is reached. If x_i is the last input, return the action labelling that leaf, otherwise start again from the root with x_{i+1} .

TR: Process the input x_i from the start of the list, and test in order whether any of the preconditions are true in x_i . If x_i is the last input, return the action corresponding to the first true precondition that leaf, otherwise start again with x_{i+1} .

Given any of these architectures, the associated ASM M we will call its *derived ASM*.

Note that the action selected by both DTs and BTs depends only on the final input x_n . This is because both are reactive—indeed, we will use this as the definition of reactivity, as we discuss in Section 4.1. Note also that BTs return the last leaf ticked, regardless of whether that leaf returned Success or Failure. To see why this definition makes sense, note that the root returns Success/Failure/Running iff the last child ticked returns Success/Failure/Running³. As a result, the overall return value of the tree can be derived immediately from its selected action at any step (see [15] for an extended explanation of this point)⁴. Other authors [19] have interpreted ‘overall tree Success/Failure’ as additional trivial actions which can be selected by the tree, but this does not integrate with tree composition—if the tree truly selects an action ‘Success’ instead of returning the value Success, that tree will not behave correctly when embedded as a subtree of another BT, violating the principle of *modularity* (Section 4.2).

³There is a further assumption inside this statement—that Negation decorators are not used. Again this assumption is harmless; by Lemma 4.18 of [16] all Negation decorators can be propagated downwards and absorbed into modified leaves.

⁴In addition, this works for any other return values being returned, instead of a specific Running value, consistent with the conclusions of Section 5.2.4

4 The principles of Behavior Tree use

In order to reason about when BTs should be used, we must know why they should be used at all. In general when explaining why BTs are useful, the arguments which are recurrent in BT literature are the following:

- Behavior Trees are reactive [5, 11, 15, 16, 20]
- Behavior Trees are modular [3, 5, 10, 11, 15, 16, 20]

Fundamentally, these properties make BTs easy to construct, analyse, read and reuse without errors. Some authors have additionally described *reusability* [3, 10, 16, 21] or *readability* [7, 11, 21] as principles of BT use. While we agree with their fundamental importance, we shall view these as the end to which reactivity and modularity are a means. That is, readability and reusability should be the goals of all ‘structured robotic AI’, and we consider modularity and reactivity to be the tools by which BTs specifically achieve those goals. Now we unpack both in some more depth.

4.1 Reactiveness

Intuitively, reactivity should mean that when the environment changes the system should always respond appropriately—that is, it should be *reactive* to changes in the input. This isn’t sufficiently specific however, because all ASMs respond to sequences of input with an action, and not all are considered reactive. For instance, a FSM with state set Q and with inputs Σ can be considered as a function $\Sigma \times Q \rightarrow \mathcal{A}$ by composing its transition and output functions. Thus an FSM always ‘reacts’ to any input $(s, q) \in \Sigma \times Q$, but FSMs are generally not considered to be reactive. The reason for this is that the internal state q is not outwardly visible to the user. In other words, reactivity means that given some input it should be clear from the structure which action is selected—no additional information should be necessary. Using this, we arrive at the following definition of reactivity, consistent with [2, 15, 18]:

Definition 4.1. An ASM $f : \Sigma^* \rightarrow \mathcal{A}$ is *reactive* if there exists a map $r : \Sigma \rightarrow \mathcal{A}$ such that $\forall x_1 \dots x_n \in \Sigma^*, f(x_1 \dots x_n) = r(x_n)$.

In other words, in a reactive ASM the selection is influenced only by the current input. We shall write the class of such ASMs as **Reactive**.

Reactivity makes reasoning about BTs extremely intuitive; given some external world state as input, it is always straightforward to determine which action will be selected in response. Reactive ASMs can be thought of as a partition of the input space, where regions of the partitions are labelled by the action selected in that region. Many authors have argued that reactivity is at the core of BT use [5, 15, 16, 20], and that a primary goal of BTs is to add a reactive layer to a control architecture. From this, we state the following principle.

Principle 1. Behavior Trees should be reactive; whenever the current input is the same, they should always select the same action regardless of input history.

Reactivity can be thought of as *memorylessness*. When considered in this way, we can think of reactive behaviors like instinctive or unconscious behaviors in a biological system. The limited computational load required to process BTs forms a core part of their readability, and their applicability to a variety of systems, including those with very limited computing power. Of course, we achieve this simplicity and clarity by restricting the access of an ASM to memory, so we would expect this to come at the cost of expressiveness. Indeed, as we show in Section 7, this is the case.

We propose the following test for the property of reactivity. Consider a robot in an empty room, and a human observer. An architecture is reactive if, when provided with the same stimulus, the robot always reacts in the same way. This is fundamentally tied to its behavior being *predictable* and *transparent*. The human observer should always be able to predict the robot’s response to any given stimulus, given a representation of its architecture, without any knowledge of what prior stimulus the robot had been exposed to. This test will be useful for highlighting cases where reactivity is violated.

4.2 Modularity

The modularity of BTs means that they can be composed, decomposed and reused because subtrees and actions are themselves well-defined BTs. This modularity comes about both from structure (being a recursively defined tree structure) and having a fixed interface given by three return values. Modularity is widely regarded as key to the use of BTs [3, 5, 10, 11, 15, 16, 20], as it allows libraries of BTs to be created which can be easily combined. Any constructed subtree of a BT can be reused in any other tree. From this, we propose the following principle.

Principle 2. Behavior Trees should be modular; every subtree, including the whole tree, should be able to be reused meaningfully in another tree.

5 Applying these principles in practice

So far, we have not justified why these principles are sufficient, or whether they can actively help us to judge cases of BT use in practice. In this section we show that properly interpreting these principles is enough to provide insights or solutions to a number of BT challenges. The examples used in this section are inspired by examples used in the BT literature. Structurally, we split this section by problems which primarily address reactivity or modularity. This partition is not strict, and many of the points made are relevant to both principles.

5.1 Reactiveness

5.1.1 Implicit memory

Consider the following BT, assumed to be controlling a robot.

$$(\text{Battery} < 10\% \rightarrow \text{Recharge}) ? \text{Other task}$$

When the robot’s battery is less than 10% it recharges, and otherwise it performs some task. The issue is, implemented as shown, we can have chattering where the robot fluctuates rapidly between Recharge and Other task as the battery level fluctuates around 10%. It is tempting to change this to the following:

$$((\text{Battery} < 10\% \vee \text{Recharging}) \rightarrow \text{Recharge}) ? \text{Other task}$$

Here we have introduced an auxiliary variable *Recharging* which becomes true after we begin recharging and becomes false once the battery reaches 100%. Now the robot remains doing ‘Recharge’ until its battery is full. However, in doing this, we have violated the principle of reactivity, and lost transparency. Consider some input where the battery level is $> 10\%$. An observer cannot predict how the robot will behave in this state without knowing whether it had begun Recharging *at some point in the past*. This auxiliary variable has introduced what we shall call *implicit memory*. As for the Finite State Machine example given in Section 4.1, the

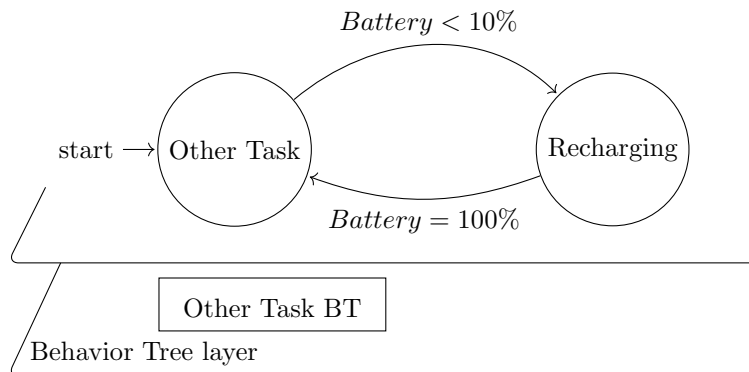


Figure 2: A solution to the chattering problem

variable *Recharging* cannot be considered part of the input because it is not externally visible. We cannot reason about this variable as we would about a property of the environment, and so the operation of the tree is no longer transparent. This variable cannot take different values at any point—it becomes true and false under prescribed conditions, but the information of these conditions is not contained in the above BT. This breaks the cohesiveness of the architecture, and from a practical perspective means that manipulation of this variable must also be implemented outside the normal software framework for the BT. This can easily lead to errors.

Indeed implicit memory will appear repeatedly in our following discussions of BT use, because it is commonly applied as a workaround for introducing memory to BTs. We argue implicit memory should not be used as it will violate reactivity and reduce readability.

How instead should we handle challenges such as this chattering problem? Here, the issue stems from the fact that the robot should not be reactive while it is charging. To be truly reactive, architectures must be able to interrupt their children in response to any new stimulus from the environment. In other words, the BT layer can always interrupt all layers beneath it. If we instead desire actions which will not be interrupted, we propose placing them at a layer above the BT layer (or any reactive layer). We show this solution in Figure 2, where an FSM controls switching in and out of a recharging state. When not recharging, the BT controlling ‘Other task’ is operated on the layer beneath. The overall architecture is still not reactive, but this approach clearly separates the reactive and non-reactive layers and the transition conditions for *Recharging* are now transparent, improving readability.

We used this example to introduce the challenges of implicit memory, but using implicit memory is not the only proposed solution to this specific chattering problem in the BT literature. For example, [11] uses a notion called *transient tasks* to address this problem.

This example and implicit memory solution were adapted from [10].

5.1.2 Modelling Success and Failure

When designing BTs or abstracting some implemented behavior into actions, one must decide on the conditions under which actions should succeed or fail. We shall call these conditions the *return conditions*. We show now that we cannot do this freely if we are to preserve reactivity—in a sense, these conditions must also be ‘reactive’.

To begin, consider an action ‘Send Message’. When should this return Success or Failure? Requiring that it returns Success if ‘a message has been sent’ is problematic, because it introduces implicit memory, violating reactivity. To see this, note that in English this condition is stated

in past tense—it refers to what has happened previously. We cannot evaluate its truth or falsity without memory. Just as with previous example of implicit memory, the variable storing when ‘a message has been sent’ is not defined within the tree, and is controlled at some other layer, breaking transparency. Consider a more extreme example, where an action returns Success if a given sequence of variables has been true in the past. Using such actions would allow us to construct arbitrary ASMs, so must violate reactivity. These same ideas apply to conditions. Consider an example of an agent who must pass through a door, with action ‘Walk Forward’ and a condition ‘Agent Has Passed’ checking when the agent reaches the next room. The problem with this condition is in its past-tense description—if the agent is simply in the next room, there is no way of knowing if it did or did not pass through the door in the past.

This problem can occur easily, but can also often be solved by rewriting conditions in present tense, or omitting them if no present-tense conditions are reasonable. Consider an action ‘Grasp Object’. One intuitive way of modelling its return conditions is to say it returns Success if an object has been grasped and failure if it cannot grasp an object. However, this requires implicit memory to record whether we have already attempted to grasp the object. However, we can express this reactively by having Grasp Object return Success if the robot is holding an object and Failure if there is no object within reach. Thus, if there is an object within reach and not grasped, the robot executes a grasp operation until either it is holding the object or the object is no longer present. Both return conditions are both present-tense, and refer to observable states. One way of thinking about the distinction between these models is that in a reactive architecture we care about outcomes not processes—it matters only that we are now holding an object, not whether we actually used the grasp action to achieve that outcome. Likewise, the Agent Has Passed condition could be replaced with ‘Agent in Room X’, a present-tense condition checking the current location.

In arguing that ‘a message was sent’ was implicit memory, we assumed that this variable is not outwardly visible, and cannot be interpreted as sensor input to the BT. Another solution to this problem is to rephrase this in a way that can be interpreted as an input from the outside world. For instance, suppose Send Message returns Success if the robot observes a human operator nodding, or another agent receiving the message provides an acknowledgement of its receipt. This may also not be particularly outwardly visible—this itself presents a challenge, discussed in Section 5.1.3. However, these options at least avoid the need for implicit memory.

One common source of these examples is discrete or non-durative actions. A simple case is the action Step Forward for a BT operating in a discrete grid world. How should we assign a Success condition to this action? We should avoid success conditions along the lines of ‘Success if we have stepped forward’ or ‘Success if we are our position is one forward from our last position’, because, again, they are past-tense. We postulate that there is no reactive way to assign a Success conditions to such actions—a better solution is to never return Success. Instead the agent should step forward repeatedly until the BT switches tasks or some present-tense Failure condition becomes true, such as *PathObstructed*. This example is also explored in Section 5.2.2, where we argue that very low-level discrete actions are hard to use in BTs.

The takeaway of this section is that for a BT to be reactive, the Success and Failure conditions of each action must also be reactive. That is, the return value of all actions must be determined only by current state.

The Sent Message example was adapted from [16], and the door example was adapted from [5]. A similar example using waypoints is discussed in [11].

5.1.3 Knowledge versus memory

In the last section we concluded that actions should be able to return Success or Failure on the basis of current input only. In other words, the return value should not be based on whether ‘something was done’ but rather whether ‘something is true’. There are, however, cases where this thinking itself introduces additional challenges.

Consider an action Unlock Door. Some Success and Failure conditions for this action might be the variables *DoorUnlocked* and *NoKey* respectively. However, from the robot’s world view, whether a closed door is locked or unlocked may not be clear from input. It is too generous to assume this information is provided when such doors appear identical. In order to determine the value of *DoorUnlocked*, the robot must first try some action, say, attempting to unlock the door. In other words, Unlock Door might return Running even though its Success condition is in fact satisfied, depending on whether or not it is aware of this, which seems to violate reactivity.

There are a few possible solutions to this problem. One is that we assume that all return conditions are based only on unambiguously visible inputs, thus rejecting any ambiguous cases of this form. However, this does impose significant limits on the operation of BTs. We recommend a more practical compromise: we require the robot to be reactive *in the context of its own world view*. In other words, from the BT perspective we allow *DoorUnlocked* to be either true, false or unknown, so the decisions of the BT are still reactive on this basis.

We must therefore make an important but subtle distinction between the concept of knowledge, which is the agent’s representation of the world, and memory, which is the agent’s representation of that world’s *history*. The former is essential for robotic applications, but reactivity argues that the latter is not. This internal world model abstracts sensor input into a cohesive structure which can be queried by Behavior Trees. Doing so is consistent with reactivity, so long as only the current state of this model is used to determine the action selected. This world view need not be static, and can update itself as new information is received, such as in the door unlocking example used above. We do instead require that this internal model be merely a representation of the external world, and not a blackboard for any information to be stored. If that were the case, variables such as *Recharging* could be stored there, allowing us the freedom to use implicit memory. Balancing these requirements can be difficult. We judge the overall approach using our test for reactivity, but where we assume that the observer’s world view is the same as that currently possessed by the robot. In this context, we require that robot’s behavior still be transparent given that the observer knows the robot does not have full information. A solution similar to this is suggested in [21].

Finally, we also note that while we should not use an internal world model as a store for arbitrary memory, we do allow, and indeed expect, that agents use the physical world as a store of memory. That is, if a robot opens a door and moves through it, then later returns to that same doorway and finds the door still open, it will move through it without reopening. This may seem obvious, but it is important. If the robot records every doorway it passes through by marking them with chalk, it could, in theory, retrace its steps by following doorways marked with chalk. Note that the robot has not remembered its path, but rather is reactively following the doors marked with chalk. While we argue against an ‘internal blackboard’ because it is not transparent, this ‘external blackboard’⁵ is transparent from the perspective of an outside observer and only depends on the present-tense conditions. If another agent added or removed these marks the robot would respond appropriately.

This example was adapted from [5].

⁵Rather literally.

5.1.4 Black-box selections

The following gives another example of the subtleties of storing information. Consider a robot which can be doing one of several ‘activities’, each of which has a corresponding BT.

$$((Activity = Chat) ? Chat_{BT}) \rightarrow ((Activity = Clean) ? Clean_{BT}) \rightarrow \dots$$

The choice of which activity to perform is governed by a variable *Activity*. This structure seems natural, but may not be reactive, depending on the visibility of ‘Activity’. If the *Activity* variable is entirely internal to the system, where its value does not change transparently with regard to visible inputs, then this is a case of implicit memory, compromising reactivity. Alternatively, suppose the variable ‘Activity’ corresponds to an outwardly visible feature, such as the state of a physical button or display listing the robot’s current activity. Then, to the observer in our reactivity test, it is reasonable to say the robot is reactive to this external input—if the Activity button is changed, the activity being performed changes. Perhaps more clearly, consider this example:

$$((RobotinRoom1) ? Chat_{BT}) \rightarrow ((RobotinRoom2) ? Clean_{BT}) \rightarrow \dots$$

It is natural to assume that ‘which room the robot is in’ is an observable part of the sensor input, and an external observer would find the robot’s behavior to be reactive and intuitive.

This example was intended to illustrate that reactivity is not always clear-cut. Sometimes context is important.

This example was adapted from [5].

5.1.5 Nodes with memory

We have seen above a number of BT challenges where the problem is caused the lack of access of a BT to memory. In fact, we defined ‘reactive ASMs’ as a subset of all ASMs satisfying a particular condition, so it is immediate that there is a trade-off between expressiveness (in terms of constructing large classes of ASMs) and reactivity. Indeed, though we argue that reactivity is useful for constructing simple and trustable AI, we are unlikely to construct an architecture where every layer is reactive. In Section 7 we focus on the specifics of which selections cannot be constructed without memory. Given that memory is generally necessary, how should we incorporate it, keeping in mind our principles and goals?

Firstly, given that we aim to construct transparent architectures, memory use should be explicit and clear. From the perspective of the BT layer, the simplest way to achieve this is to encapsulate memory within individual actions, or in a higher layer controlling access to the BT. It can be more difficult for humans to understand memory use, but not impossible if it is kept minimal and cohesive.

Unfortunately even this clean separation can be difficult to achieve, as sometimes the BT layer must work closely with layers using memory. One way to do this transparently, suggested in the BT literature, is *control flow nodes with memory*.

Nodes with memory [5] are an extension of the usual BT framework designed to prevent unnecessary ticking in circumstances where reactivity is not desired. Two variants on the usual control flow nodes are introduced, called *Sequence and Fallback with memory*, and written \rightarrow^* and $?^*$. The Sequence node with memory ticks its children from left to right, but only until they return Success, after which it remembers that value without ticking the child again. The memory is cleared once the node with memory itself returns a value other than Running, and on the subsequent tick it begins again with the leftmost node. This is not reactive, but it does at least prevent uncontrolled access to memory. Its operation is quite readable and is still modular,

as it retains the tree structure and interface of BTs. The Fallback node with memory operates similarly.

By the modularity of BTs, we can think of the node with memory and its subtree as being a non-reactive layer beneath the BT layer. The node with memory enforces a specific structure on that lower layer, which allows them to be presented as one layer in a readable fashion. Used judiciously, such nodes are fairly innocuous (in section 7.4 we show that they do not greatly increase expressiveness however).

Note that a Sequence with memory

$$Action1 \xrightarrow{*} Action2$$

can be emulated by a normal Sequence by adding extra variables [5].

$$(Action1Done ? Action1) \rightarrow (Action2Done ? Action2)$$

This is equivalent so this version is still not reactive. However the additional variables constitute implicit memory, so the non-reactiveness has essentially become less explicit. The operation of the variables *Action1Done* and *Action2Done* is no longer defined in any specific part of the structure, unlike when using an explicit node with memory.

5.1.6 Styles

Styles are another BT extension designed to operate between layers of the architecture, and can be used to introduce memory.

Styles [21] are a BT extension designed to add more complex behavior to large BTs by modifying them in runtime. Specifically, the BT may be in one of several *styles*, each of which disables some subtrees of the tree. Otherwise, while in a given style, the BT operates as usual but simply ignores the disabled subtrees. In different styles the BT may react slightly differently. This was suggested as use for coding the behavior of groups of BTs in [21], where the individual BTs are influenced by the current style of the group. The switching between styles may itself be done by a BT, or another ASM like a FSM. If the switching between styles is itself governed by a reactive ASM, then the BT layer will still always be reactive. However if the styles are controlled by a non-reactive architecture, then the overall behavior can be non-reactive.

Styles are an interesting example because, though the BT is modified at runtime and so may react differently to the same input (violating reactivity), in any specific individual style none of the BT principles are violated. Styles are an example of an extension which operates on the layer *above* the BT layer, so the use of styles is just one way of implementing a layer above BTs. The advantage of using a style on this layer is that much of the information stored in the BT is reused with styles, reflecting the fact that we often only need variations on behaviors not entirely distinct behaviors. Styles thus, are useful where they are applicable, for improving transparency of the relationships between layers.

5.1.7 Decorators

So far, we have omitted mention of the Decorator node commonly used for BTs. This is largely because its extreme generality make it difficult to summarise, and must be analysed for particular Decorator examples. Here we very briefly discuss a few of these and apply some of the principles discussed above to their use.

Negation: Negation is a Decorator node which returns Running if its child returns Running, Success if its child returns Failure and Failure if its child returns Success. This does not violate any principles of BT use. If its child is reactive then it is reactive and it is modular as it preserves

the BT interface. What’s more, it is extremely simple and interacts well with the other operators (preserving the symmetry of Success and Failure and providing an analogue of De Morgan’s laws, as shown in [16]) making it very readable.

Run until Success or eternal memory: This Decorator ticks its child and returns the child’s return value until the child returns Success, after which the Decorator returns Success always. It is easy to see that as its behavior is dependent on past return values, it violates reactivity. However, its behavior is at least fairly simple and constrained as the reactiveness is only lost after the child returns Success, minimising the impact of this loss of reactiveness. We will refer to this Decorator by the name *eternal memory* (contrasted with control flow nodes with memory, whose memory is periodically reset), and we shall discuss its relevance to increasing the expressiveness of BTs in Section 7.4.

Run n times: This Decorator ticks its child n times, and after this point returns Success always. As before, it is not reactive, and can be more opaque than the previous, because the number of internal ticks does not necessarily correspond to any outwardly visible property.

5.2 Modularity

5.2.1 Optimal gameplay

Behavior Trees were introduced in game AI to control the behavior of Non-Player Characters. In that context, one desires behavior which is realistic and complex, and which may be required to compete against the player where necessary. How much of this computation should be contained in the BT layer?

To begin, consider this BT for playing the game Pac-Man.

$$(GhostClose \rightarrow AvoidGhost) ? EatPills$$

Of course, this produces fairly trivial greedy behavior, but it is only a very simple tree. Can we do better with a more complex tree? Can we construct optimal gameplay?

One approach is to check for more conditions, refining the partition of the state space induced by this reactive ASM.

$$(GhostScared \rightarrow Chase) ? (GhostVeryClose \rightarrow Avoid) ? \\ (GhostClose \rightarrow ((PowerPelletVeryClose \rightarrow GoToPellet) ? Avoid)) ? EatPills$$

This BT produces a slightly cleverer behavior, being able to decide whether to flee or to attempt to reach a Power Pellet on the basis of the distance to the nearest Ghost. However, it also does not produce optimal play.

One might be inclined to think, given the discussion above, that the limitations here are a result of the reactiveness of BTs, but in essence it is a limitation of their finite presentation. In fact, at least in theory, reactiveness is not limiting for this kind of AI. Pac-Man is discrete, and there are only a finite number of game states defined by the locations of all Ghosts, Pills and Pac-man. Further, the optimal move in Pac-Man only ever depends on the current world state—in other words, an optimal Pac-Man policy will be reactive.⁶ Suppose such an optimal policy was given to us. By constructing a tree with all possible game states, and associating each with its optimal *Up, Down, Left, Right* move from this policy, we can construct a BT as follows:

$$(GameState1 \rightarrow OptimalMoveState1) ? \dots$$

⁶In fact, a number of games which are difficult for AI to play have reactive optimal policies (possibly ignoring some rules). Examples include chess (ignoring the *en passant*, castling and the threefold repetition rules, which depend on past states) and Go (ignoring the ko rule).

Such a BT is both completely reactive and completely optimal, but also completely unrealistic. Firstly, the assumption of having an optimal policy given is very strong or, for most games, impossible. Secondly, constructing such a tree is well beyond the realm of feasibility for even games like Pac-Man, and worse for games of even slightly more complexity ⁷. Thirdly, and most importantly for this paper, this approach violates the *modularity* of BTs. As the specificity of the actions and conditions grow, their reusability reduces. The subtrees of the tree begin to correspond less and less to identifiable behaviors. It is understandable that there would be a trade-off between optimal behavior and transparency. Optimisation, being computationally intensive, makes it infeasible for an outside observer to predict the agent’s behavior any faster than that agent can calculate its own behavior. Striking a balance between these competing goals in robotics can be difficult. In general though, we suggest that where optimisation-based policies are required they should be separated into individual actions and implemented at a layer below the BT. In this way, the BT layer is still readable and modular, and adds reactivity on top of the optimal policies.

Another proposed solution to such cases is a BT extension called the *Utility node*. Utility nodes operate like a Fallback node, except that an internal optimisation rearranges the order of the children at runtime, in order of highest to lowest ‘utility’ in that circumstance. With this tool, we could in principle restructure the above optimal Pac-Man BT as the following:

$$Up? \overset{U}{Down} ? \overset{U}{Left} ? \overset{U}{Right}$$

where $?^U$ represents the Utility node. However, this BT is no more useful than the previous. Utility BTs need not necessarily be this opaque however, depending how they are used. Suppose an observer could inspect the ‘utility scores’ being generated at runtime. This approach could provide some readable insight into how the underlying optimisation takes place, while not necessarily being reactive. So long as BTs and actions can calculate their own utility scores internally (thus maintaining their conceptual cohesiveness), this approach can still be modular.

This Pac-Man example was adapted from [5].

5.2.2 BTs on atomic actions

Thus far, our discussion has been largely agnostic to how many layers of control exist above or below the BT layer. However the previous section noted that to construct BTs that display closer to ‘optimal’ behavior, the number of conditions required grows exponentially. The section concluded by arguing that an optimisation computation was better placed at a lower layer than the BT. In this section we dig further into this problem.

In principle, BTs expressiveness does not depend on the content of the actions that are given to it. These could be extremely complex behaviors, or extremely low-level signals. In practice though, it is often the case that the limitations of BT expressiveness, as discussed in Section 7 are far more apparent for low-level actions. A common example of such use is BTs for discrete gameplay [22–24] or BTs evolved from a finite set of atomic actions and conditions [24–26]. For example, consider a BT operating in a discrete grid world. Some obvious simple actions in this world might be Step Forward, Turn Left, Turn Right, Pick Up Object, etc. Alternatively, more ‘complex’ actions such as Explore, Go to Region X, Avoid Enemy, etc could be used. How should we decide which of these sets of actions is most appropriate for the BT to operate upon?

⁷As an extremely back-of-the-envelope calculation, a BT implementing the full game tree for Pac-Man would have about 10^{85} nodes. Omitting some detail, in Pac-Man there are 240 pills, $290 \approx 256 = 2^8$ squares on the standard game board, 4 ghosts and one Pac-Man. Each pill can be in two states, Pac-Man in 256 states, ghosts in $256 \times 3 \approx 2$ states, which gives $2^{240} \times (2^{8+1})^4 \times 2^8 = 2^{284} \approx 10^{85}$, and there are between 10^{78} and 10^{82} atoms in the observable universe.

Essentially, how many layers of the architecture should lie between the inputs from this discrete world and the BT layer?

To answer this question, we must first observe that in general the layers of the control architecture provide layers of abstraction, and corresponding reductions in the size of the information passed upwards. At the lowest layer, input data is largest in volume, and for real-world robots often close to continuous. On the other hand, the data past to the highest level is a highly abstract description of the input and lower layers. Each layer performs computation, which is a form of abstraction—it does not make sense for a layer to receive input data, process it, then output more data than it received.

This means that, even for a discrete world such as the example above, there is a state-explosion problem for the lowest layers. Suppose for the sake of argument that the input is given by the states of the world grid in the 5×5 square surrounding the agent, and that each square can be in one of 5 states. This gives 5^{25} possible inputs. A BT attempting to reason in such a world then will suffer from one of two problems. The first option is that it responds differently to large number of input states. This generally causes it to grow to exponential size, which is not modular, not readable, and largely impractical, as discussed in Section 7. It is also likely to violate our assumption that action selection took negligible time on the world time-scale, as selecting from this number of inputs could be computationally intensive. In addition, recall that when justifying why reactivity was desirable, we pointed out that it made behavior intuitive and easy to understand. If an ASM is reactive but has 5^{25} possible inputs, it is essentially impossible for an observer to discern this reactivity.

The other alternative for a low-layer BT is that it significantly abstracts the input space, by constructing a small number of conditions on this input. This allows BTs of reasonable size and reactivity to be built. However, this abstraction throws away a significant amount of information, which can no longer be used by the BT or any layer built above it. This greatly limits the reasoning that can be done. As a grid-world example, suppose we construct conditions Wall Left/Right, indicating there is a wall somewhere in one of the closest two squares to the left/right of the agent, but no other conditions exist for sensing Walls. If the condition Wall Left \wedge Wall Right is true, we have lost the ability to compute on which side the wall is closer to the agent.

What's more, the restrictions on expressiveness that are induced by the criteria of reactivity (discussed fully in Section 7) become more stringent at this level of atomicity. In fact, some simple higher-level behaviors are impossible to construct using only a reactive ASM (such as a BT) from atomic actions such as Turn Left and Step Forward. Consider the problem of wall-following in a discrete world. We want a behavior which, if placed next to a wall, can follow that wall until it returns to its starting point. While such a behavior can be constructed from an FSM, we cannot with a BT. Consider the following sequence of steps, where we assume the agent has a 5×5 field of vision with no other walls within this view. In this example, the agent is following the lower wall, when it reaches a passage. It passes through the passage (Steps 1-5) but upon reaching Step 5 the input is identical to the state of Step 1, from the agent's perspective. Thus, the agent returns and passes back through the passage, and so becomes caught in a loop. While this is not a proof, it is possible to construct this or similar arguments formally. We omit such an argument because the limitations on expressiveness of reactive architectures is covered in Section 7.

While FSMs can be used to construct a wall-following algorithm, many of the arguments in this section also apply to FSMs, and indeed FSMs are rarely used at the lowest levels of abstraction in practice. We would not use an FSM to perform robotic vision, to extrapolate from sensor data or to calculate a shortest path. We should therefore not aim to construct BTs for these purposes.

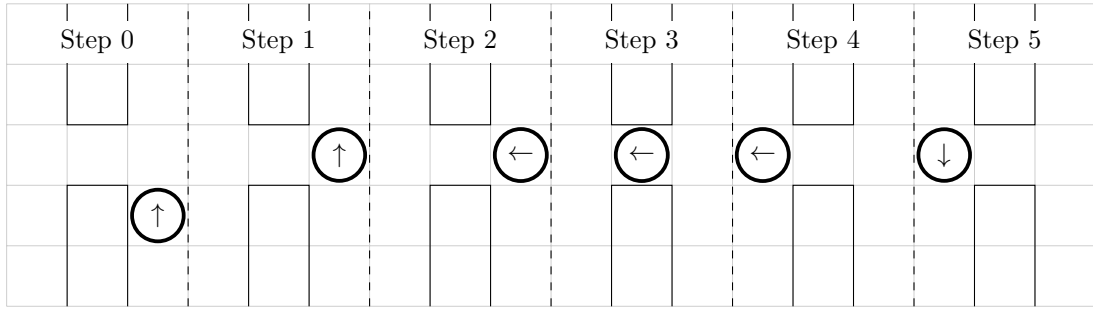


Figure 3: A reactive agent becoming stuck while attempting to follow a wall. Step 5 is identical to Step 1 from the agent’s perspective, so the agent turns back through the wall. The arrow denotes the direction the agent is facing.

5.2.3 Encapsulation

The points thus far have argued that we should use BTs where we can preserve their modularity and reactivity. However, we know [2, 5, 15] that these properties are also critical for Teleo-reactive programs and Decision Trees. How then should we decide between structuring a reactive layer as a BT or another of these reactive architectures?

One argument is based on expressiveness, and we explore this more fully in Section 7. There we show that though BTs, DTs and TRs all construct the same set of ASMs if conditions can be constructed freely, but if actions are fixed then BTs are strictly more expressive than TRs and DTs. This has been shown in [10, 27].

In practice however, the difference in expressiveness is not always sufficient in itself because it is generally straightforward to construct additional conditions. If TRs and DTs are arguably conceptually simpler than BTs, why would we use BTs? For instance, we can emulate a BT with a Decision Tree by representing action return conditions as explicit predicates in the DT, which are then linked up with appropriate branching. This produces an identical ASM, so how does this approach differ from a standard BT?

We postulate that the BT representation has the advantage of *encapsulating its own metadata*, in the form of return values. This aspect of BT’s modularity can be considered to take inspiration from the object-oriented programming paradigm. DTs use predicates on the current input to select the most ‘appropriate’ action. These predicates are often naturally related with specific actions, in every DT in which that action is used. Reusing an example from Section 5.1.2, consider the action Grasp Object. In choosing when to select this in a DT, the information of whether the robot is already grasping an object, or if there is no object to grasp, will almost always be relevant. It is for this reason that we associated these two conditions permanently to this particular action, by assigning them as the Success and Failure conditions. Given this information is naturally related to this action, it should be stored within that action so that the action can be easily reused. In a DT, if we explicitly express these Success and Failure conditions as predicates in the tree, they have become disconnected from their respective action. Because this information is not self-contained, when using the action Grasp Object in another DT, we must either store this relevant information somewhere or have it be lost. For BTs, this information is stored in the tree and accessed by the operators, allowing complex trees with complex return conditions to be derived and stored for later reuse. Recalling Section 5.1.2 on modelling return conditions, we conclude that to maximise reuse we should store as return values all the information that relates closely to that action.

5.2.4 Reuse

The findings of the above section suggest that we should associate all clearly related values with actions in order to maximise their ability to be reused—noting that doing so also maximises the ability of trees derived from these actions to be reused, in accordance with the principle of modularity.

However, consider a Teleo-reactive program, which itself is a modular [15] control architecture. TRs, like BTs, encapsulate data associated with actions to some degree, by assigning each action a precondition. When comparing BTs to TRs, as done in [5, 27], this condition is usually considered as either a Success or Failure condition. The important thing to note is that in general it is perfectly reasonable to interpret it in either sense. In addition, recall that in Section 5.1.2 we argued that not all actions have a natural sense of Succeeding or Failing. The resultant point is that we should not assign any particular meaning to Success and Failure. Instead, like the precondition of an action in a TR, they are better thought of as two useful pieces of metadata which can be accessed for the purposes of action selection. The BT framework is set up in general to emphasise symmetry between these two values. This symmetry is useful and makes reasoning about BTs intuitive, which we argue means that we should not assign any particular meaning or criterion to one value but not the other.

By doing this, we are allowing for increased reuse of actions and behaviors across multiple architectures. For instance, actions with preconditions used in TRs can be used without modifications in BTs, simply by interpreting the precondition in a fixed way as Success or Failure. Likewise, an action which never returns either Success or Failure can be used in any of BTs, TRs or DTs. This means in addition that TRs and DTs can be used as actions within BTs, for instance by using the translation of [10] (nesting ASMs is discussed formally in [15]).

We can further improve modularity by removing all reliance on the ‘Running’ return value. Instead, we assume as in [15], that there exists a return value function attached to each action, which gives a value in any input. For an action in a BT, two of these values may be interpreted as Success and Failure, but there could be any number of other values. We shall now interpret ‘Running’ for this action as *any return value not handled by the structure*⁸. A priori, nothing has changed with regard to the operation of this action in a BT, because the tree responds appropriately to the values Success and Failure and any other values are interpreted as Running. However, this definition allows any action to be used in any of these architectures—we no longer need to make assumptions about the number of provided return values. For instance, in a decision tree any value is interpreted as Running, as the tree does not handle any return values. Similarly, we could use any action in a TR, by choosing a value to represent its precondition and ignoring all other values returned. This now allows BTs to be used as subtrees in DTs, or as individual actions within a TR, maximising the ability to reuse complex behavior. By assigning return values in this way, we allow actions to be stored with their metadata independent of which architecture(s) they will eventually become a part of.

This requires something of a mental shift from the usual BT interpretation, because the somewhat loaded terms Success and Failure suggest that that action cannot be selected as it has already ‘Succeeded’ or ‘Failed’. However, as we already saw in Section 5.1.2, we shouldn’t think of these values in the past tense as following the ‘completion’ of an action. In a reactive architecture then, these values may not correspond to intuition. As in the example action Grasp Object, it is not *unreasonable* to attempt to grasp an object while already holding one, it is merely *trivial*; the resultant action has no effect. The condition that we are already holding an object, which we

⁸In some early forms of BTs there was no Running return value [28], and instead actions executed one by one until they returned Success or Failure. That is not reactive, and very different from what we mean here. We assume there is at least one other value that can be returned, and that a value is returned at every time step. We are simply allowing information to be passed that would otherwise be hidden by a ‘Running’ value.

assigned as the metadata associated with the value ‘Success’, is used to determine that in this situation it may be desirable to select a different action. In defining BTs as ASMs in Section 3, we already allowed for actions to be selected even if they returned a non-Running value because this allowed ASMs to be composed. Here we have extended this idea.

To cap off this discussion, we should recall the result of the previous section, which is that we should assign to an action precisely those return values that are naturally associated with it in all contexts. These may or may not directly correspond to Success and Failure. The results here have shown that we can still apply that action to all of these architectures. Further, all of those architectures can be used as actions within each other without modification. If for instance we construct a TR, but then later decide to embed this as a subtree in a BT, this can easily be done, and vice versa.

6 Generalising Behavior Trees: the k -BTs

The previous two subsections allowed us to conclude that we should assign all appropriate return values to actions, to maximise their modularity. However, BTs can only ever access two of these values, and TRs and DTs can only access one or zero respectively. In this section we show how a simple generalisation of BTs extends their benefits to a broader class of ASMs which can interpret any number of return values. We call these the generalised Behavior Trees or k -BTs and we show, as in the previous section, that BTs can be used without modification as subtrees of k -BTs. Further, the normal BTs are exactly the 2-BTs and the TRs are the 1-BTs. Finally, we show how k -BTs provide natural solutions to some of the examples given in Section 5.

To introduce these objects, first recall that BTs are trees with two control flow nodes, corresponding to the specific values Success and Failure (note that we are omitting the Decorator and Parallel nodes, for which analogous constructions could be made if necessary). The k -BTs will be defined as trees with k control flow nodes. We will choose k return values, which we shall write as the values $\{1, \dots, k\}$, and we shall associate a control flow node with each. Consistent with the previous section, there is no explicit ‘Running’ value.

Formally, we define the k -BTs as follows. We will denote the k control flow nodes by $*_1, \dots, *_k$. Fix an ordered rooted tree with ℓ leaves labelled by actions and internal nodes labelled by these control flow nodes. When a control flow node $*_i$ is ticked, it ticks each of its children from left to right, until some child returns a value other than i , at which point that control flow node returns that same value. As an ASM, each operator $*_i$ is defined by the following rules. If $*_i$ has one child α , select that child. Otherwise, for an input $x \in \Sigma$, if $*_i$ has children c_1, \dots, c_n ,

- Begin at c_1 . Select c_j if c_j does not return i in x .
- Otherwise, repeat for c_{j+1} .
- Select c_n if no previous are selected

The definition of the overall ASM is built by recursively combining these, in exactly the same way that BTs are built from the definitions of Sequence and Fallback.

Note now that if only two control flow nodes are used (which we assume for simplicity correspond to values 1 and 2) we can interpret them as Sequence and Fallback and determine that the 2-BTs are exactly the normal BTs. Similarly, if the value 1 (or any fixed value) corresponds to the precondition of an action being false, then the 1-BTs are exactly the TRs. Hence we have constructed a family of modular reactive control architectures which include BTs and TRs. In fact, it is easy to see that the k -BTs inherit the reactivity, modularity and readability of BTs as they inherit BT’s intuitive presentation as a tree. Finally, note that the ‘ k ’ in k -BTs

is a placeholder, which can be replaced with the 2-BTs or 3-BTs as above, but when left unspecified we assume k is any fixed finite value.

6.1 Use cases for k -BTs

Since the 2-BTs are the standard BTs, k -BTs are clearly applicable for any situation where BTs are used. The previous sections have also established that the existence of k -BTs is at least a consistent consequence of modularity, reactivity and readability. Here we show that the k -BTs are practically useful, in that they provide a natural way of extending BTs in certain cases where they are otherwise unwieldy. As examples, we will show here how k -BTs provide a possible solution to the challenges of Sections 5.1.2 (Modelling Success and Failure), 5.1.3 (Knowledge Versus Memory).

The way that Sequence and Fallback handle the values Success and Failure from their children can be thought of in terms of exceptions in programming. These values then are two distinct exceptions which can be thrown by a program, with Sequence and Fallback acting as code blocks which catch and process the respective errors. Adding further control flow nodes is no different than catching more exceptions. The ‘thrown error’ analogy provides insight to an obvious use case for k -BTs: failure handling. In [5], the authors argue that failure handling is easier in BTs than TRs, and significantly easier than in DTs, as more return values can be handled. The k -BTs make this yet more streamlined, essentially by allowing for numerous distinct error codes which can be handled differently. Failure conditions can now be broken into any number of natural failure cases and assigned distinct return values. Of course, as we do not consider either Success or Failure to have inherent meaning, the additional return values can equally be interpreted as different types of Success.

These values can also be thought of as degrees of Success (or Failure). For instance, consider a BT controlling an agent with an action which succeeds when some number of items are in a goal region. In such a case, it could be useful to have metadata storing a ‘partially successful’ value if some number are within the goal region. This same idea could be extended to risk, with various return values indicating the probability of successful or unsuccessful completion. This type of reasoning is useful for tasks where overall Success/Failure is unusual or not critical, and where a ‘reasonable’ performance is acceptable.

We need not solely interpret the additional values as variants of Success and Failure. For example, consider the discussion in Section 5.1.3, where a robot action Unlock Door returns Success if the door is unlocked and Failure if there is no key. However, its action will depend on whether the knowledge of whether the door is locked is stored within its representation of the world. In this case, we could add an additional value ‘Unknown’, which is returned when the robot is uncertain about the status of the door. This can be either handled in the tree, allowing us to not execute this action if the robot is uncertain, or ignored in the tree, in which case it is treated as ‘Running’ and the action is still selected. In both cases, ‘Unknown’ is not merely a variation on Success or Failure but a concept which cannot be obviously defined as either.

7 Expressiveness of Behavior Trees

In this section we explore the *expressiveness* of BTs. Here we would like to know, not how a given BT should operate, but whether we can construct BTs to solve various computational problems on the BT layer. In the previous sections we saw that preserving the reactivity and modularity of BTs imposes limitations on which problems can be solved with BTs. The question now arises: what can we express as a BT without violating these principles? We show that there

exists a trade-off in BTs between reactivity and expressiveness and we describe the nature of that trade-off.

7.1 Comparing by selections

We must first answer how we can measure the expressiveness of a given layer of the architecture. As the previous sections argue, BTs should not be used in isolation to construct complex architectures—at least the lower layers should be implemented in a different form. We cannot therefore simply compare the overall output of the systems, because the actual output is dictated by the actions, which are implemented at other layers to the BT itself. In such a case, every possible architecture of arbitrary complexity could be implemented by a trivial BT. Specifically, one could add a trivial top layer consisting of a BT with a single action, where that action contains the entire behavior of the agent and is always selected by the BT. This tells us nothing at all about what decision-making can and cannot be constructed by BTs.

Instead, we must compare the input and output of a single layer. ASMs will provide the formalisation to make this possible.

Definition 7.1. Let X and Y be classes of control architectures, such as FSMs or BTs, and let X_{ASM} and Y_{ASM} be their sets of derived ASMs. We say X is *more expressive than/at least as expressive as* Y if $Y_{ASM} \subset X_{ASM}/Y_{ASM} \subseteq X_{ASM}$. We say X and Y are *equivalently expressive* or simply *equivalent* if $X_{ASM} = Y_{ASM}$.

We shall from here use boldface to denote sets of ASMs. For instance, **BT** shall denote the ASMs derived from BTs, and **TR**, **DT**, **FSM** are defined similarly. We shall also write **Reactive** for the set of all reactive ASMs, and **Constant**, for the set of all ASMs that are constant functions.⁹

This idea of expressiveness is consistent with and formalises the work of other authors in this area. In [10,27] it is proven that BTs are at least as expressive as TRs and DTs, and their argument extends to this ASM definition, they construct a BT with the same ASM for each DT/TR. In addition, in [15] it is shown that a graph-structured reactive architecture called a *decision structure* generalises all the k -BTs, in this sense. We shall write the corresponding class of ASMs as **DS**. One can observe easily that DTs generalise TRs, by constructing a tree of the form ‘**if** k_1 **select** a_1 **else** (**if** k_2 **select** a_2 **else** (...))’. Similarly, the argument given in [3,5] that the FSMs can express BTs extends to this definition. In fact, this argument allows FSMs to construct any reactive ASM, by constructing a complete directed graph with all arcs into a given node labelled by the preimage of that action in the reactive architecture. Finally, it is straightforward to show that the 2-BTs and 1-BTs have the same ASMs as the BTs and TRs respectively, and that all of the aforementioned architectures generalise the constant ASMs. Note that this relation is transitive as it is derived from a subset relation.

From these arguments, we can immediately present a number of arguments from a variety of sources in a single hierarchy.

Theorem 7.2 (The Weak ASM Hierarchy).

$$\underline{\mathbf{Constant} \subseteq \mathbf{TR} \subseteq \mathbf{DT} \subseteq \mathbf{BT} \subseteq \mathbf{3-BT} \subseteq \dots \subseteq \mathbf{k-BT} \subseteq \dots \subseteq \mathbf{DS} \subseteq \mathbf{Reactive} \subseteq \mathbf{FSM}}$$

⁹One can conceive yet more computationally powerful ASMs, for instance corresponding to Turing Machines, by exploiting the close relationship between ASMs and models of computation. For instance, one can show that an ASM can be represented by an FSM if and only if the preimage of every action in \mathcal{A} is a *regular* language, which are the languages recognisable by *deterministic finite automata*. Such abstractions will not be necessarily for the arguments we make here, however.

Note that all the sets are related by non-strict subset relations, because while we have proved each generalises its predecessor we have not stated any proofs that this relationship is strict. Now we shall improve this hierarchy with some such results.

Lemma 7.3. Constant \neq TR.

This can be trivially shown by the fact that any TR which can select more than one action has a non-constant ASM.

Lemma 7.4. Reactive \neq FSM.

This is again trivial, as not all FSMs are reactive. An example is an FSM with two states and a single transition. Finally, we can show that all reactive architectures in fact recognise the same sets of ASMs.

Lemma 7.5. Reactive = TR.

Proof. We have not prohibited the construction of arbitrary conditions (which are predicates $\Sigma \rightarrow \{True, False\}$), so long as implicit memory is not used. Specifically, given any reactive ASM $r : \Sigma \rightarrow \mathcal{A}$, $\mathcal{A} = \{\alpha_1, \dots, \alpha_n\}$, the TR

$$\begin{aligned} x \in r^{-1}(\{\alpha_1\}) &\rightarrow \alpha_1 \\ x \in r^{-1}(\{\alpha_2\}) &\rightarrow \alpha_2 \\ &\vdots \\ x \in r^{-1}(\{\alpha_n\}) &\rightarrow \alpha_n \end{aligned}$$

has the same derived ASM. □

This result gives us the following stronger hierarchy.

Theorem 7.6 (The ASM Hierarchy).

$$\mathbf{Constant} \subset \mathbf{TR} = \mathbf{BT} = \mathbf{Reactive} \subset \mathbf{FSM}$$

In some sense this result is not very satisfying, as we would like to separate the reactive architectures into a useful hierarchy, rather than collapse them. We do exactly this with a stronger form of generalisation in Section 7.3, but we first digress to discuss FSMs.

7.2 BTs versus FSMs, with insights from structured programming

In this section we discuss further the relationship between BTs and FSMs, beginning with an immediate corollary of the results of the previous section.

Corollary 7.7. Finite State Machines are more expressive than BTs.

Unsurprisingly, if implicit memory is used, the above result does not hold, as BTs are no longer reactive. In fact, if implicit memory is allowed, one can construct BTs which are as expressive as FSMs. The following result was proved in [3].

Lemma 7.8. If a finite amount of Boolean implicit memory is allowed, then all the ASMs corresponding to FSMs can be derived from BTs.

Proof. Using auxiliary variables, keep track of the state of the FSM as the input is processed, and construct a BT which checks these auxiliary variables and selects the appropriate corresponding action. That is, the tree consists of a Fallback node each of whose children is of the form $FSM_in_State_i \rightarrow Transition_FSM_Appropriately \rightarrow do_Action_i$. \square

Within the broad analogies with structured programming we have explored in this paper, this result specifically has many interesting parallels.

In 1966, Böhm and Jacopini [29] published a result proving that all programs could be constructed as programs consisting of only while loops and if statements, assuming tests and assignments on some auxiliary variables were allowed. This Theorem was later shown to have an essentially trivial proof in [30] (whose roots lay in earlier work of Kleene and von Neumann [31]), where auxiliary variables store the state of the original program and a single enclosing while loop repeatedly selects the corresponding code blocks and updates the auxiliary program-state variables accordingly.

This trivial form of Böhm-Jacopini Theorem exactly matches how the BT emulates the FSM in the proof above. The repeated tick generation at the root corresponds to the outer while loop, with the additional variables storing the FSM state and selecting the appropriate action.¹⁰

This Theorem was widely cited by supporters of the structured programming paradigm as an argument for the abolition of **go tos**. It is interesting to note however that this theorem did not itself motivate structured programming, as the movement generally traces its roots to the Dijkstra’s letter [9], published two years later in 1968. To quote Knuth, “from a practical standpoint [the Böhm-Jacopini Theorem] is meaningless ... we have eliminated all the **go tos**, but we’ve actually lost all the structure” [14]. Dijkstra, who cites this result as a theoretical justification for structured programming, himself points out that “the exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one” [9]. Indeed, in their original paper, the Böhm and Jacopini conjectured, and Floyd and Knuth later proved [32], that the auxiliary variables form a critical part of this proof—there are some programs which cannot be ‘structured’ without additional variables.

Lemma 7.8, originally proved in [3] and reproved in [5, 10, 18], has similarly been used as a theoretical justification for structuring control architectures as BTs. Our aim in this paper is to outline the principles of BT use, and to identify when an action selection mechanism can be structured as a BT in a way consistent with these principles, in the same vein as [14] discusses **go tos**. We have shown already that implicit memory makes the BT layer less readable and less modular, and so we believe it should be avoided. The conclusion of this section should therefore be that BTs and FSMs *are not* the same in terms of expressiveness, because Lemma 7.8 does not reflect how we use BTs. As a final appeal to authority, we again quote Knuth: “The underlying structure of the program is what counts, and we want only to avoid usages which somehow clutter up the program. ... The real goal is to formulate our programs in such a way that they are easily understood.” [14]

7.3 Comparing by structure

In this section we introduce the definition of what we call *structural equivalence*, which will allow us to separate the reactive architectures by their expressiveness.

¹⁰In fact, this can recreate the entire proof of the Böhm-Jacopini Theorem, and use BTs to simulate ASMs which are even more complex than FSMs (see the previous footnote). However, as we argue, this is not necessarily a useful way of thinking about BTs as it goes against the principles of BTs we have set out in this paper. In addition, the full expressive power of BTs is not needed, as TRs can also be used to construct the same proof.

Considering the previous discussion, one may note that the result of Lemma 7.5 similarly deconstructs given reactive architecture in order to have a TR emulate it. Instead of additional memory, this used the weaker assumption that additional actions (in this case in the form of conditions) could be constructed as necessary. This was sufficient to distinguish between reactive and non-reactive architectures, but consistent with the goal of formulating understandable structures we would prefer a definition which can distinguish reactive architectures and which preserves structure.

To begin, note that all the control architectures we have so far discussed have been built from ‘nodes’ labelled by actions. This includes BTs, k -BTs, DTs, TRs and FSMs¹¹. The ‘nodes’ are precisely the places where actions are applied; for BTs and k -BTs this is the leaves, for FSMs and DTs this is any node in the structure and for TRs this is an entry in the list. The size of a specific such architecture will be the number of ‘nodes’, for instance, the number of leaves in a BT. By an *unlabelled* architecture we mean the structure with blank node labels, such as a BT with blank leaves.

Definition 7.9. Let X_1 and X_2 be unlabelled control architectures, both of size n . We say X_1 and X_2 are *structurally equivalent* if we can order both sets of nodes so that given any list of n actions $\alpha_1, \dots, \alpha_n$, applying these actions to the nodes in those orders gives the same derived ASM.

As an example using BTs, the unlabelled trees $(\circ \rightarrow \circ) \rightarrow \circ$ and $\circ \rightarrow (\circ \rightarrow \circ)$ are structurally equivalent as whenever we replace the leaves \circ by actions from left to right the two trees give the same ASM (this means the associativity of \rightarrow and $?$ are *structural properties* of BTs, and in fact they are really the only structural properties—see Lemma 7.12). Essentially, two trees are structurally equivalent when they produce the same ASM regardless of which actions are chosen to label the tree. As an example of two trees not structurally equivalent, consider $(a \rightarrow b) ? b$ and $(a ? b) \rightarrow b$. These both give the same ASM as written for any a and b , but in general each tree $(\circ \rightarrow \circ) ? \circ$ and $(\circ ? \circ) \rightarrow \circ$ can be labelled by any three actions a, b, c , and in general $(a \rightarrow b) ? c$ and $(a ? b) \rightarrow c$ are not the same as ASMs. The same ideas hold for the other architectures.

Now, we will compare architectures based on structure, using a stronger form of expressiveness. In the following we use calligraphic font to denote unlabelled structures, so for instance \mathcal{BT} represents the set of unlabelled BTs.

Definition 7.10. Let \mathcal{X} and \mathcal{Y} be sets of unlabelled control architectures, such as unlabelled BTs or unlabelled DTs. We say \mathcal{Y} *strongly generalises* \mathcal{X} if for every $x \in \mathcal{X}$ there exists a structurally equivalent $y \in \mathcal{Y}$. We use $\mathcal{X} \prec \mathcal{Y}$ to mean ‘ \mathcal{Y} strongly generalises \mathcal{X} , but \mathcal{X} does not strongly generalise \mathcal{Y} ’.

Note that if \mathcal{Y} strongly generalises \mathcal{X} , then $\mathcal{X}_{ASM} \subseteq \mathcal{Y}_{ASM}$, where \mathcal{X}_{ASM} and \mathcal{Y}_{ASM} are the sets of derived ASMs constructed by applying all possible sequences of actions to label the structures in \mathcal{X} and \mathcal{Y} . Hence we know that at least as strong relationships must hold between sets of ASMs for strong generalisation as for generalisation—that is, if two sets are properly distinct in the first hierarchy, such as FSMs and BTs, then they must be distinct under strong generalisation. However, this will now allow us to separate the reactive architectures (except for decision trees) as the following hierarchy.

Theorem 7.11 (The Reactive Hierarchy).

$$\mathcal{TR} \prec \mathcal{BT} \prec \mathcal{3-BT} \prec \dots \prec \mathcal{k-BT} \prec \dots \prec \mathcal{DS} \prec \mathcal{Reactive}$$

¹¹This also includes *decision structures*, a graph-structured control architecture introduced in [15]. We will only mention them in passing here as we provide a full treatment in the concurrently-written paper [15]

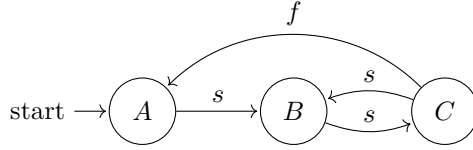


Figure 4: Finite State Machine M

Proof. Follows from Theorem 7.2 and the results of [15]. Most follow from the fact that the k -BTs strongly generalise the n -BTs for any $k > n$. \square

Decision Trees no longer fit cleanly into this hierarchy, because unlike the other architectures they cannot have any actions applied to their nodes. Instead, the internal nodes must be conditions. This makes it trickier to compare directly with the other sets. Greater discussion is provided in [15], where it is proved that $\mathcal{DT} \prec \mathcal{DS}$, but the set is otherwise incomparable to the other classes.

The definition of structural equivalence and this resultant classification of reactive architectures again has analogous grounding in structured programming. In [33], Kosaraju analyses differences in expressiveness between the programs constructible by different programming constructs, and so constructs hierarchies of classes of structured programs. This is analogous to what we have done here with control architectures. In order to compare such programs by their *flowcharts* (the graph structure of their code blocks), Kosaraju states that “we divorce ourselves from problems involved with specific data types and primitive instructions ... [and] concentrate on flow chart schemas where primitive actions and predicates are represented by **uninterpreted** symbols”. In fact, his notion of *weak reducibility* of flowcharts corresponds directly to our definition of *strong generalisation*.

7.4 Do memory nodes increase expressiveness?

In the previous section we showed that BTs are less expressive than FSMs, if we obey the principles of BT use and use the standard interpretation of the Sequence and Fallback nodes. However we discussed in Section 5 some extensions which are used to overcome these limitations on expressiveness, seemingly in a less damaging way than implicit memory. For instance, ‘Control flow nodes with memory’ and Decorator nodes implementing what we call ‘eternal memory’ are both sometimes used as simple extensions which violate reactivity by manipulating memory, but do so in a controlled and readable way. So, do these additions allow us to increase expressiveness to at least the level of an FSM? The answer, as we will show, is negative—BTs using these tools are still strictly less expressive than FSMs.

We will show this by constructing an FSM whose derived ASM cannot be realised by a BT, even with the addition of control flow nodes with memory and eternal memory Decorators. This proof is informally presented in depth because it provides insight into the limitations of how these concepts access memory.

Now consider the FSM M shown in Figure 4. Let T be some BT, which may contain control flow nodes with memory and eternal memory Decorators. Firstly, note that there cannot be any inputs for which both B and C return Success, and no input for which A and B return Success and C returns Failure, as in both cases the FSM does not make a selection. Otherwise, as A , B and C can be any actions, we can assume fairly that there exist inputs for which all other combinations of return values of A , B and C are returned.

We will write inputs $x \in \Sigma$ as $x = (v_A, v_B, v_C)$ where v_A , v_B and v_C are the return values of A , B and C respectively. There is no loss of generality by assuming this, as if there were any pair of inputs $x_1, x_2 \in \Sigma$ for which the return values of A , B and C are identical, then clearly $M_{ASM}(x_1) = M_{ASM}(x_2)$ and so if $T_{ASM}(x_1) \neq T_{ASM}(x_2)$ then T does not realise the same behavior as M . We shall write s and f to represent returning Success and Failure respectively, and r to represent returning any other value (thus returning ‘Running’ in the usual BT interpretation).

Consider the input sequence

$$z = (r, r, r), (s, r, r), (r, s, r), (r, f, s), (r, s, r), (r, r, f)$$

to which M selects

$$y = A, B, C, B, C, A$$

Eternal memory: Suppose T contains eternal memory. Thus once some subtrees return Success or Failure, they are never ticked again. Suppose we apply the input sequence z^n , which is z repeated n times. Eventually, after some large enough n to tick the whole tree, all eternal memory subtrees either always return Success or Failure or are never ticked in subsequent inputs of the sequence z . Then, let $m = n + 1$. In the last six inputs of z^m (which is the last copy of z), T must behave as if it contained no eternal memory.

Nodes with memory: Now we focus our attention to the last six inputs of z^m . Suppose T matches the first four outputs of this sequence. However, in the fourth input (r, f, s) , T must select B , which returns Failure in that state. As a result the top-level tree returns Failure, and so all subtrees which had been ticked are reset. Then for the subsequent input (r, s, r) T must select A , as BTs without memory always select their leftmost node if it runs, and by the selection of A on the first input this must be A (recall that the eternal memory has no effect). This differs from M , so T cannot realise the same ASM.

We know that preserving reactivity while increasing expressiveness is not possible, and so a trade-off between reactivity and expressiveness always exists. Now we see that these BT extensions, which come at the cost of reactivity, do not greatly increase expressiveness. This does not rule out the possibility of other (non-reactive) BT extensions increasing expressiveness to the level of FSMs, but it suggests that such constructions will not be straightforward.

7.5 How many BTs can we build?

In Section 7.3 we introduced the concept of *structural equivalence*, as a way of comparing between reactive architectures. In this context we saw that BTs were limited within the class of reactive ASMs. To gain an understanding of how expressive BTs are, we now enumerate the BTs up to structural equivalence. This both lends intuition to which reactive ASMs are realisable by BTs and provides consequences for research involving generating or evolving BTs automatically, such as [24–26], where knowing how many non-equivalent structures are possible provides insight into the proportion of those that have been constructed. The following Lemma provides a useful insight into structural equivalence within BTs (and k -BTs).

Lemma 7.12. Every k -BT is structurally equivalent to exactly one in *compressed form*, where every internal node has at least two children and all the children of an internal node are either leaves or are different control flow node than their parent. For BTs this means each tier of the tree alternates containing only Sequence and Fallback operators.

Proof. Firstly, we show there exists a structurally equivalent tree in compressed form. Let T be our original BT. If the root of T has one child, and let T' be the single subtree. Both have the

same number of nodes, and applying actions from left to right gives the same ASM as the root operator has no effect, thus they are structurally equivalent. Now suppose the root is a Sequence node (Fallback argument is identical), and one child tree c also has root node Sequence. Let T' be the tree where c is replaced under the root node by all of c 's children. The first of the children is ticked under the same conditions in T and T' , and the others are ticked in each case after the previous returns Success. Thus we observe T and T' are structurally equivalent. Repeating this process inductively gives a structurally equivalent tree in compressed form. Now we show this is unique. Suppose two compressed-form trees A and B are structurally equivalent. If A and B both have one leaf then they are identical. Suppose now the leaves are a_1, \dots, a_n and b_1, \dots, b_n respectively. Assume that for all compressed-form trees of up to $n - 1$ leaves, if they are structurally equivalent then they are identical. Assume without loss of generality that the root node of A is Sequence, with children A_1, \dots, A_k (the case for Fallback, or any of the k nodes for a k -BT, is identical to this). If any of these subtrees return Success, the next subtree is ticked, and if any other value is returned the root returns that value. By structural equivalence, we can choose sequences of actions which always select any particular subtree in A , and these must always select the same subtree in B . Thus the equivalent subsets B_1, \dots, B_k also have the property that when an action within B_i returns Success, the next node ticked is either within B_i or is the leftmost node in B_{i+1} . If nodes in B_i return other values we never tick anything subsequently that is not in B_i , as the same input into A does not leave the corresponding subtree. B is in compressed form, so we conclude the root node is a Sequence and each of these subsets is a subtree of that Sequence. Finally, as each subtree B_i is structurally equivalent to A_i , they are identical, and so A and B are identical. \square

Theorem 7.13. *The number of non-structurally equivalent BTs with n leaves is counted by the big Schröder numbers, the first few terms of which are 1, 2, 6, 22, 90, 394, 1806, ... (sequence A006318 in the OEIS [34]).*

Proof. By the above Lemma, it is sufficient to count unlabelled compressed form BTs. There is one unlabelled compressed form BT. The little Schröder numbers s_n are known to count the ordered rooted trees with n leaves and no internal node of degree less than two. For each such tree with $n > 1$, there are precisely two unlabelled compressed form BTs, one with root \rightarrow and the other with root $?$. The big Schröder numbers are twice the little Schröder numbers for $n > 1$. \square

References

- [1] R. Brooks, "A robust layered control system for a mobile robot," *IEEE journal on robotics and automation*, vol. 2, no. 1, pp. 14–23, 1986.
- [2] N. Nilsson, "Teleo-reactive programs for agent control," *Journal of artificial intelligence research*, vol. 1, pp. 139–158, 1993.
- [3] P. Ogren, "Increasing modularity of uav control systems using computer game behavior trees," in *Aiaa guidance, navigation, and control conference*, 2012, p. 4458.
- [4] I. Bojic, T. Lipic, M. Kusek, and G. Jezic, "Extending the jade agent behaviour model with jbehaviourtrees framework," in *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*. Springer, 2011, pp. 159–168.
- [5] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018.

- [6] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, “A survey of behavior trees in robotics and ai,” *arXiv preprint arXiv:2005.05842*, 2020.
- [7] J. A. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, *et al.*, “An integrated system for autonomous robotics manipulation,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 2955–2962.
- [8] A. Champandard, “Understanding behavior trees,” *AiGameDev. com*, vol. 6, p. 328, 2007.
- [9] E. W. Dijkstra, “Letters to the editor: go to statement considered harmful,” *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.
- [10] M. Colledanchise and P. Ögren, “How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees,” *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, 2017.
- [11] A. Klöckner, “Behavior Trees for UAV Mission Management,” in *Informatik angepasst an Mensch, Organisation und Umwelt*, vol. P-220, 2013, pp. 57–68.
- [12] D. Hu, Y. Gong, B. Hannaford, and E. J. Seibel, “Semi-autonomous simulated brain tumor ablation with ravenii surgical robot using behavior tree,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 3868–3875.
- [13] E. W. Dijkstra *et al.*, “Notes on structured programming,” 1970.
- [14] D. E. Knuth, “Structured programming with go to statements,” *ACM Computing Surveys (CSUR)*, vol. 6, no. 4, pp. 261–301, 1974.
- [15] O. Biggar, M. Zamani, and I. Shames, “On modularity in reactive control architectures, with an application to formal verification,” *Unpublished*, 2020.
- [16] O. Biggar and M. Zamani, “A framework for formal verification of behavior trees with linear temporal logic,” *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 2341–2348, 2020.
- [17] E. F. Moore *et al.*, “Gedanken-experiments on sequential machines,” *Automata studies*, vol. 34, pp. 129–153, 1956.
- [18] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, “Towards a unified behavior trees framework for robot control,” in *Proceedings of International Conference on Robotics and Automation*. IEEE, 2014, pp. 5420–5427.
- [19] B. Hannaford, “Hidden markov models derived from behavior trees,” *arXiv preprint arXiv:1907.10029*, 2019.
- [20] C. I. Sprague and P. Ögren, “Adding neural network controllers to behavior trees without destroying performance guarantees,” *arXiv preprint arXiv:1809.10283*, 2018.
- [21] D. Isla, “Handling complexity in the halo 2 ai, 2005,” *URL: http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml [21.1. 2010]*, 2015.
- [22] M. Colledanchise, R. N. Parasuraman, and P. Ogren, “Learning of behavior trees for autonomous agents,” *IEEE Transactions on Games*, 2018.

- [23] 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*, vol. 9, no. 3, pp. 227–238, 2016.
- [24] D. Perez, M. Nicolau, M. O’Neill, and A. Brabazon, “Evolving behaviour trees for the mario ai competition using grammatical evolution,” in *European Conference on the Applications of Evolutionary Computation*. Springer, 2011, pp. 123–132.
- [25] H. J. Christensen and J. Hoff, “Evolving behaviour trees: Automatic generation of ai opponents for real-time strategy games,” Ph.D. dissertation, Master’s Thesis, Norwegian University of Science and Technology, Trondheim . . . , 2016.
- [26] C.-U. Lim, R. Baumgarten, and S. Colton, “Evolving behaviour trees for the commercial game defcon,” in *European conference on the applications of evolutionary computation*. Springer, 2010, pp. 100–110.
- [27] M. Colledanchise and P. Ögren, “How behavior trees generalize the teleo-reactive paradigm and and-or-trees,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 424–429.
- [28] M. Mateas and A. Stern, “A behavior language for story-based believable agents,” *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 39–47, 2002.
- [29] C. Böhm and G. Jacopini, “Flow diagrams, turing machines and languages with only two formation rules,” *Communications of the ACM*, vol. 9, no. 5, pp. 366–371, 1966.
- [30] D. C. Cooper, “Böhm and jacopini’s reduction of flow charts,” *Communications of the ACM*, vol. 10, no. 8, p. 463, 1967.
- [31] D. Harel, “On folk theorems,” *Communications of the ACM*, vol. 23, no. 7, pp. 379–389, 1980.
- [32] D. E. Knuth and R. W. Floyd, “Notes on avoiding “go to” statements,” *Information processing letters*, vol. 1, no. 1, pp. 23–31, 1971.
- [33] S. R. Kosaraju, “Analysis of structured programs,” *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 232–255, 1974.
- [34] OEIS Foundation Inc., “Large Schröder numbers (or large Schroeder numbers, or big Schroeder numbers).” *The On-line Encyclopedia of Integer Sequences*. [Online]. Available: <https://oeis.org/A006318>