Behavior Trees: Breaking the Cycle of Misuse

Bobby Anguelov

1 Introduction

The Behavior Tree (BT) [Millington 09] is one of the most popular and widely used tools in modern game development. The behavior tree is an extension to the simple decision tree approach and is analogous to a large "if-then-else" code statement. This makes the BT appear to be a relatively straightforward and simply technique and it's this perceived simplicity, as well as their *initial* ease of use, which has resulted in their widespread adoption with implementation available in most of the major game engines [Epic 18, Johansen 18]. As such, there is a wealth of information regarding introduction, the implementation and optimization of the technique, but little in the way of best practices and applicability of use.

The lack of such information combined with "silver bullet" thinking, has resulted in widespread misuse across all experience levels. This misuse has resulted in monolithic trees whose size and complexity has made it all but impossible to extend or refactor without the risk of functional regression. Combine the perceived simplicity with the lack of information regarding the inherent problems as well as the lack of information on how to best leverage the technique, and the result is a ticking time bomb for both new and experienced developers alike. This chapter aims to discuss the weaknesses of this technique as well as discuss common patterns of misuse. Finally, we will discuss the suitability behavior trees for agent actuation.

2 Decision Making versus Decision Execution

Before we dive into a deep discussion about BTs, we need to briefly touch upon some AI fundamentals. In a standard AI agent model (Figure 1), we have three distinct layers/stages: sensing, decision-making and actuation (acting) [Russel 94].

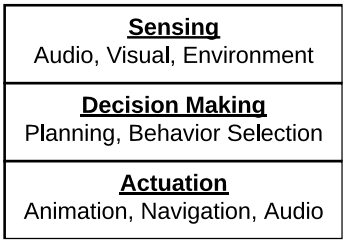


Figure 1 The AI agent model

The sensory layer is responsible for reflecting/reinterpreting the current state of the game world into a format that the AI can understand. The decision-making layer is then responsible for deciding what actions, if any, the agent needs to take given the current state of the game world. Finally, the actuation layer is responsible for the execution of the decided upon course of action.

Let's use the example of needing a cup of coffee to illustrate the various stages. Sensing might detect that we are feeling a little tired, and that maybe we are also a little cold. The decision-making layer would then process that information and come to the conclusion that we should acquire a caffeinated beverage of the warm variety and that coffee would be perfect. Actuation would then execute a series of actions to first make, then drink that cup of coffee. Information

flow between the layers should be unidirectional and no layer should interfere with any layer above it.

Now, BTs are traditionally thought of as being part of the decision-making layer, and unfortunately in most implementations that we've seen (including the author's own past implementations [Anguelov 14]), the nodes in the BTs are often responsible for both decision-making as well as the actuation. For example, in an FPS game, a BT might have nodes to determine whether we should be in cover, followed by selecting a cover and then executing move-to and shoot commands as leaf tasks. Mixing responsibilities across the two distinct layers in a single system violates the software engineering principle of 'separation of concerns' [Greer 08], and in general has a certain architectural smell to it. We will discuss, in detail, why this is a problem in sections 3 and 4 but for now it is important that we first make a simple point on the topic of decision making: *decision making is cyclical in nature*.

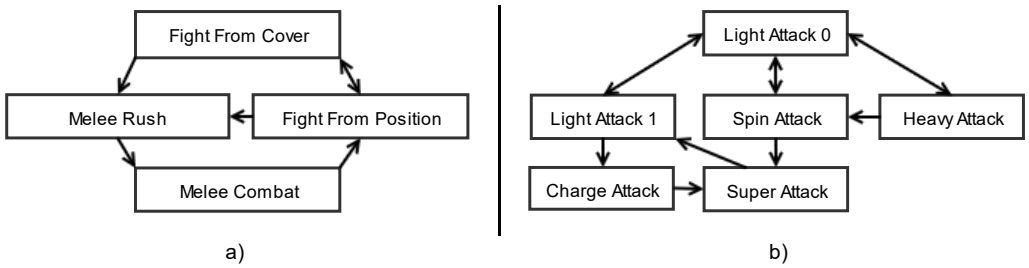


Figure 2 Example AI game loops. a) FPS loop. b) Boss Fight Loop.

What we mean by this is simply that an agent will constantly switch between various decisions over its lifetime. These decisions will often repeat according to some set of deterministic rules. If we think about the example of the FPS game we used above, we could imagine that at the highest level the combat loop would look something like that shown in Figure 2a. The agent will continuously switch between a set of combat behaviors. Figure 2b shows a game loop for a boss-fight in a brawler game, here the AI has a fixed pattern of action/behaviors it needs to perform and cycles between them for the duration of the fight. The point, stated again for emphasis, is that no matter the game or the problem: ***decision making is cyclical in nature***.

3 Behaviors Trees: How they work

As mentioned, BTs are an extension of the decision tree, in that a static tree is evaluated from its root to decide on a course based on the current environmental state. Decision trees are always evaluated from the root each time a decision needs to be taken and the tree itself is traditionally not stateful. The re-evaluation of the tree allows us to switch agent behavior irrespective of what state we are currently in as long as the relevant branch conditions are met. BTs take this concept and extend it by adding explicit tree state and long running tasks. This mean that when we re-evaluate the tree on a subsequent update we would potentially end up in the same leaf node which contains its state from the previous evaluation. This allows us to now have sequential flow constructs in the tree (i.e. sequence/parallel nodes) which were not possible in a state-less decision tree.

As BTs became popularized and the content in the trees exploded, re-evaluating the tree from its root on each AI update became computationally expensive and so resulted in the development of event-driven BTs [Epic 18, Champandard 12]. While this brought significant improvements to performance, it also created problems in terms of reactivity to environmental events as we no longer evaluate the previous branches that failed and behavioral switches will not occur until the currently selected branch completes or fails. This greatly complexified the BT concept as well as general tree authoring, requiring the development of additional techniques and tools e.g. monitor/service nodes [Epic 18, Anguelov 14] to bring back lost functionality.

An in-depth discussion into the various BTs paradigms and their development history is beyond the scope of this chapter. Interested readers are referred to the following resources for more information [Millington 08, Champandard 08, Champandard 12]. When discussing BTs in the rest of this chapter, we will be referring to the event-driven BT model as it is by far the most popular and the de facto industry standard for a BT implementation.

3.1 Tree Flow and Visualization

One of the primary draws to BTs is the fact that they are easily visualized and so tools to visualize/author/debug BTs are quite common. Due to their decision-tree origins most tools will visualize BTs as a simple top down, left-to-right tree based on a depth first traversal order. This visualization clearly conveys two sets of information to the user: the depth of the tree and the order of operations. This visualization overloads the left-to-right ordering of nodes to convey both priority of the decision and sequence of actions. In the context of a BT's functionality, the tree depth is an irrelevant bit of information, and yet we often dedicate an entire axis to it. The most important visual data points for a given tree are: the priority of decisions and the sequence of actions. As such, we recommend a visualization in which top-to-bottom ordering represents the priority of decisions and the left-to-right ordering represents the sequence of actions. We also highly recommend that standard flow control nodes such as sequence and parallel node be omitted from the visualization as they only add clutter and are of little to no value to the user. The visual distinction between various control flow nodes can achieved using different colors, containers and line styles to visualize the node connections. To illustrate, Figure 4 shows the representation of the same BT in a both formats below. It is important to note that there are a lot of quality of life visualization that cannot be shown in the figure e.g. highlight all nodes for a given sequence, etc. The rest of this chapter will make use of this visualization scheme for all BT examples.

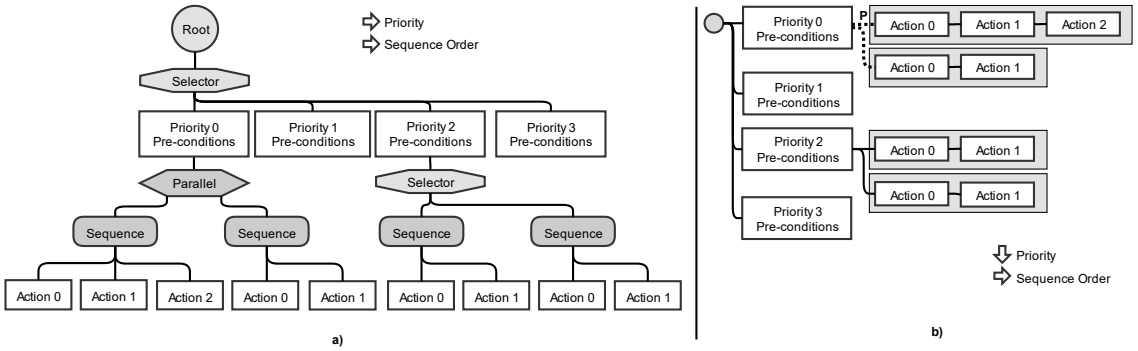


Figure 3 Behavior Tree Visualization. a) Traditional BT visualization b) Simplified BT visualization

3.2 Reactivity

To discuss the topic of reactivity (i.e. high-level behavior transitions) we will use the example of a behavior tree for an FPS game (shown in Figure 4a). Let's assume that on our first evaluation on the tree, the agent will enter the 'Idle' state. Since the tree evaluation will resume from the idle branch on the next evaluation, we can only trigger a full re-evaluation in one of two ways: the current active behavior complete (or fails), or we explicitly reset the tree.

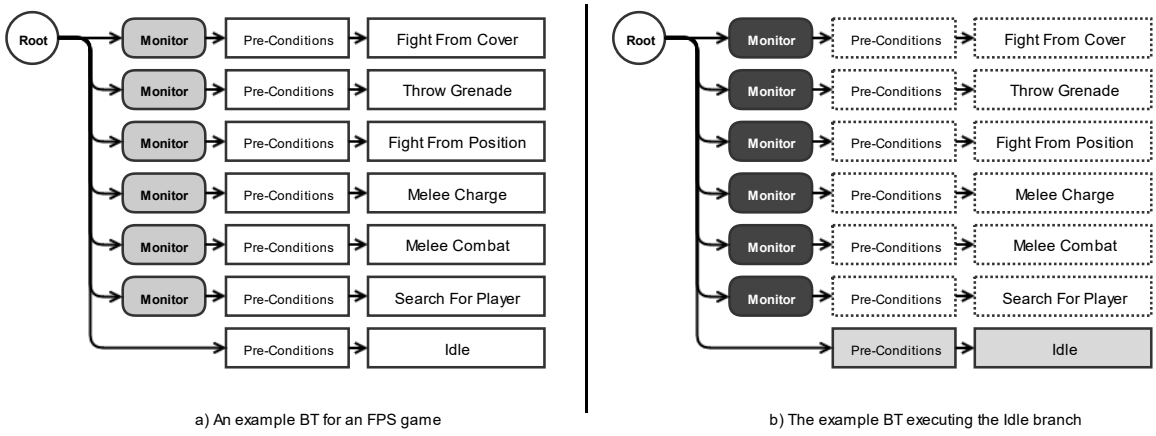


Figure 4 An example of an FPS behavior tree.

As such, we need to have some mechanism to notify the tree when an environmental change has occurred that requires a tree reset (behavior transition). This is most commonly achieved by registering monitor/service nodes that will continuously poll the environment outside of the tree execution and trigger a tree reset (i.e. behavior change) once their reset conditions are met. Figure 4b shows the registered monitor nodes (in dark gray) when we are in the idle behavior. It is also important to note that the deeper in the tree we are the more monitors we have registered and therefore the greater the evaluation cost of the tree for each update. When one of these registered monitors trigger, we reset the entire tree state and re-evaluate the tree from the root node, this will select a new behavior appropriate for the current state of the environment.

3.3 Strengths and Weaknesses

This mechanism of polling for events to reset the tree leads to the first major flaw when using BTs: ***static behavior prioritization***. Since the priority of the behaviors is fixed when authored, the conditions for picking a specific behavior are evaluated in a static order i.e. "*should we fight*" will always be checked before "*should we be idle*". Now imagine that we are currently executing a higher priority behavior and due to an environmental event, we wish to switch to a lower priority behavior. Since we never made it past the higher priority behavior in the initial tree evaluation, we can't register any monitors from the lower priority behavior to trigger a tree reset. Even if we did register all the event monitors, how would we prevent the tree re-entering the higher priority behavior again?

Let's look at the example in Figure 4a to better illustrate the problem. The "*Throw Grenade*" behavior is a lower priority behavior than "Fight from Cover" behavior but a higher priority than "*Melee Charge*" and "*Melee*" behaviors. This initial prioritization of behaviors implies certain things: the agent will prefer cover combat over everything else, melee is a last resort, etc… Now assume that we are working on the combat loop of the AI and want to experiment with certain chains of actions. Let's say we want to have the following rule: agents should always charge the player if they've just thrown a grenade at the player and the player is within some specified distance from the agent. Additionally, this specified distance is longer than the regular melee charge distance. In the current state of the BT, we have two main problems with this rule: the first being that the melee charge may not be selected after the grenade throw since 'fight from cover' and 'fight from position' might be valid options and are of a higher priority. This is the more complex problem of the two and so we will come back to it after first discussing the second issue.

The second problem, assuming we have triggered the 'melee charge', is ensuring that the agent will actually engage in melee combat given its low priority. To achieve this, we would commonly add exclusionary pre-conditions that block the higher priority behaviors e.g. only execute if you are further than some distance to target. This means that for every behavior we add to the tree above melee, we need to add that check to its pre-conditions. This causes new behaviors' pre-conditions to bleed across to other behaviors in the same tree making it difficult to both add and remove behaviors since each behavior now potentially affects a large portion of the tree. We could prevent this bleeding of conditions across multiple behaviors by adding a single ranged branch with that condition and moving all ranged behaviors within that branch. Unfortunately, this will then prevent us from using those behaviors in special cases that violate that rule i.e. agents should 'fight from position' even though they are close to the target if the target is already engaged in melee with another agent. This pre-condition/prioritization juggling makes the experimentation of new behaviors extremely cumbersome and error prone. The bleeding of behaviors to other behaviors becomes even worse once transition behaviors come into play, that is small behaviors that are meant to transition the character between various state i.e. when going from investigate to combat we may want to visually signal the behavior transition through animation or audio in a different manner than if we were transitioning from idle to combat. The problems with transitions in the context of BT is discussed in more detail in [Anguelov 17] and has been left out of this chapter for the brevity's sake.

Now coming back to the issue of correctly triggering ‘melee charge’ after a grenade throw. To actually achieve the desired behavior implies that once we complete the throw grenade behavior and reset the tree, we need to evaluate the preconditions for ‘Melee Charge’ before anything else. We cannot use preconditions to block the higher priority states since if the ‘Melee Charge’ conditions fail, we will end up with ‘Melee Combat’ as our only other option which is probably invalid as well. Not only that but when we evaluate the conditions for ‘Melee Charge’ we need to do so with a different set of rules just because we executed ‘Throw Grenade’ first. This seems like an awful amount of complexity just to try out a relatively simple gameplay rule.

One common approach to this problem is to have a global monitor at the root of the tree that will check various conditions, which, if valid, will set some evaluation flags that can be used to explicitly guide the tree evaluation. In our example we could have a global monitor that checks a) whether we have just thrown a grenade and b) that the condition for the melee charge are met and in the case that both conditions are true, we would set a “*ForceMeleeCharge*” flag and a “*DisableAllBehaviors*” flag. Each branch will then fail based on the “*DisableAllBehaviors*” flag, except for the “Melee Charge” behavior which has been told to explicitly succeed. This approach is extended and formalized in [Ocio 12]. While this works, what we have effectively done is added a secondary decision tree to override the original tree. We have also created an implicit transition between nodes in unrelated branches and so introduced the concept of cycles in the original acyclic BT. Furthermore, we have completely undermined the visualization aspect of the tree as now we can no longer reason about the execution flow of the tree purely based on the visual representation. The reality is that with this approach, we have implicitly converted the BTs into an expensive, unreadable and error-prone finite state machine. We are also now trying to represent what is for all intents and purposes a cyclic graph, as an acyclic directed tree (at this point alarm bells should hopefully be ringing). Lets us try to drive this point home, figure 5 shows the same agent loop represented as both a state machine as well as, as a BT – while both representation are identical in function one is significantly more understandable that the other. A new hire or a programmer/designer unfamiliar with the system will have a significantly easier time understanding the one over the other, and therefore ramp up faster and be less likely to make a mistake when extending the design.

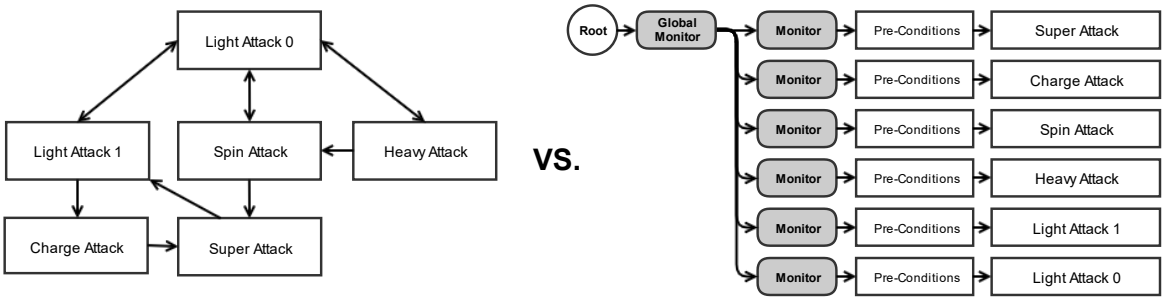


Figure 5 The same AI game loop represented as both a State Machine (left) and as a BT(right).

At this point, we've only discussed the problems and weaknesses associated with BTs. In terms of strength, BTs are one of the best formats in which to describe a discrete sequence of actions. They offer the ability to chain actions in a simple sequence. They offer all the basic flow control functionality needed to execute a complex sequence of actions, as well as offering the ability to execute multiple sequence of actions in parallel, allowing users to layer actions on top of other actions e.g. layering aim/shooting/reloading on top of navigation. Finally, we've already touched on the other strengths of BTs: the ease of visualization and conceptual simplicity.

4 The right tool for the wrong job

While we have spent a significant portion of this chapter discussing the problems with BTs, the case can also be made that those problems are not really problems with the BT techniques themselves but rather problems that result from our misuse thereof. BTs can be defined as a mathematical model for plan execution [Wikipedia 18] and the implication of this definition is that a BT is intended as an execution system for an already decided upon plan. This squarely puts the BTs into the actuation layer of the AI agent model whereas in games, we tend to typically use it as part of the decision-making layer. We also tend to use it in a way that blends in actuation into the decision-making process further complicating the overall problem. Recall the earlier points we made regarding the nature of decision-making, there exists a fundamental logical disconnect in using an **acyclic** directed graph (BT) to model a **cyclic** problem.

By forcing the use of BT to model cyclic behavior, we end up artificially blowing up the complexity of the problems we are trying to solve. We end up violating the encapsulation of behaviors and logic, and implicitly create unnecessary dependencies between unrelated behaviors in the overall agent model. At this point, the cost of introducing new behaviors in an already complex BT can become exponential. As such, the use of a BT as the decision-making layer can severely affect prototyping and iteration times.

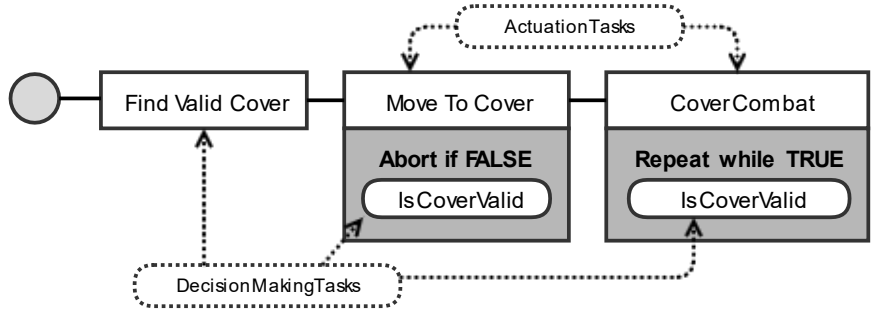
If we step away from the idea of using a BT as a decision-making tool and think of it purely as an actuator (i.e. an action execution script), we open the door to techniques better suited for actual decision making as well as allowing us to build and validate actuation behaviors independently of decision making. This also implies that the granularity of actions has to be increased. In using BTs as a decision-making tool, the granularity of actions has often been set quite low often as means of trying to reduce complexity and tree size. For example, instead of having BT tasks for actions such as move-to, look-at, reload, etc…, the tasks instead would be entire behaviors such as "Fight from position". Unfortunately, at this level of granularity, the tree would essentially devolve into a complex decision tree further devaluing the utility of using a BT in the first place.

If we wish to move toward using the BT as an actuator rather than a decision maker, we need to understand what decisions need to be moved out of the BT and what decisions can remain. Let's use the example of the 'Fight from Cover' branch from Figure 4, if we expanded that branch, we can imagine that there were three individual actions that needed to be taken (again with the assumption that a low granularity of actions was used). An example version of the

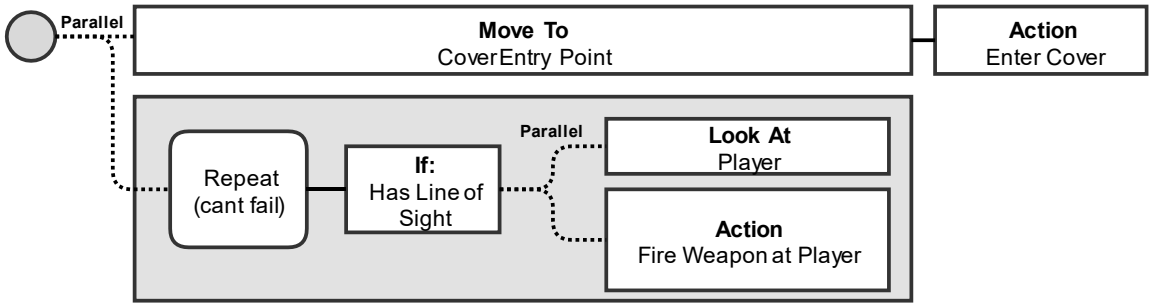
expanded ‘Fight from Cover’ branch is shown in Figure 6a.

In looking at the BT in Figure 6a, the ‘*FindValidCover*’ and ‘*IsCoverValid*’ actions are clearly needed for the decision-making layer to be able to terminate that branch when the agent doesn’t have a valid cover. On the other hand, the ‘*MoveToCover*’ and ‘*CoverCombat*’ actions are only concerned with actuation of the agent. As a first step, lets remove the decision-making level task and only focus on the actuation tasks. Both the remaining actuation tasks can be thought of as simple behaviors: with the ‘*MoveToCover*’ being responsible for moving to and entering a cover while ‘*CoverCombat*’ is responsible for the combat loop when already in a cover. Both tasks are perfect candidates to be modeled using a BT.

Let’s focus on the ‘*MoveToCover*’ actuation task and try to represent it as a BT with a higher level of granularity of actions. The resulting BT is shown in Figure 6b. We will immediately notice that there still exists some degree of decision making in it: the choice of whether to shoot at the player, a decision which given the context is perfectly valid. The distinction here being that the decision made only affect the actuation of the agent and cannot affect the overall agent state i.e. which behavior we currently executing, is that behavior still valid, etc…



a) **Fight From Cover** behavior at a lower granularity of actions



b) **Move To Cover** behavior at a higher granularity of actions

Figure 6 The expanded ‘FightFromCover’ branch from Figure 4 and a BT implementation of the ‘MoveToCover’ action.

We are not advocating removing all decision making from the BT but rather limiting it to only decisions necessary for the executing of the given task. Decision making should only be concerned with selecting a goal (i.e. deciding upon a course of action). The reality of how we’ve seen BTs used so far is as a decision-tree/execution tree hybrid which violates the

separation of concerns principle. As such we are strongly recommending that AI programmers and designers stop thinking of BTs as a tool used for decision making and steer away from using them as such.

8 Leveraging Behaviors Trees for actuation and archetype variety

In most of the game AI architecture implementations we've seen, actuation is the most likely layer to end up in code as well as interspersed with decision making. This make life difficult for us both when authoring as we are trying to solve two problems at once and our conditions and edge cases being muddied i.e. was this edge case due to a problem with decision making or was there something wrong with actuation. There is a lot to be gained by moving the actuation problem out of code and into data and this is where the BTs come in to play. We can easily build a set of generic action BT nodes that can then be combined to author complex actuation behaviors. As each of these actuation behaviors will have a clear singular goal and a clear set of inputs, they will naturally remain quite small. These smaller behaviors can be easily tested using a basic test harness and we can easily validate whether an agent's actuation layer is functioning as expected without needing to bring the decision making or sensory layer into the mix.

Additionally, as the number of these smaller actuation behaviors grow, we can easily make variations of them for the various archetypes within our games. We can even use them with the same high-level decision making to increase challenge and add variety. For example, the in a shooter, lower level agents might only have a '*MoveToCover*' behavior that simply has a move-to node with the highest possible speed set in it. Higher level agents might have a version of that behavior where they also run a blind fire action in parallel with the move-to while our most advanced AI might enable strafing on the move-to and have an aimed fire branch layered on top of the move (e.g. Figure 6b). If these three variants are all used with the same decision-making structure, we will have a noticeable difference in the behavior of the final AI at a very low production cost. It is also important to note that with the same set of BT nodes that we used to build the various '*MoveToCover*' behaviors, we can build a whole set of other combat behaviors e.g. '*FightFromPosition*', '*CoveringFire*', '*AdvanceOnPosition*', etc…

There is one last benefit to mention, we spent a lot of time explaining how great it is removing decision-making from actuation, well the reverse is also true. In removing the actuation problem, decision-making becomes a significantly easier problem to solve. It also allows us to more easily experiment with various approaches to decision-making. If we had a huge tree that contained the entire decision-making and actuation logic of the AI, we'd be hard pressed to try to convince management to let us potentially destroy everything just to experiment with a new approach. Now, in the case where our actuation behaviors are built and tested independently, we can easily replace the decision-making or even build it in parallel. Also, in testing out a new decision-making system with the same set of actuation behaviors, the author can more easily judge the quality of the decision making by removing any potential variance due to differences in actuation implementation.

9 Conclusion

In conclusion, we've discussed how behavior trees are commonly used in the industry as well as the problems associated with this usage. We've discussed the inherent problems of static prioritization and reactivity/transitioning that BTs exhibit and why we should stop using them as decision makers within our AI architectures. We further discussed the problem of agent actuation and suggested that behavior trees could potentially be a perfect fit for modelling an agent's actuation behavior. We hope that this a chapter will serve as a turning point for the use of and the revival of the humble behavior tree which has over the last few years unfairly garnered a bad reputation due our misuse thereof.

9 References

[Anguelov 17] Anguelov, B., Vehkala, M., and Weber, B. AI Arborist: Proper Cultivation and Care for Your Behavior Trees – GDC 2017.
https://www.gdcvault.com/play/1024218/AI-Arborist-Proper-Cultivation-and
[Anguelov 14] Anguelov, B. 2014. Synchronized Behavior Trees.
https://takinginitiative.wordpress.com/2014/02/17/synchronized-behavior-trees/
[Champandard 08] Champandard, A.J. 2008. Behavior Trees for Next-Gen Game AI.
http://aigamedev.com/insider/presentation/behavior-trees/
[Champandard 12] Champandard, A.J. 2012. Understanding the Second-Generation of Behavior Trees. http://aigamedev.com/premium/tutorial/second-generation-behavior-trees/
[Epic 18] Epic Games. 2018. Behavior Trees in Unreal Engine 4.
https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/index.html
[Greer 08] Greer, D. 2008. The Art of Separation of Concerns
http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns
[Johansen 18] Johansen, E. 2018. Angry Ant's Behave. Unity Behavior Tree Implementation. http://angryant.com/behave/
[Millington 09] Millington, I. and Funge, J. 2009. "Artificial Intelligence for Games." CRC Press.
[Ocio 12] Ocio, S. 2012, Adapting AI Behaviors to Players in Driver San Francisco: Hinted-Execution Behavior Trees. Eighth Artificial Intelligence and Interactive Digital Entertainment Conference.
[Russel 94] Russel, S. and Norvig, P. 1994. Artificial Intelligence: A Modern Approach (3rd Edition). Pearson Education.
[Wikipedia 18] Wikipedia 2018. Behavior tree (artificial intelligence, robotics and control).
https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)