Participant:5 - Interview Transcript

1. Behavior Trees (BTs)

Conditional Structures and Failure Semantics

> Concerning the BT conditional structure, one of your comments was that this kind of construct is influenced by failure semantics, and therefore it does not correspond to a pure if-then-else.
We realized that this structure does not represent a pure conditional in the same sense as an if-then-else. If the guard condition is true, action A1 is attempted, but if A1 then fails, A2 may still be executed. So the construct is "contaminated," so to speak, by BT failure handling. We had also noted this as a weakness of BTs.

Yes, because this is really an OR, not a pure if-then-else. The issue is that people often use it as if it were an if-else, but then they need to be careful. If you want to force that semantics, you must ensure that A2 is only triggered if A1 is not applicable, not merely because A1 failed.

At the BT level, the subtree should be treated as a black box. If the subtree returns success, then the parent should not care whether A1 or A2 was executed. So in practice this means "do A1 if condition C1 holds; otherwise do A2," but not in the strict mutually exclusive sense of a classical if-then-else.

To encode a pure if-then-else, you would need to make A2 guarded explicitly, for instance with a sequence node containing not C1 followed by A2. That would prevent the case where A2 is taken only because A1 failed after C1 was true.

> So the usual description "if C is true, do A1, else do A2" is not really correct.

Exactly. It happens that way in some cases, but not only for that reason. It is sufficient, not necessary.

Practical Use of Conditional BT Structures

> Would it still be correct to present this kind of BT pattern as a conditional structure, provided that we explicitly clarify its semantics?

Yes, but you should clearly state that it is not a pure mutually exclusive if-then-else. A1 and A2 are not strictly mutually exclusive in the same way as in classical control-flow languages.

If you want the pure if-then-else, then the proper way is to encode the complementary condition explicitly, for example with C for the first branch and not C for the second branch.

> *So we could keep the current structure in the mapping, but add a disclaimer explaining that it realizes a conditional-like structure, although not a pure if-then-else.*

Yes, that would be correct.

Backchaining vs. Explicit Conditionals

> *In practice, when modeling with BTs, do you avoid these conditional structures and instead prefer approaches such as backchaining?*

Yes, generally, I use backchaining more. I only use structures like this when, from the point of view of the rest of the system, executing A1 or A2 is effectively equivalent.

For example, I had a robot that needed to move in front of a conveyor belt to inspect bottles. To improve the camera viewpoint, it could either tilt its head or tilt its torso. We preferred to tilt the head, but in some situations the head could not be tilted because that would make the robot lose sight of another part of the system. So the behavior was: if condition C1 holds, tilt the head; otherwise tilt the torso.

This is a good use case because, for the rest of the system, both alternatives achieve the same higher-level objective.

Memoryful vs. Stateless / Reactive Execution

> *Do control structures change substantially depending on whether the BT is stateless/reactive or memoryful?*
> *We were also discussing execution semantics such as memoryful versus stateless/reactive BTs. Does the same control structure have substantially different implications in those two cases?*

Yes. In my book and earlier papers I essentially assumed vanilla reactive nodes, meaning nodes without decorations or memory. That made sense because BTs were intended to support reactive systems.

Later, when tools became popular, many people started using BTs mainly because they were easy to design, not because they wanted reactivity. Libraries such as Groot and BehaviorTree.CPP made memoryful sequences and fallbacks popular, and that created some confusion.

If you replace reactive nodes with non-reactive ones, the execution is obviously not the same, though the BT may still function. The difference is that with non-reactive nodes, some checks are performed only once, so you lose part of the reactivity, which is one of the major advantages of BTs.

For example, some actions in robotics, such as navigation in Nav2, immediately return success if the goal condition is already satisfied. That fits well with a reactive BT interpretation: the action node encapsulates the check internally, and the BT does not need to know whether the behavior was implemented as a primitive action, a BT, or a hierarchical task.

However, if checking a condition requires a heavy process, such as several seconds of object recognition, then a fully reactive BT may block there, which changes the practical trade-offs.

Converting Memoryful Trees into Reactive Ones

> If one has a memoryful BT and wants to turn it into a reactive/stateless one, would it make sense to replace each action with an action that explicitly checks whether the goal is already satisfied?

It depends on the action. If an action has already been effectively completed, then the action itself should return success immediately instead of returning running again. Otherwise, in a reactive BT, you can get undesirable oscillations: one action starts, then another one interrupts, then the first starts again, and so on.

A practical example is again navigation in Nav2: if the robot is already close enough to the destination, the navigation action does not even start planning; it simply returns success. That implies there is some control condition in the system that checks whether the action is still necessary.

So yes, but it requires that the action or surrounding control node explicitly manages the success condition.

Loops in BTs and Decorators

> Many interviewees mentioned loops. Since decorators are not standardized and are often custom, can BTs realize different kinds of loops through decorators?
The loop issue came up several times. We noticed that decorators are somewhat problematic because they are not standardized, and in fact you yourself write that decorators are often entirely custom. So if I model a loop using a decorator, this seems rather open-ended.

Yes, but in reality there are no real standard decorators. Decorators are inherently custom because they modify the execution of a subtree and they have a single child. That is exactly their peculiarity.

For example, you can define decorators like: repeat forever, repeat until success, repeat until failure, and so on.

So with decorators you can implement do-while, while, for, and many other variants.

The important thing is simply to define: when the decorator returns success, failure, or running, and when it re-ticks its child.

That is enough.

> So with a decorator we can realize many different types of loops, and it is more appropriate than forcing a loop through conditional structures.

Exactly. The reason why people sometimes tried to realize loops using only actions, conditions, sequences, and fallbacks was mostly to demonstrate the Turing completeness of BTs. If you have memory and use it together with conditions, you can realize any loop you want. But from a modeling perspective, using a decorator is the more natural solution.

Strengths and Weaknesses of BTs

Tracing the Origin of Failure

> One weakness we had written was: "It is difficult to trace the origin of a failure returned by a deeper node without explicit logging." Would you agree?

Not exactly. If you have a BT with failures, you can traverse the tree and eventually identify which action returned failure. So the origin can be found.

The real point is different: BTs manage execution flow, not data flow. If you want to know why an action failed, then that information has to be handled explicitly, for example through ports, a blackboard, or an error message logged by the action itself.

That is how I usually do it: every action can produce an error message, and I log it. Sometimes I even use that error information to decide among different fallback branches.

For instance, suppose a navigation action fails. If all failures lead to the same recovery action, then I do not care why it failed. But if the recovery depends on the reason — e.g., obstacle in front, no destination available, low battery — then I need to encode and inspect that reason explicitly.

> So the issue is not really that the origin cannot be traced, but that the reason for the failure is not automatically represented and must be explicitly modeled if relevant.

Exactly. And that is not really unique to BTs. The same issue appears in state machines too, unless you explicitly distinguish different failure events.

Use of Nodes with Memory

> We had also listed "use of nodes with memory" as a weakness. Would you agree?

It is true that I am generally against nodes with memory, even though I use them more often than I would like to admit. But I would not call them a weakness of BTs as such.

They are more like a feature that has trade-offs. If you use BTs with memory, then yes, some conditions may not be re-evaluated, and you lose part of the reactivity. But this is not a weakness of the formalism itself; it is a consequence of choosing that feature.

> Our concern came from task saving and resuming. If a subtree has memory and is tracking the mission progress, it helps resume execution later. But at the same time it may prevent re-checking conditions. For example, if I place a ball down and then it rolls away, a memoryful subtree may not notice that anymore.

Yes, exactly. That is the right framing. I would move this discussion to the section on state saving and task resuming, rather than presenting memory as a standalone weakness.

Another Weakness: "State Machine Mindset"

> Are there other weaknesses of BTs that should be mentioned?

One weakness is actually not in the formalism but in how people use it: many people approach BTs with what I call a "state machine mindset."

They try to use BTs as if they were state machines, but BTs require a different way of thinking. You need to reason more in terms of: backchaining, reactivity, conditions, ticking semantics, rather than simply sequencing actions one after another.

That is probably the main practical weakness I see: people often do not change paradigm when they move from state machines to BTs.

Another possible limitation is that some strange "goto-like" loop patterns are harder to represent in BTs. For example, "do A, then B, then C, but if C fails in a certain way, go back to B." Those are less natural in BTs than in state machines. I have not found many real use cases for such patterns, but they are more awkward to model in BTs.

3. State Machines (SMs)

Guards on Outgoing Transitions

> Regarding state machines: for a choice pseudostate, should outgoing guards be considered mutually exclusive?
We had already discussed that, but I wanted to confirm: in a state machine conditional branching structure, are outgoing guards supposed to be mutually exclusive?

I have never found a formulation saying that they must be mutually exclusive in the formalism. However, they have to be modeled as mutually exclusive in practice.

That is the problem: if you have two outgoing transitions with guards A and B, and both are true, then where should the system go? The model itself does not define it.

When implementing it, of course, one guard may be checked before the other, but then the actual behavior depends on the implementation, not on the model. So the implementation is no longer fully adherent to the model, because the model never specified the resolution policy.

Controller State vs. World State

> In the questionnaire you wrote that the comparison was missing a distinction between controller state and world state. Can you explain this point?

Yes. In a state machine, you are generally modeling the controller state of the robot. But the conditions on transitions depend on the world state, or at least on the belief state of the world.

So the point I wanted to make is that if you say “the state machine models the system,” that can be ambiguous. It may sound as if the state machine models the actual evolution of the whole robot-plus-environment system. But in many cases what it actually models is just the control state followed by the controller.

The evolution of the world, or the robot plus environment as a whole, is not directly modeled as machine states. Instead, that information appears in the transition conditions.

> So in our case, where we are mapping robotic mission models, we should explicitly state that the state machine models the controller behavior, not the full evolution of the physical system.

Exactly. Otherwise someone might think that the state machine is meant to model the whole system dynamics, which is not the case here.