9

Overcoming Pitfalls in Behavior Tree Design

Anthony Francis

9.1 Introduction

Unless you have been living under a rock, or are new to the game industry, you have probably heard of behavior trees (Isla 2005, Champandard and Dunstan 2012). Behavior trees are an architecture for controlling NPCs based on a hierarchical graph of tasks, where each task is either an atomic, a simple behavior an agent can directly perform, or a composite, a behavior performed by a lower level behavior tree of arbitrary complexity. As they provide a cleaner decomposition of behavior than alternatives such as hierarchical finite-state machines—and because of the many good reports from people who have successfully used them—behavior trees are increasingly used to allow large software teams to collaborate on complex agent behaviors, not just in games but in the robotics industry. But there are a few key choices in their implementation, which can either make them much harder to architect than they need to be—or much more flexible, easy to extend, and easy to reuse.

I found this out the hard way. By a weird trick of fate, I have implemented behavior trees five times, including three systems in Lisp similar to the architecture that later got the name behavior trees as part of the *Halo 2* AI, and more recently two behavior tree implementations in C++ for robots at Google. Along the way, I have learned some of the things to NOT do when creating a behavior tree—including needlessly multiplying core primitives, inventing a whole programming language for control, and jumping the gun on routing all communication through "proper" channels like a blackboard—and I have developed

specific recommendations on how to create a system which avoids these pitfalls, especially if licensing or interoperability concerns prevent you from building on existing commercial or open-source solutions.

9.2 What Makes Behavior Trees Work

Ideas like behavior trees were around long before *Halo*. Hierarchical decompositions of behavior into tasks were pioneered in particular by reactive action packages, or RAPs (Firby 1987): high-level RAPs get decomposed into low-level RAPs, ultimately bottoming out in leaf skills used to directly control real and simulated robots (Bonasso et al. 1997). My TaskStorm architecture is even closer to modern behavior trees, integrating hierarchical task decomposition directly into a blackboard system encapsulating the agent state, enabling cognitive operations to be time sliced with each other over the course of behavior (Francis 2000).

Although these systems are *functionally* similar to behavior trees, the key insight that Damian Isla added is a software design focus on simplifying the code needed to generate behaviors (Isla 2005). Isla argues that four key features made it possible to create the *Halo 2* AI and its successors at large scales: customizability, explicitness, hackability, and variability. Behavior trees are customizable because behaviors are decomposed into smaller components, which can be individually changed or parameterized; they are explicit because the things that an agent does can be represented as distinct behavior nodes, and the whole behavior as a graph; they are hackable because the radical decomposition of behaviors makes it easy to replace nodes with custom versions, and they allow variability because the control algorithms themselves are represented as nodes.

In my experience, it is easy to get lost building functionality that you think you will need and in the process lose sight of this simplification focus. A system which functionally breaks behaviors into atomic behaviors orchestrated by trees of composite behaviors may act like a behavior tree, but it will not be customizable or hackable if behaviors are too tightly coupled, nor will it be explicit or variable if the nodes are too large. What is worse, architectural constraints to enforce these features can become a cure worse than the disease, making the overall system harder to maintain. Although some of these problems are unavoidable, other pitfalls are easy enough to avoid if we are careful about how we design our behavior trees.

9.3 Pitfalls in Behavior Tree Design

A behavior tree runs as a decision-making component within a larger structure—a game engine, a robot operating system, or a cognitive architecture—and it is tempting to design something that serves all the needs of the parent system. But software components work better when they have clearly defined responsibilities and clear separation of concerns, and making your behavior tree too much can, perversely, leave you with something that does too little. Three of these pitfalls are

1. Adding too many kinds of classes to the decision architecture of your behavior tree.
2. Building a complete programming language into your behavior tree before you need it.
3. Forcing all communication to route through the blackboard as a point of principle, rather than when required by the needs of the application at hand.

If you are on a large project and have a clear specification for (or experience with) what you need, (2) and (3) may not apply to you; but if you are on a smaller or novel project, simpler structures may work for you.

9.3.1 Pitfall #1: Creating Too Many Organizing Classes

Occam's Razor gets reduced to "Keep It Simple, Stupid," but originally stated, translated more like "Do not multiply entities needlessly"—which is more on point for understanding a pitfall of behavior tree architecture: creating a separate architectural category for every kind of thing that your system needs to do. This, perversely makes it harder to develop the features you need.

For example, in addition to atomic behaviors that perform actions and composite behaviors which orchestrate them, your characters may need low-level "skills" that directly manipulate your animation system or your robot controllers, or high-level "modules" that manage the resources provided by your game engine or your robot operating system. But it is probably a mistake to have a separate architectural category for each of these—that is root classes for Behaviors, Composites, Skills, and Modules. Behavior trees already provide low-level and high-level constructs, and with the addition of an external memory store, their decision-making power is Turing-complete. So why not just use behaviors?

Many of the prehistoric behavior tree-like systems, described earlier, made distinctions between concepts like skills, tasks, task networks, or modules—and created classes or modules to implement each. This approach got the job done, and maybe was a fine way to do things in the bad old days of Lisp hacking—but this multiplication of entities introduces pitfalls in modern C++ development, which often depends on having smaller numbers of classes to enable quicker refactoring.

For example, one behavior tree we developed for an internal Google robotics project had both `Modules`, which talked to the robot operating system via `Contexts`, and `Tasks` that made up the behavior tree themselves, which communicated to agents indirectly through thin wrappers called `AgentHandles` that hid the underlying communication protocols. But as the architecture of the system evolved, every simple refactoring affected both trees of classes—`Modules` and `Tasks`—and more complex new functionalities affecting things like the blackboard system had to be developed for and woven into both since their features were not quite in parity.

The converse of creating too many kinds of organizing abstractions is to use just one: tasks. We have yet to be burned by thinking of everything we can do in a behavior tree as creating a new kind of task. Performing an atomic action? A task. Grouping several actions? Another kind of task. Accessing system resources? A task. Running an entire behavior tree script? Yet another kind of task. We ultimately adopted a mantra "It's Tasks, All the Way Down."

When we ported this behavior tree to a new robotic platform, everything became a task. Well, technically, not everything—resources from the robot operating system were provided uniformly by `ExecutionContext`, but all classes responsible for behaviors—Modules, things like the old `AgentHandles`, and scripts—were implemented as children of a single task, the `SchedulableTask`, shown in Listing 9.1; this class provides an interface for a single decision-making step, given a particular `ExecutionContext`.

Forcing all these different types into a single model did not hurt us; on the contrary, the result was that our new behavior tree became a system with a single responsibility—decision-making—whose API had just two points of external contact: the `ExecutionContext`

Listing 9.1. The root of our behavior tree system: the `SchedulableTask`.

```
class SchedulableTask {
 public:
  SchedulableTask(
    const string& name,
    ExecutionContext* execution_context);
  virtual ~SchedulableTask() = default;
  virtual TaskStatus Step();
  // More member functions relevant to our use case ...

 private:
  string name_;
  ExecutionContext* execution_context_;
  TaskStatus status_;
};
```

provided at the top or custom code in leaf tasks at the bottom. This simpler design enabled us to build out functionality in weeks where the old system had taken us months.

9.3.2 Pitfall #2: Implementing a Language Too Soon

A behavior tree is a decision-making system, which means that the behavior nodes that make up the tree need two separate properties: a node needs both the ability to run a behavior and to decide whether to run it. This raises the question of whether the behavior tree system itself should implement a condition language.

You can do a lot with a little; even very simple decision-making structures are Turing-complete: you can perform virtually any computation with just NANDs or NORs. Therefore, you can implement sophisticated control behaviors with very few node types. Tests can be implemented by atomic actions that simply succeed or fail; if–thens can be implemented by composites like Selectors that execute the first nonfailing action, and blocks of code can be implemented by Sequences that execute all their nonfailing actions or Parallel nodes, which allow multiple actions to be executed concurrently (Champandard and Dunstan 2012).

But implementing complicated behaviors out of very simple components, Tinkertoy-style, can lead to a lot of boilerplate—so it is tempting to add more control constructs. A Decorator task that wraps other tasks enables the creation of Not that inverts the significance of failures; a subclass of a Sequence can create an And task that succeeds only if all its tasks succeed, and so on, and so on. Soon you find yourself developing a whole programming language, which enables you to parsimoniously represent character logic.

Perhaps surprisingly for a "pitfalls" section, I am not recommending that you do not implement a whole programming language (how is that for a double negative); if your behavior tree gets a lot of usage, you probably do want to implement a language. But I have implemented these kinds of languages in two different ways: driving from the top–down from language concerns and driving from the bottom up based on an understanding of the problem domain, and the latter is a much more effective strategy.

There are an enormous number of possible language constructs—`Not`, `And`, `Or`, `If-Then-Else`, `Cond`, `Loop`, `For`, `While`, `Repeat-Until`, `Progn`, `Parallel`, `Any`, `All`, `Try-Catch-Except`—one or more of which you may have

fallen in love with when learning your favorite programming language (which may not be the C++ you are forced to program in). The logic for most of these constructs is very clear, so it is easy to build up a large library for any possible need.

The problem is, much of this logic is better implemented using just standard behavior tree nodes—`Actions`, `Decorators`, `Sequences`, `Parallels`, and `Selectors`. (I challenge you to find a meaningful difference between a Selector and a Cond). The best case scenario is that you end up with a node with a nonstandard name. A worse scenario is that you code functionality you do not need. The actual worst-case scenario is that you end up with duplicated logic between very similar nodes that you actually need but now have trouble maintaining as your API evolves.

A better approach is to begin with the standard set of behavior tree nodes—Actions, Decorators, Sequences, Parallels, and Selectors—and to push these as far as you can for your problem domain. If you are implementing it yourself, each behavior tree will have slightly different semantics for task execution, failure and success, and logical tests. Once you understand what patterns you need for your problem domain, you can then expand out your language—often by specializing an existing construct for a new need. This will result in less duplicated, easier to maintain code—and a library of constructs driven by what is useful for your game.

This is easy to see for structural features like object containment; for example, if the `SchedulableTask` has `GetChildren` and `AddChild` to manage its children, these should be overridden differently in atomic leaf tasks with no children and in composite tasks which have children. For almost all of the different kinds of containers in our use case, one high-level class, the `ContainerTask` shown in Listing 9.2, suffices to handle all these implementation details.

Listing 9.2. Expanding `SchedulableTask` to support `ContainerTask`.

```
// Expanded SchedulableTask definition ...
class SchedulableTask {
 public:
  // ... previously declared member functions
  virtual std::vector<SchedulableTask*> GetChildren() = 0;
  virtual bool AddChild(std::unique_ptr<SchedulableTask> child) = 0;
  ...

// ContainerTask definition ...
class ContainerTask : public SchedulableTask {
 public:
  ContainerTask(
    const string& name,
    ExecutionContext* execution_context);
  ~ContainerTask() override = default;

  std::vector<SchedulableTask*> GetChildren() override;
  bool AddChild(
      std::unique_ptr<SchedulableTask> child) override;

 private:
  std::vector<std::unique_ptr<SchedulableTask>> children_;
};
```

This is bread and butter class design, so I will assume you have no trouble carrying forth the implementation of this class, or of its counterpart, the `AtomicTask`. Virtually every composite task we have inherits from `ContainerTask`; the only departures we have from this pattern are certain kinds of queues that do not inherit from a vector.

But we have an even better opportunity for reuse in the area of behavior. Rather than simply implementing each task's Step member function separately, we should decompose the Step member function and expose the innards of the stepping API to subclasses through the class's protected interface, shown in Listing 9.3.

Listing 9.3. Implementing stepping.

```
// Expanded SchedulableTask definition ...
class SchedulableTask {
 public:
  // ... previously defined member functions
  virtual TaskStatus Step();
  virtual bool IsFailure() const;
  virtual bool IsTerminated() const;
  virtual TaskStatus GetStatus() const;
  // More member functions for our use case ...

protected:
  virtual TaskStatus PerformAction() = 0;
  virtual void SetStatus(TaskStatus status);
  // More member functions for our use case …
 ...

// Expanded SchedulableTask implementation ...
TaskStatus SchedulableTask::Step() {
  if (!IsTerminated()) {
      SetStatus(PerformAction());
  }
  return GetStatus();
}
```

Using these protected member functions, the `Step` member function is implemented in `SchedulableTask` in a way which looks almost trivial. But by defining this (and similar member functions) near the top of the tree, we define the execution model for tasks, so all that we really need to know about a task is how it differs from the norm. The `ContainerTask` we have already shown just holds tasks; it does not have execution semantics. But at the very next level of the tree appear tasks like `ParallelTask` or `SequenceTask`, which do override these protected methods, as shown in Listing 9.4.

The meat of an actual `SequenceTask` can be fairly simple, encoded in its `PerformAction` member function. But we did more than just implementing this member function; we exposed some of its innards in the protected API as well, including `HandleChildFailure` and `AdvanceToNextChild`.

Listing 9.4. Implementing stepping in `SequenceTask`.

```
// SequenceTask definition ...
class SequenceTask : public ContainerTask {
 public:
  SequenceTask(string name,
               ExecutionContext* execution_context);
  ~SequenceTask() override = default;

 protected:
  uint current_task_;
  TaskStatus PerformAction() override;
  virtual bool AdvanceToNextChild();
  virtual TaskStatus HandleChildFailure(
      TaskStatus child_status);
  // More member functions related to our use case ...
};

// SequenceTask implementation ...
TaskStatus SequenceTask::PerformAction() {
  if (GetStatus() == WAITING) {
      SetStatus(STEPPING);
      current_task_ = 0;
  }

  TaskStatus child_status{STEPPING};
  auto child = GetCurrentChild();
  if (child != nullptr) {
    // Step the child unless it has stopped.
    if (!child->IsTerminated()) {
      child->Step();
          }

    // Now check for and handle completed children.
    if (child->IsTerminated()) {
      if (child->IsFailure()) {
          // Propagate child failure up
          child_status = child->GetStatus();
      } else {
          // Move to next task
          AdvanceToNextChild();
      }
    }
  }

  // Now check for and handle failed children.
  if (IsStatusFailure(child_status)) {
    SetStatus(HandleChildFailure(child_status));
  } else if (current_task_ >= GetChildren().size()) {
    SetStatus(SUCCESS);
  }
  // Propagate status up the API.
  return GetStatus();
}
```

(*Continued*)

```
bool SequenceTask::AdvanceToNextChild() {
  // Advance until we run out of children.
  current_task_++;
  return HasCurrentChild();
}

TaskStatus SequenceTask::HandleChildFailure(
    TaskStatus child_status) {
  // Propagate failure up to parent.
  return child_status;
}
```

We chose to do so by looking at the actual decision-making use cases of our robot application, where we wanted finer-grained control on how the tasks responded to various circumstances without making the actual `SequenceTask` very complicated.

For example, we wanted a `TryTask` which tried tasks in sequence but did not fail itself if a child failed. By exposing `AdvanceToNextChild` and `HandleChildFailure` in the protected API, we were able to write a `TryTask` with (essentially) half a page of code, shown in Listing 9.5.

Listing 9.5. Implementing a `TryTask`—essentially, the whole thing.

```
// try_task.h
// Header includes omitted ...
class TryTask : public SequenceTask {
 public:
  TryTask(const string& name,
          ExecutionContext* execution_context);
  ~TryTask() override = default;
 protected:
  TaskStatus HandleChildFailure(TaskStatus_) override;
};

// try_task.cc
// More header includes omitted ...
TaskStatus TryTask::HandleChildFailure(TaskStatus _) {
  // Ignore failures, continue until we are done.
  return AdvanceToNextChild() ? STEPPING : SUCCESS;
}
```

The only member function we needed to override was `HandleChildFailure`, and that override itself was very simple. This made developing and testing this class easy; we could rely on the containment and stepping logic of the parent class, and focus our testing on `HandleChildFailure` and its impact on, well, child failure.

This design breaks behavior tree down based on a class hierarchy with overridable member functions. In C++, this can be less efficient than carefully written code specialized to the use case at hand. The chapter on the Behavior Tree Starter Kit in the first volume of *Game AI Pro* describes a series of these tradeoffs, which you could consider in more detail (Champandard and Dunstan 2012).

For our use case, this explicit breakdown of behavior was appropriate for a robot whose top-level decision-making cycle needed to run no more than 30 hertz. This may not necessarily work for a game that needs to optimize every cycle, but we were able to run a behavior tree step for a simple benchmark at 60 million hertz on a not too beefy computer. So you may want to begin with a clean breakdown, which is easily extensible and return to optimize it later as your benchmarks, profiling, and the needs of your game demand.

9.3.3 Pitfall #3: Routing Everything through the Blackboard

Previously, BT-like systems were designed to break cognitive tasks apart into small parts (Francis 2000), so the thinking that the system did could be interleaved with the system's memory retrieval processes; for that memory retrieval to work, virtually all the data that the system manipulated needed to be exposed in the system's blackboard. A robot control system where multiple independent processes communicate, like ROS (ROS.org 2016), has similar constraints, and the first version of Google's robotics behavior tree supported close interaction between the system blackboard and the behavior tree. Although not a required part of the system, this enabled us to specify a behavior tree and its data at the same time.

Similar concerns operate in games with characters that need sensory models, like stealth games (and many other modern shooters), but it is probably a mistake to tie the overall behavior tree logic too closely to the logic of the blackboard—and even more of a mistake during prototyping to make simple tasks dependent on a blackboard for communication. Coding all tasks to use a system blackboard or sensory manager, while providing many advantages as systems grow larger and more complicated, is much more complex than using native language features for communication—and is often less reliable, unless you are enough of a C++ template wizard to create a fully typesafe blackboard (and if so, more power to you).

In the prototyping phase, doing things "right" this way can actually interfere with exploring the design space of your behavior trees and perfecting their logic. For our first behavior tree, the blackboard and the behavior tree were designed together to complement each other. Unfortunately, this meant a change to the API of one affected the other—particularly in how the hierarchy of tasks referred to the hierarchy of the blackboard, and vice versa. On at least two separate occasions, a lower level API change led to a month or more work rewiring the tasks and the blackboard so that all the data were available to the right tasks.

The alternative is to strongly decouple the behavior tree from the blackboard. The same behavior tree logic should work with a complicated hierarchical blackboard with a rich knowledge representation—or with a simple C++ plain old data structure (POD), or even with no explicit blackboard at all, and communication performed ad hoc between cooperating tasks. Using ad-hoc communication can make it more difficult to build a full data-driven behavior tree, but for many problems, it is sufficient. For example, one machine learning system that successfully learned better-than-human behavior trees for controlling an unmanned aerial vehicle used a C++ POD containing only a handful of values as its "blackboard."

When we reimplemented our behavior tree for a new robot platform, we temporarily abandoned the blackboard in favor of ad-hoc communication, which had two benefits. First: we were able to complete and test the logic of the blackboard to a very high degree; second, when we made the decision to abandon a low-level API, only two leaf tasks needed

to be changed. We still have a use case for the old blackboard in the new behavior tree—but the clean separation of concerns from the old blackboard means we have a behavior tree which is reliable, well tested—and can work with a variety of communication mechanisms.

I cannot as cleanly show you this difference between these two approaches using just code—the old blackboard is a lot of code, and the new communications mechanisms are peculiar to their use case—but if your behavior tree is properly architected, you should be able to use the same core decision logic in both a small toy example whose "blackboard" is a C++ POD and whose decisions are made by C++ lambdas and a full system that has a distributed blackboard and a complete behavior tree condition system that is purely data driven.

9.4 Conclusion

I have slogged through creating many, many behavior trees and BT-likes, and it is really easy to get bogged down. But the pattern I am outlining above is simple: figure out what you need to do, don't jump the gun on building too much of it until you have a good idea of what you need, and refine your abstractions until the complexity is squirreled away in a few files and the leaves of your functionality are *dead bone simple*. This approach can be applied everywhere, and when you do, it can radically improve your development experience.

I could preach about how test-driven development and continuous integration made this easier, or how refactoring tools help (and what to do when you cannot use them; sed, awk, and shell scripting are your friends). But the major point I want to make is that behavior trees can be surprisingly complicated—and yet surprisingly regular in structure—and it is very important to look carefully at the places you are repeating work and to aggressively seek ways to eliminate them using judicious use of class hierarchies.

Our first shot at creating a behavior tree had too many concepts, too much repeated code, and too much boilerplate. Through the process of reducing the number of entities, looking at the needs of our application domain, and *corralling* complexity into carefully chosen superclasses and support files, we not only radically reduced the amount of similar code we had to maintain, but the actual code we needed often collapsed to a single page—or occasionally a single line. When you have done that, and you can benchmark your system to show it is still efficient, you have done your job architecting your behavior tree right.

References

Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D. P., and Slack, M. G. 1997. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence* 9(2–3):237–256.

Champandard, A., and Dunstan, P. 2012. The behavior tree starter kit. In *Game AI Pro*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 73–95.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. *AAAI* 87:202–206.

Francis, A. G. 2000. Context-sensitive asynchronous memory. PhD diss. Georgia Institute of Technology. Available online http://dresan.com/research/publications/thesis.pdf (accessed June 8, 2016).

Isla, D. 2005. Handling complexity in the Halo 2 AI. *2005 Game Developer's Conference*. Available online http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php (accessed June 8, 2016).

ROS.org. 2016. Powering the world's robots. Available online http://www.ros.org (accessed June 8, 2016).