Template Tricks for Data-Driven Behavior Trees

Anthony Francis

1 Introduction

Behavior trees are an architecture for controlling NPCs based on a hierarchical graph of tasks, where each task is either atomic (a simple behavior an agent can directly perform) or composite (a behavior performed by a lower-level behavior tree of arbitrary complexity) (Isla 05, Champandard 12). Because of their clean decomposition of complex behaviors, 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.

One thing that makes behavior trees useful is that they can serve as a foundation for data-driven behavior design. However, there are a couple thorny areas in creating data-driven behavior trees worth highlighting. This article will walk through one such example, the template machinery needed to cleanly load a behavior tree from script.

2 Template Tricks for Data Driven Behavior Trees

Recently, I worked on a team that developed a behavior tree for robotic control on an internal project at Google (Francis 17). The first versions of this tree were pure C++, but when we finally deployed data-driven behavior trees based on XML, we generated more demos in a week than we had the previous quarter.

XML initially worked well for our use case, but because of its verbose syntax and complex parsing model, XML is an anti-pattern. So, when we were forced to reimplement this language for a new robot platform, we chose instead simpler and smaller data structures with a direct machine readable format – in our case, ASCII Protocol Buffers (Google 18a), though any similar format, like JSON, would do, as long as it had a simple parsing model and broad language support – a point we will elaborate later in this article.

Switching to this model enabled us to code the new parser much more quickly – mostly because we had already worked out the key logic for a C++ class factories that enabled instantiating behavior tree nodes from data. This was done with C++ templates, which are often difficult to work with – but because our template logic was properly encapsulated, after some initial hard work, the port was almost painless.

2.1 Why (and How) You Should Avoid Templates

Templates sometimes get a bad rap, especially if you've had to work with a template system designed by someone who's fully absorbed in the "zero-cost abstraction" C++ mindset, but has not yet learned to properly encapsulate them. C++ templates are painful abstractions with terrible syntax, horrible error messages, and can lead to lots of unreadable boilerplate. While they can create zero-cost abstractions, if you are creating a template it's worth asking

the question whether a simple subclassing relationship or a pointer argument will do.

Done right, however, templates can encapsulate an enormous amount of complexity into a single file hidden in your system which, after creation, should rarely be disturbed. The guts of `vector.h` in the versions available to me are several pages of summary and 1500 verbose lines – but all you need to use it is a simple stanza like:

```
std::vector<int> items;
```

For our reimplemented behavior tree, we took a similar approach, spending a fair amount of effort creating a system which enabled us to add a new task to our task library using stanzas like:

```
REGISTER_TASK(YourTaskName);
```

That's a macro, not a template, but that macro was merely the last star-stone in the wall safely entombing our data-driven behavior tree template wizardry in a file called `task_factory.h`, never to be opened until we needed to do a serious refactor.

2.2 Specifying Behavior Trees in Data

What's `REGISTER_TASK` really for? The key functionality needed to make a data-driven behavior tree is an engine that turns text into objects. Being a bit more specific, we want a software component which turns a specification of a behavior tree as data (often text) into live C++ objects that can be executed in your behavior tree architecture. To do so, you need to be able to specify the tree – and specify what C++ objects get produced from that tree.

To represent the tree, you need a language. Our first attempt was in pure C++, but that became hard to extend, update or reuse. A superior choice is a language with a clear declarative specification – data, not code. One option is XML, but I snarked earlier that XML is an anti-pattern for a reason. XML can be a good choice to support interoperability or document validation semantics, but it requires libraries to transform it into usable data. For most tasks I recommend that you choose a language which is more directly interpretable as standard programming language data structures, like JSON – or, better, something like Protocol Buffers, commonly called protobufs.

Protobufs are a data serialization format designed by Google to be smaller and faster than XML, and provide both clean semantics and a wire format. There are many potential alternatives to JSON and protobufs, but the important part is to let someone else do the heavy lifting of creating a parser, and focus on your tree.

But what does a protobuf definition of a behavior tree look like? Google's protobuf language uses "messages" which are glorified structs, and since a behavior tree is a hierarchical structure, we use a simple recursive definition, shown in Listing 1.

Listing 1 Defining a behavior tree language in protobufs.

```
message TaskConfig {
  // Unique field numbers define the wire format.
  optional string task_type = 1;
```

```
  optional string task_name = 2;
  optional TaskParameters parameters = 3;
  repeated TaskConfig children = 4;
}

message TaskParameters {
  // How many extensions TaskParameters allows.
  extensions 1000 to max;
}

message UtteranceParameters {
  // Extend stanza injects this into TaskParameters.
  extend TaskParameters {
    // Field number must be from extension range.
    optional UtteranceParameters ext = 1001;
  }

  // Actual wire format of UtteranceParameters.
  optional string human_readable_command = 1;
}
```

The first thing to note in this definition are the `TaskConfig` objects, which contain a list of children `TaskConfig` objects, enabling a single message type to recursively define an entire tree of tasks.

But each distinct type of behavior tree task will require its own fields for configuration. To handle this, we use `TaskParameters`. These aren't a fixed message; instead they're defined as extensions (Google 18b), which enable the definition of arbitrary additional messages to a base message, as long as the message is marked within a predefined extension range. Note this example is proto2 syntax; the more modern proto3 languages use an alternative to extensions called the Any type (Google 18c).

To use an extension, you define, in a slightly odd inverted inheritance syntax, a new message that extends `TaskParameters`. In the example above, to support our robots emitting a bark, we created an `UtteranceParameter` for a behavior tree node that specified the human readable command string. To inject this new message into the existing `TaskParameters`, we specify an `extend` stanza that specifies its type and extension number.

With this structure, we can fully specify the data needed to define a behavior tree entirely in protobufs, with arbitrary additional message data at any level of the tree. Thanks to the protobuf libraries, we can store this data either in a compact wire format more suitable for tooling, or a human-readable ASCII proto for debugging, as shown in Listing 2.

Listing 2 Example task definition in human-readable ASCII protobuf format.

```
task_type: "SequenceTask"
task_name: "navigation_tour"
children {
```

```
  task_type: "NavigateTask"
  task_name: "navigate_to_micro_kitchen"
  parameters {
    [robotics.executive.proto.NavigateTaskParameters.ext]
    // navigate parameters omitted for space ...
}
children {
  task_type: "MessageTask"
  task_name: "announce_arrival_in_micro_kitchen"
  parameters {
   [robotics.executive.proto.UtteranceTaskParamters.ext] {
      human_readable_command: "I have arrived."
   }
  }
}
```

That gets us the mechanisms we need to create a *specification* of a behavior tree, but the data structure the proto compiler will read in will merely be a tree of protobufs, not our desired behavior tree C++ classes. To create an actual *executable* behavior tree, we would need to either create something that parsed the protobufs like an interpreter – or, better, something that turned protobufs into the right C++ classes.

2.3 Creating a Task Factory

Our behavior tree library defines all task classes as subclasses of one root class, the `SchedulableTask`, which gets its resources from an `ExecutionContext`. The precise details of that behavior tree don't matter; what is important is that the next stage of our data-driven behavior tree system is to turn a hierarchy of `TaskConfig` protobuf message data into an executable tree of `SchedulableTask` C++ objects.

To do so, we broke the problem into two parts: finding the right C++ class based on a name, and configuring it based on a set of parameters. That's why the `TaskConfig` protobuf specifies, in addition to its parameters, a `task_type` as a string. For each such `task_type` string, one can imagine defining a `TaskMaker` that returns the right class for a given `ExecutionContext`:

```
using TaskMaker =
std::function<std::unique_ptr<SchedulableTask>(
     const string name,
     ExecutionContext* execution_context)>;
```

Note a `TaskMaker` is just a function which returns a `unique_ptr` to a `SchedulableTask`, the root of our behavior tree hierarchy; by convention, this is an empty `unique_ptr` if the task cannot be created. (We really, really have benefited from using `std::unique_ptrs` and similar ownership semantics for everything).

But once we have `TaskMaker`s, we need a facility to manage which `TaskMaker`s

are active, and that structure is a `TaskFactory`, shown in Listing 3, which enables you to add and retrieve `TaskMaker`s based on a task name.

Listing 3 Defining the `TaskFactory`.

```
class TaskFactory {
 public:
  TaskFactory();
  virtual ~TaskFactory() = default;

  // API to specify what tasks we can create.
  bool AddTaskMaker(const string& task_type,
                    TaskMaker task_maker);
  TaskMaker GetTaskMaker(const string& task_type);

 private:
  std::map<string, TaskMaker> task_registry_;
};
```

To make the `TaskMaker` really useful for loading behavior trees, we need two more classes of support functions – a `DefaultFactory` which is our go-to place to add `TaskMaker`s and from which we make `DefaultTask`s, plus a pair of `MakeTask` helper functions which call `GetTaskMaker` for you.

2.4 Template Tricks for Registering Tasks

We need a `DefaultFactory` because simply having protobuf task definitions and a `TaskFactory` doesn't actually solve the problem of knowing which `TaskMaker`s to create for which protobufs – and, ideally, based on the principle of Don't Repeat Yourself, we'd like to do this when the task classes are created. We can do so by adding static methods that support a default `TaskFactory` and a default way to make tasks, shown in Listing 4.

Listing 4 Defining the `DefaultFactory` and `DefaultTask` methods.

```
class TaskFactory {
 public:
  // ... existing member functions above
  static TaskFactory* DefaultFactory();
  static std::unique_ptr<SchedulableTask> DefaultTask(
      proto::TaskConfig task_config,
      ExecutionContext* execution_context);
```

With a default factory, we can create a `RegisterTask` helper registration class templatized on the task name which encapsulates both the task of creating a task maker and adding it to the task factory, shown in Listing 5.

Listing 5 Defining the `RegisterTask` helper class.

```
template <class Task>
class RegisterTask {
 Public:
  RegisterTask(const string& task_type,
               TaskFactory* task_factory) {
    // On construction, make a TaskMaker for the task,
    // and add it to library under the provided task name.
    task_factory->AddTaskMaker(
        // Name of the task so we can look it up later.
        task_type,

        // New TaskMaker created for templatized Task.
       [](const string task_name,
          ExecutionContext* execution_context)
            ->std::unique_ptr<SchedulableTask> {
          return std::unique_ptr<Task>(
              new Task(task_name, execution_context));
        });
  }

  // Explicit one-arg constructor using default factory.
  explicit RegisterTask(const string& task_type)
     : RegisterTask(task_type,
                    TaskFactory::DefaultFactory()) {}
};
```

When a `RegisterTask` class is created, it automatically builds a `TaskMaker` for the task type it is templated on and adds it to the default factory. The default factory is statically created, which means `RegisterTask<YourTask>` can be statically created, which means you can add a `TaskMaker` for a given task type using a stereotyped static declaration pattern, like this:

```
static RegisterTask<YourTask> labelYourTask("YourTask");
```

where "`YourTask`" in all three places that it appears is your task's class name, and "label" is an arbitrary keyword of your choice that you add to make the member function name not collide with the constructor. This template trick becomes more useful if we add a macro to register tasks in the default library, so we don't even need to see that label anymore, as shown in Listing 6.

Listing 6 Defining the `REGISTER_TASK` macro.

```
#define REGISTER_TASK(task_type) \
  static ::robotics::executive::RegisterTask<task_type> \
  label##task_type(#task_type);
```

Finding the right combination of ##'s and #'s to get this macro resolve correctly is a pain, but we need do it only once – and now the expression

```
REGISTER_TASK(YourTask)
```

expands into the default `RegisterTask<YourTask>` pattern above, eliminating the triple repetition of the task name, and enabling us to add one line to each `.cc` file, `REGISTER_TASK`, which makes the task available for loading.

One trick to be aware of here is that the registration needs to happen in your class definition, rather than its declaration – that is, in the `.cc` file, not the `.h` file. Under the hood, `REGISTER_TASK` is statically creating a tiny little class, and if you place that macro in a header, you'll likely quickly fall afoul of the One Definition Rule. The pattern we choose is to place `REGISTER_TASK` just before the class definition, like an annotation, as illustrated in Listing 7 for our `SequenceTask`.

Listing 7 Example of the use of `REGISTER_TASK`.

```
REGISTER_TASK(SequenceTask);
SequenceTask::SequenceTask(const string& name,
                           ExecutionContext*
execution_context)
     : ContainerTask(name, execution_context) {}
// More definitions ...
```

Adding the one line `REGISTER_TASK` makes `SequenceTask` available to be loaded by the default task library – if the C++ library it is defined in is linked into your binary. For our robot application, we wanted the tasks available to be fixed so Skynet could not dynamically update our robot with a `TerminateSarahConnorTask`, so we chose a static linking pattern – all the tasks we want the robot to perform must be compiled into libraries and linked in at build time. If you wish to dynamically update your task list, you will need additional machinery to ensure that all the tasks defined in your scripts get loaded at the appropriate time.

2.5 Automatically Configuring Tasks from Script

The above `REGISTER_TASK` mechanism enables us to create tasks, but there are a few more tricks. Avoiding all that repeated boilerplate depends on a default task library existing; one way to do this is to create a pointer to an object on the heap, while being careful to avoid potential order-of-destruction issues that can occur with static objects (ISOCPP 16); the code for doing this is shown in Listing 8.

Listing 8 Statically defining a `DefaultFactory`.

```
TaskFactory* TaskFactory::DefaultFactory() {
  static TaskFactory* default_factory = new TaskFactory();
```

```
  return default_factory;
}
```

The rest of the implementation of `TaskRegistry` – the `AddTaskMaker` and `GetTaskMaker` and so on – is a straightforward application of `std::map` under the hood. For our use case, we made the registry write-once to catch duplicated task definition errors, and rather than throwing an exception (rare at Google, for historical reasons) we simply return a `nullptr` if an appropriate task maker cannot be found.

More significant is are the `MakeTask` member functions; there is one "raw" function that creates the task explicitly from a name, type and context using our `GetTaskMaker`, and a second one which calls that based on information extracted from a protobuf; both are shown in Listing 9:

Listing 9 Defining the `MakeTask` functions.

```
std::unique_ptr<SchedulableTask> TaskFactory::MakeTask(
    const string& task_type,
    const string& task_name,
    ExecutionContext* execution_context) {
  // If we can find a task maker, then call it.
  TaskMaker task_maker = GetTaskMaker(task_type);
  if (task_maker != nullptr) {
    return task_maker(task_name, execution_context);
  }

  // Return an empty task as a signal of failure.
  return std::unique_ptr<SchedulableTask>();
}

std::unique_ptr<SchedulableTask> TaskFactory::MakeTask(
    proto::TaskConfig task_config,
    ExecutionContext* execution_context) {
  // Do we have enough information to make the task?
  if (task_config.has_task_type() &&
      task_config.has_task_name()) {
    std::unique_ptr<SchedulableTask> task_object(
        MakeTask(
           task_config.task_type(),
           task_config.task_name(),
           execution_context));

     // If we got a task, can it be configured?
    if (task_object &&
        task_object->Configure(task_config)) {
      return task_object;
    }
```

```
  }

  // Return an empty task as a signal for failure.
  return std::unique_ptr<SchedulableTask>();
}
```

Each task will have its own unique parameters, which we pass into the task by passing in the proto that created it via its `Configure` member function. For example, for a `ContainerTask`, the basic composite task in our class hierarchy, the `Configure` member function instantiates all its children, as shown in Listing 10:

Listing 10 Example of `Configure` for `ContainerTask`.

```
bool ContainerTask::Configure(
    const proto::TaskConfig& configuration) {
  for (const auto& child_config
       : configuration.children()) {
    // Create a child task from the default task maker.
    std::unique_ptr<SchedulableTask> child_task =
      TaskFactory::DefaultTask(
          child_config.task_type(),
          child_config.task_name(),
          GetExecutionContext());

          // If we got a child which can be configured,
          // attempt to add it and move to next child.
          if (child_task &&
              child_task->Configure(child_config) &&
              AddChild(std::move(child_task))) {
       continue;
     }
     return false;
  }
  return true;
}
```

Now each task, in its implementation, can register itself as a task that can be instantiated from a proto, and can provide a member function to configure itself from that proto. Once we have that, it's a simple matter of machinery to loading a behavior tree from a text script – which, when you unpack it, is really just creating, from the default factory for a given binary, the default task that you get from the protobuf read in from a file, as shown in Listing 11.

Listing 11 Loading a behavior tree from a file.

```
std::unique_ptr<SchedulableTask> TaskFactory::DefaultTask(
    proto::TaskConfig task_config,
```

```
    ExecutionContext* execution_context) {
  return DefaultFactory()->MakeTask(task_config,
                                    execution_context);
}

std::unique_ptr<robotics::executive::SchedulableTask>
LoadScript(
    StringPiece script_path,
    robotics::executive::ExecutionContext* context) {
  // If we can load the protobuf from a file ...
  robotics::executive::proto::TaskConfig tour_config;
  if (::OK == file::GetTextProto(
          script_path, &tour_config, file::Defaults())) {

    // ... turn the proto into a behavior tree.
    // DefaultTask returns an empty unique_ptr on failure.
    return robotics::executive::TaskFactory::DefaultTask(
        tour_config, context);
  }

  // Return an empty unique_ptr if proto can't be loaded.
  return std::unique_ptr<
      robotics::executive::SchedulableTask>();
}
```

If you really want to get clever, you can write an `ExecuteScript` member function which loads and runs a behavior tree in one swell foop, but that's probably more useful in a robotic control context, where we're deploying a small number of behaviors on a few classes of robots, than it is in a game which has a resource manager for many agents and many scripts, so we omit it.

3 Conclusion

Whew. There was a lot of template goop and some tricky details of macros and static initialization in the pattern above – but when we were done, most of that complexity was hidden behind the `REGISTER_TASK(YourTask)` macro and the `LoadScript` function, and to participate in this process, all you needed to add to the task was a `TaskParameters` protobuf extension method and a `Configure` method to parse it. This pattern – refining your abstractions until the complexity is squirreled away in a few files and the leaves of your functionality are dead simple – can be applied everywhere, and when you do, it can radically improve your development experience.

For example, templates can be used for hardware or device abstraction. Unsurprisingly, different robots have different features, and we use hardware abstraction layers to talk to multiple robots. Building behavior tree tasks at first required a lot of

boilerplate, but once we'd developed a few, we created a templatized behavior tree task for one of our robots which enabled creating a library of behaviors by only overriding three member functions – with no repeated boilerplate.

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 can't 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's very important to look carefully at the places you are repeating work, and to aggressively seek ways to eliminate them using judicious use of templates.

4 References

[Champandard 12] Champandard, A., Dunstan, P. 2012. The behavior tree starter kit. *Game AI Pro: Collected Wisdom of Game AI Professionals,* ed. S. Rabin. Boca Raton, FL: CRC Press.

[Francis 17] Francis, A. "Overcoming Pitfalls in Behavior Tree Design." 2017. *Game AI Pro 3: Collected Wisdom of Game AI Professionals,* ed. S Rabin. Boca Raton, FL: CRC Press.

[Google 18a] Protocol Buffers. Retrieved from https://developers.google.com/protocol-buffers/ on February 5, 2018.

[Google 18b] Protocol Buffers: Extensions. Retrieved from https://developers.google.com/protocol-buffers/docs/proto#extensions on February 5, 2018.

[Google 18c] Protocol Buffers: Any. Retrieved from https://developers.google.com/protocol-buffers/docs/proto3#any on February 5, 2018.

[Isla 05] D. Isla. "Handling complexity in the Halo 2 AI." 2005 Game Developer's Conference, 2005. Available online (http://www.gamasutra.com/view/feature/130663/gdc_2005_ proceeding_handling_.php).

[ISOCPP 16] What's the "static initialization order fiasco"? Retrieved from https://isocpp.org/wiki/faq/ctors#static-init-order on June 7, 2016.