Extended Behavior Trees

Matthias Hölzl

December 3, 2014

Contents

1 About this document

The source of this document is an Org-Mode file that contains text, Julia source code and some additional data such as spreadsheets or tables with the results of experiments. This file is completely self-contained, so that it is straightforward to recreate the results of the experiments once you have set up the required environment. To recompile the code in this document and re-run the experiments you need to install Julia 0.4 as well as Emacs with Org-Mode and ESS (Emacs Speaks Statistics). If you lack parts of this setup you will still be able to read the pre-generated output (in TEX and HTML format), and you can run the extracted Julia code on data you provide to a stand-alone Julia implementation, but it will be more difficult to recreate the results.

To evaluate all code in this file evaluate `org-babel-execute-buffer` in Emacs (i.e., open the file in an instance of Emacs in which the prerequisites mentioned in the previous paragraph are installed, and enter `M-x org-babel-execute-buffer`). Wait a while, and the results in the buffer will be updated with the results of executing the examples on your machine.

We add a warning (not shown in the typeset output) at the beginning of the generated file `ExtendedBehaviorTrees.jl`, so that people don't accidentally edit the generated source.

The whole file is in the `ExtendedBehaviorTrees` module; the `end` for this module declaration is at the end of the file. We disable evaluation of the module statement inside Org-Mode since it causes the Julia process to hang. (Module disabled for now to simplify reloading the code during development.)

```
module ExtendedBehaviorTrees
```

2 Behavior Trees and Extended Behavior Trees

Behavior Trees (BTs) are a flexible approach for behavioral modeling. BTs were originally introduced to model the AI of computer games, and they have recently been become more popular in areas such as robotics or avionics. BTs compose atomic behaviors (that should correspond to more or less simple actions an agent can take) using operations such as sequence (execute multiple behaviors one after the other) or choice (pick among behaviors until one succeeds). We sometimes call behaviors *actions* or *tasks*. One of the advantages of BTs is that the interface between a task and its subtasks is very simple:

- Each task can be started; once started it runs until it returns control to its parent. Because of the root in game engines, this behavior is often called *ticking*, and the corresponding function is `tick`. Scheduling is cooperative; behaviors have to return after taking a small amount of time if the BT is integrated into a frequently executing control loop.
- A behavior can return one of three status indications: it succeeded, it failed, or it is still running. This last status allows BTs to integrate long-running behaviors in an event loop.

2.1 Extended Behavior Trees

Extended behavior trees (XBTs) are an extension of behavior trees that support various kinds of reasoning and learning beyond the reactive planning of BTs. To this end, XBTs have a slightly more complicated interface between parent task and subtasks:

- The `tick` function is called with a *state* parameter and still returns three status indications. However, when a task of an XBT succeeds it passes back a measure of the quality of the solution that it has achieved, and an indication whether it could continue running to improve this solution.
- All effects caused in the world have to be mediated by the state passed in as argument.
- The state can be *virtualized*: Conceptually normal BTs operate directly on the "real world" so that actions triggered by the BT immediately affect the environment in which the agent running the BT operates. A virtual state decouples the actions of the BT temporarily from the rest of the system, so that the agent can, e.g., plan its future course of actions by observing the effects various behaviors have on the world.

2.2 Notes about the implementation

The implementation is not an exact clone of the SCEL semantics:

- There are two major strategies for implementing the evaluation of the BT:
  - Regard the XBT instance as the description of the tree and store all information for the evaluation in an external environment.
  - Have a description layer above the classes that implement the BT and store the data required for evaluating the tree directly inside the nodes.

  The first alternative is closer to the SCEL semantics, but the second one seems slightly cleaner from a programming perspective.
- There is no reason why cloning the state should require a call to an external service except that knowledge repositories are not a first class concept in SCEL. So here we simply expect the state to implement a `xbtclone` method.

3 Implementation of the basic function

In this section we set up the basics for XBTs: the results nodes may return when ticking, the execution states of a node, and finally the class structure of the node graph and the `tick` function.

3.1 Execution status of nodes

Each node can be in one of four execution states: *inactive*, *running*, *succeeded*, or *failed*. To avoid confusion with the state passed to the `tick` function we cal this execution status of a node its *status*. A node that has not yet been ticked has the *inactive* status. Once it starts execution it transitions into the *running* status; when it returns a result or fails it moves into either the *succeeded* or *failed* status. In the *succeeded* status we keep track of the value the node achieved.

```
export XbtNodeStatus, Inactive, Running, Succeeded, Failed;

abstract XbtNodeStatus;
immutable Inactive <: XbtNodeStatus end;
immutable Running <: XbtNodeStatus end;
immutable Succeeded <: XbtNodeStatus
    value
end;
immutable Failed <: XbtNodeStatus end;
```

We define predicates to test which status value we have.

```
export isinactive, isrunning, issucceeded, isfailed;

isinactive(x) = false;
isinactive(x::Inactive) = true;

isrunning(x) = false;
isrunning(x::Running) = true;

issucceeded(x) = false;
issucceeded(x::Succeeded) = true;

isfailed(x) = false;
isfailed(x::Failed) = true;
```

3.2 Results of ticking nodes

For each tick of an XBT, the nodes return either *succeeded* (with a quality value), *failed* or *running*, and an indication whether they can improve the result they have obtained so far. We therefore return simply a tuple consisting of the status of the node and a Boolean value. The status in a result may never be an instance of `Inactive`; furthermore when the status is *running*, the second value has to be `true`, when the status is *failed*, the second value has to be `false`.

```
export XbtNodeResult;
typealias XbtNodeResult (XbtNodeStatus, Bool);
```

Node results are typically used to determine whether we should continue executing this node or not. To simplify this we define a function `isdone` that tells us whether we should continue after obtaining a certain result. `isdone` can either take an XBT node (see definition below), an `XbtNodeResult`, or a `XbtNodeStatus` and a Boolean value as arguments.

```
export isdone;
isdone(x) = false;
isdone(x::XbtNodeResult) = isdone(x...);
isdone(x, cont) = false;
isdone(x::Succeeded, cont::Bool) = !cont;
isdone(x::Failed, cont::Bool) = true;
```

We define abbreviations for commonly used return values. When a computation fails or wants to keep running we can simply return one of the constants `failed` or `running`; in these cases there is no question whether the computation wants to continue or not, a failed computation never wants to continue, a running computation always wants to. In the case of successful computation we have to return a value, and either wanting to continue or not is possible. Since the former is the more likely case we make it the default.

```
export failed, running, succeeded;
const failed = (Failed(), false);
const running = (Running(), true);
succeeded(val, cont=false) = (Succeeded(val), cont);
```

3.3 XBT Nodes

Nodes in XBTs can either be composite (if they have children) or atomic. We might parameterize the classes on the type of the value successful computations return, but this complicates the definitions and does not seem to provide many benefits (since all functions have type `Function`, we cannot really use the type parameter in the places where it might affect performance).

```
export XbtNode, AtomicXbtNode, CompositeXbtNode;
abstract XbtNode;
abstract AtomicXbtNode <: XbtNode;
abstract CompositeXbtNode <: XbtNode;
```

Each task has to either store its execution status in a slot `status` or provide a method on `status` so that we can determine the execution status of tasks in a generic manner. Similarly for continuing with the node.

```
export status, setstatus;
status(node::XbtNode) = node.status;
setstatus(node::XbtNode, status::XbtNodeStatus) = node.status = status;

export cont, setcont;
cont(node::XbtNode) = node.cont;
setcont(node::XbtNode, cont::Bool) = node.cont = cont;

export result, setresult;
result(node::XbtNode) = status(node), cont(node);

function setresult(node::XbtNode, result::XbtNodeResult)
    setstatus(node, result[1])
    setcont(node, result[2])
    result
end;

isinactive(node::XbtNode) = isinactive(status(node));
isrunning(node::XbtNode) = isrunning(status(node));
```

```
issucceeded(node::XbtNode) = issucceeded(status(node));
isfailed(node::XbtNode) = isfailed(status(node));
isdone(node::XbtNode) = isdone(status(node), cont(node));

export setinactive;
function setinactive(node::XbtNode)
    setstatus(node, Inactive());
    setcont(node, true); # Slightly superfluous
end;
```

We allow two kinds of atomic nodes: `XbtTask` and `XbtFun`. Tasks are the more general nodes that are invoked as coroutines so that they can suspend their computation while they are still running. Instances of `XbtFun` are simply wrappers around functions that succeed or fail but don't suspend. (Maybe we should simply allow functions as leaves?)

```
type XbtTask <: AtomicXbtNode
    task::Task
    status::XbtNodeStatus
    cont::Bool
end;

function XbtTask(fun::Function, status::XbtNodeStatus, cont::Bool)
    XbtTask(Task(fun), status, cont);
end;
XbtTask(task, status) = XbtTask(task, status, false);
XbtTask(task) = XbtTask(task, Inactive());

function tick(node::XbtTask, state)
    if (isdone(node))
        return result(node)
    end
    setresult(node, consume(node.task))
end;

type XbtFun <: AtomicXbtNode
    fun::Function
    status::XbtNodeStatus
    cont::Bool
end;

XbtFun(fun::Function, status::XbtNodeStatus) = XbtFun(task, status, false);
XbtFun(fun::Function) = XbtFun(fun, Inactive());

function tick(node::XbtFun, state)
    if (isdone(node))
        return result(node)
    end
    setresult(node, node.fun())
end;
```

The following is a task that runs for three ticks, then succeeds with value 1 and the possibility to continue. When ticked after succeeding for the first time it will continue to run for two more ticks and then succeed with value 10 without being able to improve. After that it will continue to succeed with value 10 until it is reset to its initial state.

```
function f1()
    for i=1:3
        produce((Running(), true))
    end
    produce((Succeeded(1), true))
    produce((Running(), true))
    produce((Running(), true))
    produce((Succeeded(10), false))
end;
task1 = XbtTask(f1);
for i=1:8
    println(tick(task1, ()));
end;
```

The results of this are example are as follows:

```
(Running(),true)
(Running(),true)
(Running(),true)
(Succeeded(1),true)
(Running(),true)
(Running(),true)
(Succeeded(10),false)
(Succeeded(10),false)
```

The simplest kinds of composite XBT nodes are sequences and choices. A Sequence executes its child nodes sequentially until a child node fails. In that case the sequence node fails as well. If all child nodes succeed the sequence node succeeds. Choice nodes work in the reverse manner: They execute their children in turn until the first child succeeds in which case the choice succeeds. If all children fail the choice fails. Since we will later have several nodes that are sequence- or choice-like we define abstract types for these two behaviors.

```
abstract XbtSequenceNode <: CompositeXbtNode;

function tick(node::XbtSequenceNode, state)
    local sum = 0, status, cont;
    for child in node.children
        status, cont = tick(child, state);
        if isa(status, Failed) return (status, false) end;
        if isa(status, Running) return (status, true) end;
        sum += status.value;
    end;
    # TODO: what about continuing?
    return Succeeded(sum, false), cont;
end;

immutable XbtSeq <: XbtSequenceNode
    children::AbstractArray{XbtNode,1}
end;

abstract XbtChoiceNode <: CompositeXbtNode;

function tick(node::XbtChoiceNode, state)
    local status, cont;
```

```
    for child in node.children
        status, cont = tick(child, state);
        if isa(status, Succeeded) return (status, cont) end;
        if isa(status, Running) return (status, true) end;
    end;
    return Failed(), cont;
end;

immutable XbtChoice <: XbtChoiceNode
    children::AbstractArray{XbtNode,1}
end;
```

4 HTN Planning

4.1 States and Goals

4.2 The Planner

5 Reinforcement Learning

```
end; # module ExtendedBehaviorTrees
```