Version: 4.0.1

Reactive and Asynchronous behaviors

The next example shows the difference between a `SequenceNode` and a `ReactiveSequence`.

We will implement an **Asynchronous Action**, i.e. an action that takes a long time to be completed and will return RUNNING while the completion criterias are not met.

An Asynchronous Action has the following requirements:

- It should not block in the method `tick()` too much time. The flow of execution should be returned as fast as possible.
- It should be aborted as fast as possible, if the `halt()` method is called.

> CAUTION
>
> **Learn more about Asynchronous Actions**
>
> Users should fully understand how Concurrency is achieved in BT.CPP and learn best practices about how to develop their own Asynchronous Actions. You will find an extensive article here.

StatefulAsyncAction

The **StatefulAsyncAction** is the preferred way to implement asynchronous Actions.

It is particularly useful when your code contains a **request-reply pattern**, i.e. when the action sends an asynchronous request to another process, and checks periodically if the reply has been received.

Based on that reply, it may return SUCCESS or FAILURE.

If instead of communicating with an external process, you are performing some computation that takes a long time, you may want to split it into small "chunks" or you may want to move that computation to another thread (see AsyncThreadedAction tutorial).

A derived class of **StatefulAsyncAction** must override the following virtual methods, instead of `tick()`:

- **NodeStatus onStart()**: called when the Node was in IDLE state. It may succeed or fail immediately or return RUNNING. In the latter case, the next time the tick is received the method `onRunning` will be executed.
- **NodeStatus onRunning()**: called when the Node is in RUNNING state. Return the new status.
- **void onHalted()**: called when this Node was aborted by another Node in the tree.

Let's create a dummy Node called **MoveBaseAction**:

```
// Custom type
struct Pose2D
{
    double x, y, theta;
};

namespace chr = std::chrono;

class MoveBaseAction : public BT::StatefulAsyncAction
{
  public:
    // Any TreeNode with ports must have a constructor with this signature
    MoveBaseAction(const std::string& name, const BT::NodeConfig& config)
      : StatefulAsyncAction(name, config)
    {}

    // It is mandatory to define this static method.
    static BT::PortsList providedPorts()
    {
        return{ BT::InputPort<Pose2D>("goal") };
```

```
    }

    // this function is invoked once at the beginning.
    BT::NodeStatus onStart() override;

    // If onStart() returned RUNNING, we will keep calling
    // this method until it return something different from RUNNING
    BT::NodeStatus onRunning() override;

    // callback to execute if the action was aborted by another node
    void onHalted() override;

  private:
    Pose2D _goal;
    chr::system_clock::time_point _completion_time;
};

//------------------------

BT::NodeStatus MoveBaseAction::onStart()
{
  if ( !getInput<Pose2D>("goal", _goal))
  {
    throw BT::RuntimeError("missing required input [goal]");
  }
  printf("[ MoveBase: SEND REQUEST ]. goal: x=%f y=%f theta=%f\n",
         _goal.x, _goal.y, _goal.theta);

  // We use this counter to simulate an action that takes a certain
  // amount of time to be completed (200 ms)
  _completion_time = chr::system_clock::now() + chr::milliseconds(220);

  return BT::NodeStatus::RUNNING;
}

BT::NodeStatus MoveBaseAction::onRunning()
{
  // Pretend that we are checking if the reply has been received
  // you don't want to block inside this function too much time.
  std::this_thread::sleep_for(chr::milliseconds(10));

  // Pretend that, after a certain amount of time,
  // we have completed the operation
  if(chr::system_clock::now() >= _completion_time)
  {
    std::cout << "[ MoveBase: FINISHED ]" << std::endl;
    return BT::NodeStatus::SUCCESS;
```

```
  }
  return BT::NodeStatus::RUNNING;
}

void MoveBaseAction::onHalted()
{
  printf("[ MoveBase: ABORTED ]");
}
```

Sequence VS ReactiveSequence

The following example should use a simple `SequenceNode`.

```
<root BTCPP_format="4">
    <BehaviorTree>
       <Sequence>
           <BatteryOK/>
           <SaySomething   message="mission started..." />
           <MoveBase           goal="1;2;3"/>
           <SaySomething   message="mission completed!" />
       </Sequence>
    </BehaviorTree>
</root>
```

```
int main()
{
  BT::BehaviorTreeFactory factory;
  factory.registerSimpleCondition("BatteryOK", std::bind(CheckBattery));
  factory.registerNodeType<MoveBaseAction>("MoveBase");
  factory.registerNodeType<SaySomething>("SaySomething");

  auto tree = factory.createTreeFromText(xml_text);

  // Here, instead of tree.tickWhileRunning(),
  // we prefer our own loop.
  std::cout << "--- ticking\n";
  status = tree.tickWhileRunning();
  std::cout << "--- status: " << toStr(status) << "\n\n";

  while(status == NodeStatus::RUNNING)
  {
    // Sleep to avoid busy loops.
```

```
    // do NOT use other sleep functions!
    // Small sleep time is OK, here we use a large one only to
    // have less messages on the console.
    tree.sleep(std::chrono::milliseconds(100));

    std::cout << "--- ticking\n";
    status = tree.tickOnce();
    std::cout << "--- status: " << toStr(status) << "\n\n";
  }

  return 0;
}
```

Expected output:

```
--- ticking
[ Battery: OK ]
--- status: RUNNING

--- ticking
Robot says: mission started...
--- status: RUNNING

--- ticking
[ MoveBase: SEND REQUEST ]. goal: x=1.0 y=2.0 theta=3.0
--- status: RUNNING

--- ticking
--- status: RUNNING

--- ticking
[ MoveBase: FINISHED ]
Robot says: mission completed!
--- status: SUCCESS
```

You may have noticed that when `executeTick()` was called, `MoveBase` returned **RUNNING** the 1st and 2nd time, and eventually **SUCCESS** the 3rd time.

`BatteryOK` is executed only once.

If we use a `ReactiveSequence` instead, when the child `MoveBase` returns RUNNING, the sequence is restarted and the condition `BatteryOK` is execut **again**.

If at any point, `BatteryOK` returned **FAILURE**, the `MoveBase` action would be *interrupted* (*halted*, to be specific).

```
<root>
    <BehaviorTree>
       <ReactiveSequence>
           <BatteryOK/>
           <Sequence>
               <SaySomething   message="mission started..." />
               <MoveBase           goal="1;2;3"/>
               <SaySomething   message="mission completed!" />
           </Sequence>
       </ReactiveSequence>
    </BehaviorTree>
</root>
```

Expected output:

```
--- ticking
[ Battery: OK ]
Robot says: mission started...
--- status: RUNNING

--- ticking
[ Battery: OK ]
[ MoveBase: SEND REQUEST ]. goal: x=1.0 y=2.0 theta=3.0
--- status: RUNNING

--- ticking
[ Battery: OK ]
--- status: RUNNING

--- ticking
[ Battery: OK ]
[ MoveBase: FINISHED ]
Robot says: mission completed!
--- status: SUCCESS
```

Event Driven trees?

> TIP
>
> We used the command `tree.sleep()` instead of `std::this_thread::sleep_for()` for a reason!!!

The method `Tree::sleep()` should be preferred, because it can be interrupted by a Node in the tree when "something changed".

`Tree::sleep()` will be interrupted when the method `TreeNode::emitStateChanged()` is invoked.

Edit this page