Programming with Behavior Trees and ROS

Since a ready-made behavior tree library for ROS was not available at the time of this writing, a new ROS package called pi_trees was created for use with this book. In this section and those that follow, we will install the pi_trees package and learn how to use it to program our Patrol Bot and house cleaning robot using behavior trees.

**Installing the pi_trees library**

Before running the examples that follow, we need to install the pi_trees ROS package using the following commands:

```bash
$ sudo apt-get install graphviz-dev libgraphviz-dev \
   python-pygraph python-pygraphviz gv
$ cd ~/catkin_ws/src
$ git clone -b indigo-devel https://github.com/pirobot/pi_trees.git
$ cd ~/catkin_ws
$ catkin_make
$ rospack profile
```

**Basic components of the pi_trees library**

Behavior trees are fairly easy to implement in Python and while there are several different approaches one can take, the methods used in the pi_trees package lend themselves well to integrating with ROS topics, services and actions. In fact, the pi_trees package was modeled after SMACH so that some of the code might already seem familiar.

The core pi_trees library is contained in the file pi_trees_lib.py in the pi_trees/pi_trees_lib/src directory and the ROS classes can be found in the file pi_trees_ros.py under the pi_trees/pi_trees_ros/src directory. Let's start with pi_trees_lib.py.

**Link to source:** [pi_trees_lib.py](#)

```python
class TaskStatus():
    FAILURE = 0
    SUCCESS = 1
    RUNNING = 2

def __init__(self, name, children=None, *args, **kwargs):
    self.name = name
    self.status = None
    if children is None:
        children = []
    self.children = children

def run(self):
    pass

def reset(self):
    for c in self.children:
        c.reset()

def add_child(self, c):
    self.children.append(c)
```

First we define the possible task status values using the class TaskStatus as a kind of enum. One can include additional status values such as ERROR or UNKNOWN but these three seem to be sufficient for most applications.
The base Task class defines the core object of the behavior tree. At a minimum it must have a name and a run() function that in general will not only perform some behavior but also return its status. The other key functions are add_child() and remove_child() that enable us to add or remove sub-tasks to composite tasks such as selectors and sequences (described below). You can also use the prepend_child() or insert_child() functions to add a sub-task with a specific priority relative to the other tasks already in the list.

When creating your own tasks, you will override the run() function with code that performs your task’s actions. It will then return an appropriate task status depending on the outcome of the action. This will become clear when we look at the Patrol Bot example later on.

The reset() function is useful when we want to zero out any counters or other variables internal to a particular task and its children.

A Selector runs each child task in list order until one succeeds or until it runs out of subtasks. Note that if a child task returns a status of RUNNING, the selector also returns RUNNING until the child either succeeds or fails.
def __init__(self, name, *args, **kwargs):
    super(Sequence, self).__init__(name, *args, **kwargs)

def run(self):
    for c in self.children:
        c.status = c.run()
        if c.status != TaskStatus.SUCCESS:
            return c.status
    return TaskStatus.SUCCESS

A Sequence runs each child task in list order until one fails or until it runs out of subtasks. Note that if a child task returns a status of RUNNING, the sequence also returns RUNNING until the child either succeeds or fails.

class Iterator(Task):
    """Iterate through all child tasks ignoring failure."
    ""
    def __init__(self, name, *args, **kwargs):
        super(Iterator, self).__init__(name, *args, **kwargs)
    
def run(self):
        for c in self.children:
            c.status = c.run()
            if c.status != TaskStatus.SUCCESS and c.status != TaskStatus.FAILURE:
                return c.status
        return TaskStatus.SUCCESS

An Iterator behaves like a Sequence but ignores failures.

class ParallelOne(Task):
    """A parallel task runs each child task at (roughly) the same time. The ParallelOne task returns success as soon as any child succeeds."
    ""
    def __init__(self, name, *args, **kwargs):
        super(ParallelOne, self).__init__(name, *args, **kwargs)
    
def run(self):
        for c in self.children:
            c.status = c.run()
            if c.status == TaskStatus.SUCCESS:
                return TaskStatus.SUCCESS
        return TaskStatus.FAILURE

The key difference between the ParallelOne composite task and a Selector is that the ParallelOne task runs all of its tasks on every "tick" of the clock unless (or until) one subtask succeeds. A Selector continues running the first subtask until that task either succeeds or fails before moving on to the next subtask or returning altogether.

class ParallelAll(Task):
    """A parallel task runs each child task at (roughly) the same time. The ParallelAll task requires all subtasks to succeed for it to succeed."
    ""
    def __init__(self, name, *args, **kwargs):
        super(ParallelAll, self).__init__(name, *args, **kwargs)
    
def run(self):
        n_success = 0
        for c in self.children:
            c.status = c.run()
            if c.status == TaskStatus.SUCCESS:
                n_success += 1
            else:
                return TaskStatus.FAILURE
        return TaskStatus.SUCCESS

        return TaskStatus.FAILURE
n_children = len(self.children)

for c in self.children:
    c.status = c.run()
    if c.status == TaskStatus.SUCCESS:
        n_success += 1
    elif c.status == TaskStatus.FAILURE:
        return TaskStatus.FAILURE

if n_success == n_children:
    return TaskStatus.SUCCESS
else:
    return TaskStatus.RUNNING

Similar to the ParallelOne task, the ParallelAll task runs each subtask on each tick of the clock but continues until all subtasks succeed or until one of them fails.

class Loop(Task):
    """    Loop over one or more subtasks for the given number of iterations
    Use the value -1 to indicate a continual loop.
    """
    def __init__(self, name, announce=True, *args, **kwargs):
        super(Loop, self).__init__(name, *args, **kwargs)
        self.iterations = kwargs.get('iterations', 0)
        self.announce = announce
        self.loop_count = 0
        self.name = name
        print("Loop iterations: "+str(self.iterations))

    def run(self):
        while True:
            if self.iterations != -1 and self.loop_count >= self.iterations:
                return TaskStatus.SUCCESS

            for c in self.children:
                while True:
                    c.status = c.run()
                    if c.status == TaskStatus.SUCCESS:
                        break

                    return c.status
                c.reset()

            self.loop_count += 1

            if self.announce:
                print(self.name + " COMPLETED "+str(self.loop_count) + " LOOP(S)")

The Loop task simply executes its child task(s) for the given number of iterations. A value of -1 for the iterations parameters means "loop forever". Note that a Loop task is still a task in its own right.

class IgnoreFailure(Task):
    """    Always return either RUNNING OR SUCCESS.
    """
    def __init__(self, name, *args, **kwargs):
        super(IgnoreFailure, self).__init__(name, *args, **kwargs)

    def run(self):
        for c in self.children:
            c.status = c.run()
if c.status != TaskStatus.RUNNING:
    return TaskStatus.SUCCESS
else:
    return TaskStatus.RUNNING
return TaskStatus.SUCCESS

The IgnoreFailure task simply turns a FAILURE into a SUCCESS for each of its child behaviors. If the status of a child task is RUNNING, the IgnoreFailure also takes on a status of RUNNING.

The CallbackTask turns any function into a task. The function name is passed to the constructor as the cb argument along with optional arguments. The only constraint on the callback function is that it must return 0 or False to represent a TaskStatus of FAILURE and 1 or True to represent SUCCESS. Any other return value is interpreted as RUNNING.

class CallbackTask(Task):
    """Turn any callback function (cb) into a task"
    ""
    def __init__(self, name, cb=None, cb_args=[], cb_kwargs={}, **kwargs):
        super(CallbackTask, self).__init__(name, cb=None, cb_args=[], cb_kwargs={}, **kwargs)
        self.name = name
        self.cb = cb
        self.cb_args = cb_args
        self.cb_kwargs = cb_kwargs

    def run(self):
        status = self.cb(*self.cb_args, **self.cb_kwargs)
        if status == 0 or status == False:
            return TaskStatus.FAILURE
        elif status == 1 or status == True:
            return TaskStatus.SUCCESS
        else:
            return TaskStatus.RUNNING

ROS-specific behavior tree classes

You can find the ROS-specific behavior tree classes in the file pi_trees_ros.py in the directory pi_trees/pi_trees_ros/src. This library contains three key ROS tasks: the MonitorTask for monitoring a ROS topic; the ServiceTask for connecting to a ROS service; and the SimpleActionTask for send goals to a ROS action server and receiving feedback. We will describe these tasks only briefly here as their use will become clear in the programming examples that follow.

Link to source: pi_trees_ros.py

Let’s begin by looking at the MonitorTask class:

class MonitorTask(Task):
    """Turn a ROS subscriber into a Task."
    ""
    def __init__(self, name, topic, msg_type, msg_cb, wait_for_message=True, timeout=5):
        super(MonitorTask, self).__init__(name)
        self.topic = topic
        self.msg_type = msg_type
        self.timeout = timeout
        self.msg_cb = msg_cb
        rospy.loginfo("Subscribing to topic " + topic)
The MonitorTask subscribes to a given ROS topic and executes a given callback function. The callback function is defined by the user and is responsible for returning one of the three allowed task status values: SUCCESS, FAILURE or RUNNING.

The ServiceTask wraps a given ROS service and optionally executes a user-defined callback function. By default, a ServiceTask will simply call the corresponding ROS service and return SUCCESS unless the service call itself fails in which case it returns FAILURE. If the user passes in a callback function, this function may simply execute some arbitrary code or it may also return a task status.

The SimpleActionTask turns a ROS action into a Task.
def _init_(self, name, action, action_type, goal, rate=5, connect_timeout=10, result_timeout=30, reset_after=False, active_cb=None, done_cb=None, feedback_cb=None):
    super(SimpleActionTask, self)._init_(name)
    self.action = action
    self.goal = goal
    self.tick = 1.0 / rate
    self.rate = rospy.Rate(rate)
    self.result = None
    self.connect_timeout = connect_timeout
    self.result_timeout = result_timeout
    self.reset_after = reset_after
    if done_cb == None:
        done_cb = self.default_done_cb
    self.done_cb = done_cb
    if active_cb == None:
        active_cb = self.default_active_cb
    self.active_cb = active_cb
    if feedback_cb == None:
        feedback_cb = self.default_feedback_cb
    self.feedback_cb = feedback_cb
    self.action_started = False
    self.action_finished = False
    self.goal_status_reported = False
    self.time_so_far = 0.0
    self.goal_states = ['PENDING', 'ACTIVE', 'PREEMPTED', 'SUCCEEDED', 'ABORTED', 'REJECTED', 'PREEMPTING', 'RECALLING', 'RECALLED', 'LOST']
    rospy.loginfo("Connecting to action " + action)
    self.action_client = actionlib.SimpleActionClient(action, action_type)
    rospy.loginfo("Waiting for move_base action server...")
    try:
        self.action_client.wait_for_server(rospy.Duration(self.connect_timeout))
    except:
        rospy.loginfo("Timed out connecting to the action server " + action)
    rospy.loginfo("Connected to action server")

def run(self):
    # Send the goal
    if not self.action_started:
        rospy.loginfo("Sending " + str(self.name) + " goal to action server...")
        self.action_client.send_goal(self.goal, done_cb=self.done_cb, active_cb=self.active_cb, feedback_cb=self.feedback_cb)
        self.action_started = True
        # We cannot use the wait_for_result() method here as it will block
        # the entire tree so we break it down in time slices of duration
        # 1 / rate.
        if not self.actionFinished:
            self.time_so_far += self.tick
            self.rate.sleep()
            if self.time_so_far > self.result_timeout:
                self.action_client.cancel_goal()
                rospy.loginfo("Timed out achieving goal")
                return TaskStatus.FAILURE
            else:
                return TaskStatus.RUNNING
        else:
            return TaskStatus.RUNNING
The SimpleActionTask mimics the SimpleActionState defined in SMACH. Its main function is to wrap a ROS simple action client and therefore takes an action name, action type, and a goal as arguments. It can also take arguments specifying user-defined callback functions for the standard active_cb, done_cb and feedback_cb callbacks that are passed to the ROS simple action client. In particular, the SimpleActionTask defines default done_cb function reports the final status of the action which is then turned into a corresponding task status to be used in the rest of the behavior tree.

We will examine the SimpleActionTask more closely in the context of a number of example programs that we turn to next.

**A Patrol Bot example using behavior trees**

We have already seen how we can use SMACH to program a robot to patrol a series of waypoints while monitoring its battery level and recharging when necessary. Let's now see how we can do the same using the pi_trees package.

Our test program is called patrol_tree.py and is located in the rbx2_tasks/nodes subdirectory. Before looking at the code, let's try it out.

Begin by bringing up the fake TurtleBot, blank map, and fake battery simulator:

```
$ roslaunch rbx2_tasks fake_turtlebot.launch
```

Next, bring up RViz with the nav_tasks.rviz config file:
Finally, run the `patrol_tree.py` script:

```
$ rosrun rbx2_tasks patrol_tree.py
```

The robot should make two loops around the square, stopping to recharge when necessary, then stop. Let's now look at the code.

**Link to source:** [patrol_tree.py](#)

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist
from rbx2_msgs.srv import *
from pi_trees_ros.pi_trees_ros import *
from rbx2_tasks.task_setup import *

class Patrol():
    def __init__(self):
        rospy.init_node("patrol_tree")

        # Set the shutdown function (stop the robot)
        rospy.on_shutdown(self.shutdown)

        # Initialize a number of parameters and variables
        setup_task_environment(self)

        # Create a list to hold the move_base tasks
        MOVE_BASE_TASKS = list()

        n_waypoints = len(self.waypoints)

        # Create simple action navigation task for each waypoint
        for i in range(n_waypoints + 1):
            goal = MoveBaseGoal()
            goal.target_pose.header.frame_id = 'map'
            goal.target_pose.header.stamp = rospy.Time.now()
            goal.target_pose.pose = self.waypoints[i % n_waypoints]

            move_base_task = SimpleActionTask("MOVE_BASE_TASK_" + str(i), "move_base",
                                                MoveBaseAction, goal)

            MOVE_BASE_TASKS.append(move_base_task)

        # Set the docking station pose
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = 'map'
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose = self.docking_station_pose

        # Assign the docking station pose to a move_base action task
        NAV_DOCK_TASK = SimpleActionTask("NAV_DOC_TASK", "move_base", MoveBaseAction, goal,
                                                reset_after=True)

        # Create the root node
        BEHAVE = Sequence("BEHAVE")

        # Create the "stay healthy" selector
        STAY_HEALTHY = Selector("STAY_HEALTHY")

        # Create the patrol loop decorator
        LOOP_PATROL = Loop("LOOP_PATROL", announce=True, iterations=self.n_patrols)
```
# Add the two subtrees to the root node in order of priority
BEHAVE.add_child(STAY_HEALTHY)
BEHAVE.add_child(LOOP_PATROL)

# Create the patrol iterator
PATROL = Iterator("PATROL")

# Add the move_base tasks to the patrol task
for task in MOVE_BASE_TASKS:
    PATROL.add_child(task)

# Add the patrol to the patrol loop
LOOP_PATROL.add_child(PATROL)

# Add the battery check and recharge tasks to the "stay healthy" task
with STAY_HEALTHY:
    # The check battery condition (uses MonitorTask)
    CHECK_BATTERY = MonitorTask("CHECK_BATTERY", "battery_level", Float32, self.check_battery)

    # The charge robot task (uses ServiceTask)
    CHARGE_ROBOT = ServiceTask("CHARGE_ROBOT", "battery_simulator/set_battery_level", SetBatteryLevel, 100, result_cb=self.recharge_cb)

    # Build the recharge sequence using inline construction
    RECHARGE = Sequence("RECHARGE", [NAV_DOCK_TASK, CHARGE_ROBOT])

    # Add the check battery and recharge tasks to the stay healthy selector
    STAY_HEALTHY.add_child(CHECK_BATTERY)
    STAY_HEALTHY.add_child(RECHARGE)

# Display the tree before beginning execution
print "Patrol Behavior Tree"
print_tree(BEHAVE)

# Run the tree
while not rospy.is_shutdown():
    BEHAVE.run()
    rospy.sleep(0.1)

def check_battery(self, msg):
    if msg.data is None:
        return TaskStatus.RUNNING
    else:
        if msg.data < self.low_battery_threshold:
            rospy.loginfo("LOW BATTERY - level: " + str(int(msg.data)))
            return TaskStatus.FAILURE
        else:
            return TaskStatus.SUCCESS

def recharge_cb(self, result):
    rospy.loginfo("BATTERY CHARGED!")

def shutdown(self):
    rospy.loginfo("Stopping the robot...")
    self.move_base.cancel_all_goals()
    self.cmd_vel_pub.publish(Twist())
    rospy.sleep(1)

if __name__ == '__main__':
tree = Patrol()
We begin by importing the `pi_trees_ros` library which in turn imports the core `pi_trees` classes from the `pi_trees_lib` library. The first key block of code involves the creating of the navigation tasks shown below:

```python
for i in range(n_waypoints + 1):
    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = 'map'
    goal.target_pose.header.stamp = rospy.Time.now()
    goal.target_pose.pose = self.waypoints[i % n_waypoints]

    move_base_task = SimpleActionTask("MOVE_BASE_TASK_" + str(i), "move_base",
                                       MoveBaseAction, goal, reset_after=False)

    MOVE_BASE_TASKS.append(move_base_task)

# Set the docking station pose
goal = MoveBaseGoal()
goal.target_pose.header.frame_id = 'map'
goal.target_pose.header.stamp = rospy.Time.now()
goal.target_pose.pose = self.docking_station_pose

# Assign the docking station pose to a move_base action task
NAV_DOCK_TASK = SimpleActionTask("NAV_DOC_TASK", "move_base",
                                     MoveBaseAction, goal, reset_after=True)
```

Here we see nearly the same procedure as we used with SMACH although now we are using the `SimpleActionTask` from the `pi_trees` library instead of the `SimpleActionState` from the SMACH library.

Note the parameter called `reset_after` in the construction of a `SimpleActionTask`. We set this to `False` for the `move_base` tasks assigned to waypoints but we set it to `True` for the docking `move_base` task for the following reason. Recall that when a behavior or task in a behavior tree succeeds or fails, it retains that status indefinitely unless it is reset. This "memory" property is essential because on every execution cycle, we poll the status of every node in the tree. This enables us to continually check condition nodes whose status might have changed since the last cycle. However, if the robot has just successfully reached a waypoint, we want that status to be retained on the next pass through the tree so that the parent node will advance the sequence to the next waypoint. On the other hand, when it comes to recharging, we need to reset the navigation task once the robot is docked so that it can be executed again the next time the battery runs low.

Once we have the navigation and docking tasks created, we move on to building the rest of the behavior tree. The order in which we create the nodes in the script is somewhat flexible since it is the parent-child relations that determine the actual structure of the tree. If we start at the root of the tree, our first behavior nodes would look like this:

```python
BEHAVE = Sequence("BEHAVE")

# Create the "stay healthy" selector
STAY_HEALTHY = Selector("STAY_HEALTHY")

# Create the patrol loop decorator
LOOP_PATROL = Loop("LOOP_PATROL", iterations=self.n_patrols)

# Build the full tree from the two subtrees
BEHAVE.add_child(STAY_HEALTHY)
BEHAVE.add_child(LOOP_PATROL)
```

The root behavior is a `Sequence` labeled `BEHAVE` that will have two child branches; one that starts with the `Selector` labeled `STAY_HEALTHY` and a second branch labeled `LOOP_PATROL` that uses the `Loop`
decorator to loop over the patrol task. We then add the two child branches to the root node in the order that defines their priority. In this case, the STAY_HEALTHY branch has higher priority than LOOP_PATROL.

```python
58 # Create the patrol iterator
59 PATROL = Iterator("PATROL")
60 61 # Add the move_base tasks to the patrol task
62 for task in MOVE_BASE_TASKS:
63    PATROL.add_child(task)
64 65 # Add the patrol to the patrol loop
66 LOOP_PATROL.add_child(PATROL)
```

Next we take care of the rest of the patrol nodes. The patrol sequence itself is constructed as an Iterator called PATROL. We then add each move_base task to the iterator. Finally, we add the entire patrol to the LOOP_PATROL task.

```python
68 # Add the battery check and recharge tasks to the "stay healthy" task
69 with STAY_HEALTHY:
70    # The check battery condition (uses MonitorTask)
71    CHECK_BATTERY = MonitorTask("CHECK_BATTERY", "battery_level", Float32, self.check_battery)
72    # The charge robot task (uses ServiceTask)
73    CHARGE_ROBOT = ServiceTask("CHARGE_ROBOT", "battery_simulator/set_battery_level", SetBatteryLevel, 100, result_cb=self.recharge_cb)
74    # Build the recharge sequence using inline construction
75    RECHARGE = Sequence("RECHARGE", [NAV_DOCK_TASK, CHARGE_ROBOT])
```

Here we flesh out the STAY_HEALTHY branch of the tree. First we define the CHECK_BATTERY task as a MonitorTask on the ROS topic battery_level using the callback function self.check_battery (described below). Next we define the CHARGE_ROBOT behavior as a ServiceTask that connects to the ROS service battery_simulator/set_battery_level and sends a value of 100 to recharge the fake battery.

We then construct the RECHARGE task as a Sequence whose child tasks are NAV_DOCK_TASK and CHARGE_ROBOT. Note how we have used the inline syntax to illustrate how we can add child tasks at the same time that we construct the parent. Equivalently, we could have used the three lines:

```python
RECHARGE = Sequence("RECHARGE")
RECHARGE.add_child(NAV_DOCK_TASK)
RECHARGE.add_child(CHARGE_ROBOT)
```

You can use whichever syntax you prefer.

```python
80       STAY_HEALTHY.add_child(CHECK_BATTERY)
81       STAY_HEALTHY.add_child(RECHARGE)
```

We complete the STAY_HEALTHY branch of the tree by adding the CHECK_BATTERY and RECHARGE tasks. Note again that the order is important since we want to check the battery first to see if we need to recharge.

```python
83       # Display the tree before beginning execution
84       print "Patrol Behavior Tree"
85       print_tree(BEHAVE)
86 87       # Run the tree
88       while not rospy.is_shutdown():
89           BEHAVE.run()
```
Before starting execution, we use the `print_tree()` function from the `pi_trees` library to display a representation of the behavior tree on the screen. The tree itself is executed by calling the `run()` function on the root node. The `run()` function makes one pass through the nodes of the tree so we need to place it in a loop.

Finally, we have the `check_battery()` callback:

```python
def check_battery(self, msg):
    if msg.data is None:
        return TaskStatus.RUNNING
    else:
        if msg.data < self.low_battery_threshold:
            rospy.loginfo("LOW BATTERY - level: " + str(int(msg.data)))
            return TaskStatus.FAILURE
        else:
            return TaskStatus.SUCCESS
```

Recall that this function was assigned to the `CHECK_BATTERY` MonitorTask which monitors the `battery_level` topic. We therefore check the battery level against the `low_battery_threshold` parameter. If the level is below threshold, we return a task status of `FAILURE`. Otherwise we return `SUCCESS`. Because the `CHECK_BATTERY` task is the highest priority task in the `STAY_HEALTHY` selector, if it returns `FAILURE`, then the selector moves on to its next subtask which is the `RECHARGE` task.

The `patrol_tree.py` script illustrates an important property of behavior trees that helps distinguish them from ordinary hierarchical state machines like `SMACH`. You'll notice that after a recharge, the robot continues its patrol where it left off even though nowhere in the script did we explicitly save the last waypoint reached. Remember that in the `SMACH` example (`patrol_smach_concurrence.py`), we had to save the last state just before a recharge so that the robot would know where to continue after being charged. Behavior trees inherently store their state by virtue of each node's `status` property. In particular, if the robot is on its way to a waypoint, the navigation task doing the work of moving the robot has a status of `RUNNING`. If the robot is then diverted to the docking station for a recharge, the status of the previously active navigation status is still `RUNNING`. This means that when the robot is fully charged and the `CHECK_BATTERY` task returns `SUCCESS`, control returns automatically to the running navigation node.

**A housing cleaning robot using behavior trees**

Earlier in the chapter we used `SMACH` to simulate a house cleaning robot. Let us now do the same using behavior trees. Our new script is called `clean_house_tree.py` and is found in the `rbx2_tasks/nodes` subdirectory. The program is similar to the `patrol_tree.py` script but this time we will add a few tasks that simulate vacuuming, scrubbing and mopping just as we did with the `SMACH` example. We will also include battery checking and recharge behavior.

Before describing the code, let's try it out. If you don't already have the `fake_turtlebot.launch` file running, bring it up now:

```
$ roslaunch rbx2_tasks fake_turtlebot.launch
```

Recall that this launch file also runs a `move_base` node, the map server with a blank map, and the fake battery node with a default runtime of 60 seconds.

Next, bring up RViz with the `nav_tasks.rviz` config file:
Finally, run the `clean_house_tree.py` script:

\[
\text{\$ rosrun rbx2_tasks clean_house_tree.py}
\]

The robot should make one circuit of the square, performing cleaning tasks in each room and recharging when necessary.

The overall behavior tree implemented by the `clean_house_tree.py` script looks like this:

```
- BEHAVE
  - STAY HEALTHY
  - CLEAN HOUSE
    - CHECK_BATTERY
    - RECHARGE
    - LIVING ROOM
    - KITCHEN
    - BATHROOM
    - HALLWAY
      - NAV DOCK
      - NAV CHARGE
      - NAV VACUUM
      - NAV MOP
      - NAV SCUB
      - NAV MOP
      - NAV VACUUM
```

In addition to the main tasks shown above, we also require a few condition nodes such as "is the room already clean" and "are we at the desired location?" The need for these condition checks arises from the recharge behavior of the robot. For example, suppose the robot is in the middle of mopping the kitchen floor when its battery level falls below threshold. The robot will navigate out of the kitchen and over to the docking station. Once the robot is recharged, control will return to the last running task—mopping the kitchen floor—but now the robot is no longer in the kitchen. If we don't check for this, the robot will start mopping the docking station! So to get back to the kitchen, we include a task that checks the robot's current location and compares it to where it is supposed to be. If not, the robot navigates back to that location.

A good way to understand the behavior tree we will create is to imagine that you are asked to clean a room yourself. If you were asked to clean the bathroom, you might use a strategy like the following:

- First find out if the bathroom even needs cleaning. Perhaps your roommate felt energetic and took care of it already. If there is a task checklist somewhere such as on the refrigerator, make sure the bathroom isn't already checked off.
- If the bathroom does need cleaning, you need to be in the bathroom to clean it. If you are already in the bathroom, then you can start cleaning. If not, you have to navigate your way through the house to the bathroom.
- Once you are in the bathroom, check the list of tasks to perform. After each task is completed, put a check mark beside the task on the list.

We can mimic this very same process in a behavior tree. For a given room, the subtree will look something like the following. In parenthesis beside each task, we have indicated the type of task it is: selector, sequence, iterator, condition, or action.

```
- CLEANING_ROUTINE (selector)
  - IS_ROOM_CLEAN (condition)
  - CLEAN_ROOM (sequence)
    - NAV_ROOM (selector)
      - CHECK_LOCATION (condition)
```

\[
\text{\$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz}
\]
Here's how the tree would look if the only task was to vacuum the living room and we omit the battery checking subtree:

We interpret this tree as follows. The top level node (CLEANING_ROUTINE) is a selector so if the condition node IS_ROOM_CLEAN returns SUCCESS, then we are done. Otherwise we move to selector's next child task, CLEAN_ROOM.

The CLEAN_ROOM task is a sequence whose first sub-task is NAV_ROOM which in turn is a selector. The first sub-task in the NAV_ROOM selector is the condition CHECK_LOCATION. If this check returns SUCCESS, then NAV_ROOM also returns SUCCESS and the CLEAN_ROOM sequence can move to the next behavior in its sequence which is the TASK_LIST iterator. If the CHECK_LOCATION task returns FAILURE, then we execute the MOVE_BASE behavior. This continues until CHECK_LOCATION returns SUCCESS.

Once we are at the target room, the TASK_LIST iterator begins. First we check that we are still at the correct location, then we execute each task in the iterator and update the task list.

To make our script more readable, the simulated cleaning tasks can be found in the file cleaning_tasks_tree.py under the rlx2_tasks/src folder. We then import this file at the top of the clean_house_tree.py script. Let's look at the definition of one of these simulated tasks:

```python
class VacuumFloor(Task):
    def __init__(self, name, room, timer, *args):
        super(VacuumFloor, self).__init__(self, name, *args)
        self.name = name
        self.room = room
        self.counter = timer
        self.finished = False
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
        self.cmd_vel_msg = Twist()
        self.cmd_vel_msg.linear.x = 0.05

    def run(self):
        if self.finished:
return TaskStatus.SUCCESS
else:
    rospy.loginfo('Vacuuming the floor in the ' + str(self.room))

    while self.counter > 0:
        self.cmd_vel_pub.publish(self.cmd_vel_msg)
        self.cmd_vel_msg.linear.x *= -1
        rospy.loginfo(self.counter)
        self.counter -= 1
        rospy.sleep(1)
    return TaskStatus.RUNNING

self.finished = True
self.cmd_vel_pub.publish(Twist())
message = "Finished vacuuming the " + str(self.room) + "!
rospy.loginfo(message)

The VacuumFloor class extends the basic Task class. Since we want to move the robot back and forth in a simulated vacuuming motion, we create a ROS publisher to send Twist message to the cmd_vel topic. We then override the run() function which creates the desired motion. Since the run() function is visited on every pass through the behavior tree, we return a status of RUNNING until the motion is complete at which time we return a status of SUCCESS.

The clean_house_tree.py script is similar to the patrol_tree.py program we have already described in detail earlier. Let's therefore focus only on the key differences.

class BlackBoard():
    def __init__(self):
        # A list to store rooms and tasks
        self.task_list = list()

        # The robot's current position on the map
        self.robot_position = Point()

Recall that some behavior trees use an object called the global "black board" for tracking certain properties of the tree and the world. In Python, the black board can be a simple class with a number of variables to hold the data. At the top of the clean_house_tree.py script we define the BlackBoard() class shown above with a list variable to store the task list and a ROS Point variable to track the robot's current coordinates on the map.

black_board = BlackBoard()

# Create a task list mapping rooms to tasks
black_board.task_list = OrderedDict(
    ("living_room", [Vacuum(room="living_room", timer=5)]),
    ("kitchen", [Mop(room="kitchen", timer=7)]),
    ("bathroom", [Scrub(room="bathroom", timer=9), Mop(room="bathroom", timer=5)]),
    ("hallway", [Vacuum(room="hallway", timer=5)])
)

Next we create an instance of the BlackBoard class and create an ordered list of cleaning tasks using the task definitions from the file clean_house_tasks_tree.py in the src/rbx2_tasks subdirectory. This task list will be convenient for iterating through all the tasks assigned to the robot. It also means that we can add or remove tasks by simply editing the list here at the top of the script.

The heart of the script involves creating the desired behavior tree from this task list. Here is the relevant block in its entirety.

```
for room in black_board.task_list.keys():
    # Convert the room name to upper case for consistency
    ROOM = room.upper()
```
As you can see, the behavior tree is built by looping over all the tasks in the task list stored on the black board. The inline comments should make clear how we build a subtree for each room and its tasks. We then add each subtree to the overall CLEAN_HOUSE task.
Parallel tasks

Some times we want the robot to work on two or more tasks simultaneously. The pi_trees library includes the Parallel task type to handle these situations. There are two flavors of Parallel task. The ParallelAll type returns SUCCESS if all the simultaneously running tasks succeed. The ParallelOne type returns SUCCESS as soon as any one of the tasks succeeds. The sample script called parallel_tree.py in the rlx2_tasks/nodes directory illustrates the ParallelAll task type. In this script, the first task prints the message "Take me to your leader" one word at a time. The second task counts to 10. Try out the script now:

```
$ rosrun rlx2_tasks parallel_tree.py
```

You should see the following output:

```
Behavior Tree Structure
   --> PRINT_AND_COUNT
       --> PRINT_MESSAGE
       --> COUNT_TO_10
Take me to your leader! 1 2 3 4 5 6 7 8 9 10
```

Notice how both the message task and the counting task run to completion before the script exits but the output alternates between the two tasks since they are running in parallel.

Let's take a look at the core part of the code:

```python
class ParallelExample():
    def __init__(self):
        # The root node
        BEHAVE = Sequence("behave")

        # The message to print
        message = "Take me to your leader!"

        # How high the counting task should count
        n_count = 10

        # Create a PrintMessage() task as defined later in the script
        PRINT_MESSAGE = PrintMessage("PRINT_MESSAGE", message)

        # Create a Count() task, also defined later in the script
        COUNT_TO_10 = Count("COUNT_TO_10", n_count)

        # Initialize the ParallelAll task
        PARALLEL_DEMO = ParallelAll("PRINT_AND_COUNT")

        # Add the two subtasks to the Parallel task
        PARALLEL_DEMO.add_child(PRINT_MESSAGE)
        PARALLEL_DEMO.add_child(COUNT_TO_10)

        # Add the Parallel task to the root task
        BEHAVE.add_child(PARALLEL_DEMO)

        # Display the behavior tree
        print "Behavior Tree Structure"
        print_tree(BEHAVE)

        # Initialize the overall status
        status = None

        # Run the tree
        while not status == TaskStatus.SUCCESS:
            status = BEHAVE.run()
            time.sleep(0.1)
```
The construction of the behavior tree shown above should be fairly self explanatory from the inline comments. Note that the `PrintMessage()` and `Count()` tasks are defined later in the script and are fairly straightforward so we will not display them here.

If you modify the `parallel_tree.py` script so that the `ParallelAll` task on line 19 above is replaced with a `ParallelOne` task instead, the output should look like this:

Behavior Tree Structure
--> PRINT_AND_COUNT
    --> PRINT_MESSAGE
    --> COUNT_TO_10
Take 1 me 2 to 3 your 4 leader!

Now the script exits as soon as one of the tasks finishes. In this case, the `PRINT_MESSAGE` task completes before the `COUNT_TO_10` task so we do not see the numbers 5-10.

**Adding and removing tasks**

One of the key features of behavior trees is the ability to add or remove behaviors in a modular fashion. For example, suppose we want to add a "dish washing" task to the house cleaning robot when it is in the kitchen. All we need to do is define this new task and add it to our task list at the top of our script and we are done. There is no need to worry about how this new task interacts with other tasks in the behavior tree. It simply becomes another child task in the `TASK_LIST` iterator for the kitchen. Conversely, suppose your robot does not have a docking station and you want to eliminate the recharging task from the behavior tree but you still want to be notified when the battery is low. Then we can simply comment out the line that adds the `RECHARGE` behavior to the `STAY_HEALTHY` task, then add a new task that sends us an email or cries for help.

The addition or removal of nodes or even whole branches of the behavior tree can even be done dynamically at run time. The sample script `add_remove_tree.py` demonstrates the concept. The script is identical to the `parallel_tree.py` script described in the previous section except that at the end of alternate cycles through the tree, we remove the counting task so that only the words are displayed. On the next cycle, we add back the counting task. Here is the key block of code that does the work:

```python
remove = True
while True:
    status = BEHAVE.run()
    time.sleep(0.1)
    if status == TaskStatus.SUCCESS:
        BEHAVE.reset()
        if remove:
            PARALLEL_DEMO.remove_child(COUNT_TO_10)
        else:
            PARALLEL_DEMO.add_child(COUNT_TO_10)
        remove = not remove

Try out the script with the command:

$ rosrun rbx2_tasks add_remove_tree.py

The output should start out the same as the `parallel_tree.py` script except that on successive loops through the script, the counting task will be omitted then reappear, and so on.
It's not hard to imagine a behavior tree where nodes or branches are added or pruned based on a robot's experience to better adapt to the conditions on hand. For example, one could add or remove entire branches of the tree depending on which room the robot is in or switch its behavior from cleaning to patrolling by simply snipping one branch of the tree and adding the other.