WORKSHOP: State Machine Prototyping

1 Introduction

This tutorial is split into two main topics: the SMACH framework and the SCXML State Machine framework. It's better to have basic notions with ROS and Ubuntu to fully understand the tutorial.

All the commands are given for ROS Kinetic and Ubuntu 16.04

Commands and Python code are highlighted in `grey boxes`.

Commands are executed in a terminal:

- Open a new terminal → use the shortcut *ctrl+alt+t*.
- Open a new tab inside an existing terminal → use the shortcut *ctrl+shift+t.*

During the SMACH tutorial you will learn how to:

- Define a state that you can use in SMACH
- Create a Hierarchical State machine using SMACH framework and run it.
- Setup a Concurrence container
- Use the introspection and smach_viewer

During the SCXML State Machine (SSM) tutorial you will learn how to:
- Convert a SMACH state into SSM state
- Use Qt SCXML editor to create a SCXML file usable with the SSM framework
- Run a SCXML file using the ssm_plugin

Before starting:
This tutorial is from https://github.com/ipa-led/state_machine_tutorial.git
Be sure that you have the last version.

2 SMACH

SMACH is a ROS compatible State Machine Python framework to ease the conception of Hierarchical Finite State Machine.

Online documentation can be found here: http://wiki.ros.org/smach
Source code is here: https://github.com/ros/executive_smach

To install SMACH and SMACH Viewer:

```
$ sudo apt-get install ros-kinetic-smach
$ sudo apt-get install ros-kinetic-smach-viewer
```

2.1 SMACH State

The states are the core of a state machine. This is where the code is executed and the basic bloc that you assemble in a State Machine.
Let's breakdown a very simple state. This state just waits for 1 second, and, then returns an outcome.
WARNING:
 As the code will be in Python, be sure that your editor "indent setting" is set to **SPACE**.
Open the file **BasicStates.py** from the package *state_machine_tutorial* in the src/smach_tutorial/

You can use the file explorer or from the terminal:

```
$ roscd state_machine_tutorial
$ cd src/smach_tutorial/
$ gedit BasicState.py
```

This is very basic state:

```python
#!/usr/bin/env python

import smach
import rospy
import smach_ros


##-------------------------------------------------------------------------------
##Exercise 0

class EmptyState(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=["continue"], input_keys=[], output_keys=[],
io_keys=[])

    def execute(self, ud):
        rospy.sleep(1)
        return "continue"
```

Line by Line explanation:

```
#!/usr/bin/env python

import smach
import rospy
import smach_ros
```

These line setup the python file and import the libraries we will need:
- smach - The SMACH framework. It contains the core libraries like State, StateMachine or Userdata
- rospy – To get access to the ros API, like the subscribers, publishers, rostimes, etc…
- smach_ros – The SMACH ROS oriented templates or API (not used here but usually useful)

```
class EmptyState(smach.State):
```

The class we will create. It's called EmptyState and inherits from "smach.State" to have access to the core interface from SMACH framework, including outcomes for transitioning, consistency checking, and interface for the userdata ….

```
    def __init__(self):
        smach.State.__init__(self, outcomes=["continue"], input_keys=[], output_keys=[],
io_keys=[])
```

We initialize the class and the interfaces.
- outcomes - the possible "string" return from the state execute function
- input_keys - read access keys inside the userdata dictionary
- output_keys - write access keys inside the userdata dictionary
- io_keys – read and write access keys inside the userdata dictionary

Here we have just one possible outcome: "continue" and we don't need any access to the userdata.

```
    def execute(self, ud):
        rospy.sleep(1)
        return "continue"
```

We have to implemented the execute function of the smach.State. This is what is actually run when the state machine enter the state (execute this state).
"ud" refer to the userdata structure. It can "shortcut" the access to certain data:
For example, to access the data "**msg**" inside this userdata, both are valid:
- ud["msg"]
- ud.msg

EXERCISES:
Exercise 0:
You can run a simple state machine that will just run a stack of three EmptyState instances.
Open smach_viewer:

```
$ rosrun smach_viewer smach_viewer.py
```
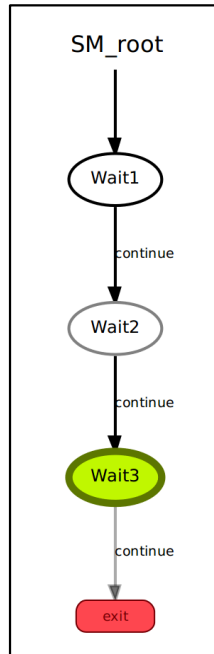
Then, in another terminal, launch the already created state machine:

```
$ roslaunch state_machine_tutorial exo_basicstates.launch exercise:=0
```

You can see the log from the state machine in this terminal.
From smach_viewer:

Exercise 1:
Create a **WaitState** state that uses the userdata **"sleep_time"** to set the sleep time.
- Outcome : "**continue**"
- Userdata : "**sleep_time**"

Try out:

```
$ roslaunch state_machine_tutorial exo_basicstates.launch exercise:=1
```

You should see the same state machine than the exercise 0.

Exercise 2:
Create a **MessageReader** state that prints the string inside the userdata **"msg"** then empty it (set is value to "") then return the outcome.
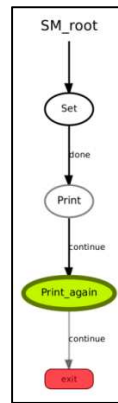- Outcome : "**continue**"
- Userdata : " **msg** "

Note:
    To print on the roslog system you can use *rospy.loginfo("This is a general info message")*

Try out:

```
$ roslaunch state_machine_tutorial exo_basicstates.launch exercise:=2
```

You should see on smach_viewer:

Exercise 3:

Extend the **MessageReader2** state, based on the previous state with:

- If the string inside the userdata "**msg**" is not empty, print it
  - If the userdata "**reset**" is True, empty the message else don't do it.
  - Then return with the outcome "**continue**"
- If the string inside the userdata "**msg**" is empty, print an error message and return with the outcome "**empty**"

- Outcome : "**continue**", "**empty**"
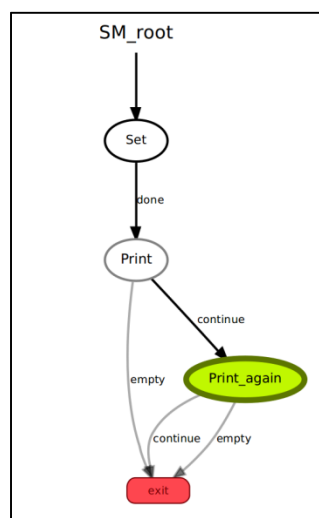- Userdata : "**msg**", "**reset**"

Note:

To print on the roslog system an error, you can use *rospy.logerr ("THIS IS AN ERROR !!!")*

To print on the roslog system a warning, you can use *rospy.logwarn ("This is a WARING !!")*

Try out:

```
$ roslaunch state_machine_tutorial exo_basicstates.launch exercise:=3
```

You should see on smach_viewer:

Solutions are available inside the solution folder: BasicStates.py

2.2 SMACH State Machine

State machines are containers in which you can organize the state execution order list. They have the same interfaces than the state. For a state machine, executing a state or state machine is similar, so, it's possible to nest state machines into other state machine.

Open the file **BasicStateMachine.py** from the package *state_machine_tutorial* in the src/smach_tutorial/

We have two simple states:
- Set : Set a string from his parameter inside the userdata "msg"
- MessageReader : print the message inside the userdata "msg"

Then we define the state machine inside a function:

```python
def SetPrintStateMachine():
    SetPrint_sm = smach.StateMachine(outcomes=["exit"])
    SetPrint_sm.userdata.msg = "Message in user data"

    with SetPrint_sm:
        SetPrint_sm.add('Set', Set("Hello World"), transitions={"done": 'Print'},
                                            remapping={"Set_msg_in":"msg",
                                            "Set_msg_out":"msg"})
        SetPrint_sm.add('Print', MessageReader(), transitions={"done" : 'exit'},
                                     remapping={"msg":"msg"})

return SetPrint_sm
```

(You don't have to make a function to create a state machine, but for the exercise, it's easier to do so).

Line by Line explanation:

```python
    SetPrint_sm = smach.StateMachine(outcomes=["exit"])
```

We don't re-implement the StateMachine interfaces; we use them as a new instance. For this example, our StateMachine only has 1 possible outcome called "**exit**". Has we don't transfer userdata from upper level (there is no upper level for this example), we don't use the input_keys nor the output_keys (note that there is no io_keys interfaces for state_machine).

```python
    SetPrint_sm.userdata.msg = "Message in user data"
```

We initialize the userdata.msg to a value. We only need to initialise the data:
- The first state to use this data, use it has an input (input_keys or io_keys).
- And the data is not initialize by a parent state (in case of a nested state machine)

```python
    with SetPrint_sm:
```

We open the StateMachine containers so we can add state in it. When we return from this block, it will perform a consistency checking and return an error if something is wrong.

```
SetPrint_sm.add('Set', Set("Hello World"), transitions={"done": 'Print'},
                                remapping={"Set_msg_in":"msg",
                                "Set_msg_out":"msg"})
```

We add the first state, it's the initial state of the state machine if we don't specify it otherwise using: *set_initial_state("initial_state_label")*.

The parameters of the ***add(label, state,* transitions**=None, **remapping**=None) function are:

- Label - a unique string to reference the state in the state machine (like a key in a dictionary). Here it's 'Set'
- State – an instance of the state class. Here we create a new instance with "Hello world" as a parameter.
- Transitions – a dictionary mapping state outcomes to other state labels or this state machine outcome. Here we map the Set outcome "**done**" to the State label 'Print'.
- Remapping – a dictionary mapping State userdata (here "**Set_msg_in**" and "**Set_msg_out**") to the state machine userdata (here "**msg**").

```
SetPrint_sm.add('Print', MessageReader(), transitions={"done" : 'exit'},
                            remapping={"msg":"msg"})
```

We add the second state before closing the StateMachine and performing the consistency checking. Here we map the outcome "**done**" on the State Machine outcome "**exit**"

To actually start the state machine, you can call:

```
SetPrint_sm.execute()
```

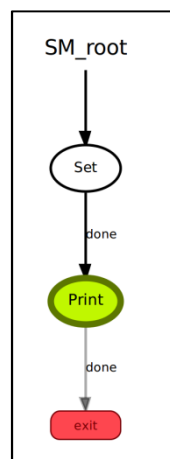But here, all the execution part is already done inside other scripts.

EXERCISES:
Exercise 0:
You can run this simple state machine by calling:

```
$ roslaunch state_machine_tutorial exo_basic_statemachine.launch
exercise:=0
```

Here is the result that can be seen on the smach_viewer.
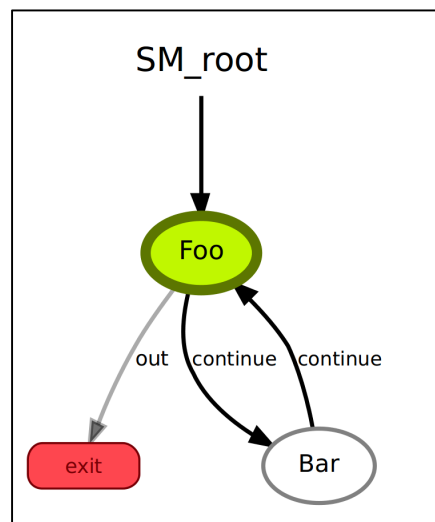
Exercise 1:

There are two states:

- Foo – A state that increase his inside counter by one every time it's called and if the counter is 3 or higher return with "**out**", otherwise it return "**continue**"
- Bar – A state that wait 2 seconds, before returning "**continue**"

Create a StateMachine instance named "FooBat_sm" with the outcome "**exit**" then that loop between Foo and Bar until Foo return "**out**".
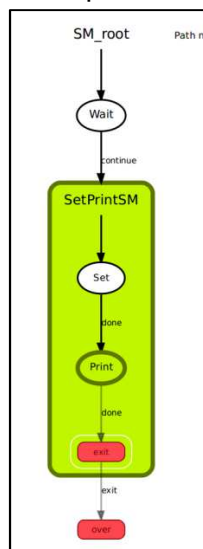
Try out:

```
$ roslaunch state_machine_tutorial exo_basic_statemachine.launch
exercise:=1
```

You should see on smach_viewer:

Exercise 2:

Create a nested state machine, for example like this:

To try out:

```
$ roslaunch state_machine_tutorial exo_basic_statemachine.launch
exercise:=2
```

Typical methodology to use efficiently State Machine:
- Define your application as a list of simple tasks
- Create the states corresponding to the define tasks
- Test your states in small, simple state machine
- Assemble your state machines to create your application

2.3 SMACH Introspection and execution

The introspection, which is used by smach_viewer is actually part of the smach_ros. In order to publish the introspection, you have to create an introspection server and make it inspect your state machine.

Open the file **Introspection.*py*** from the package *state_machine_tutorial* in the src/smach_tutorial/

The state machine is the same as in the Basic State Machine exercise 0.

```
SimpleSM = SetPrintStateMachine()


introspection_server = smach_ros.IntrospectionServer('SM', SimpleSM, '/SM_root')
introspection_server.start()


outcome = SimpleSM.execute()
rospy.loginfo("Result : " + outcome)
introspection_server.stop()
```

Line by Line explanation:

```
SimpleSM = SetPrintStateMachine()
```

We create a simple State Machine instance (see basic state machine exercise 0)

```
introspection_server = smach_ros.IntrospectionServer('SM', SimpleSM, '/SM_root')
```

This line generates an IntrospectionServer. The parameters are:
- Server Name - 'SM' this is a namespace for the introspection topics.
- State – SimpleSM the state machine instance.
- Path – '/SM_root' the path for the introspection root. (can be used to ease the comprehension of very complex state machine)

```
introspection_server.start()
```

This start the publication on the different introspection topics (if not customized, the frequency is 0.5Hz).

```
outcome = SimpleSM.execute()
rospy.loginfo("Result : " + outcome)
```

We start the execution of the State Machine and will print the result.

```
introspection_server.stop()
```

We stop the introspection server.

To run the script, call the command line:

```
$ rosrun state_machine_tutorial introspection.py
```

EXERCISES:

Exercise 1:

Complete the main1 function with an introspection server and execute the FooBarStateMachine (created before).

Change the function call inside the __main__ from main() to main1()

Run the script by calling thus command line:

```
$ rosrun state_machine_tutorial introspection.py
```
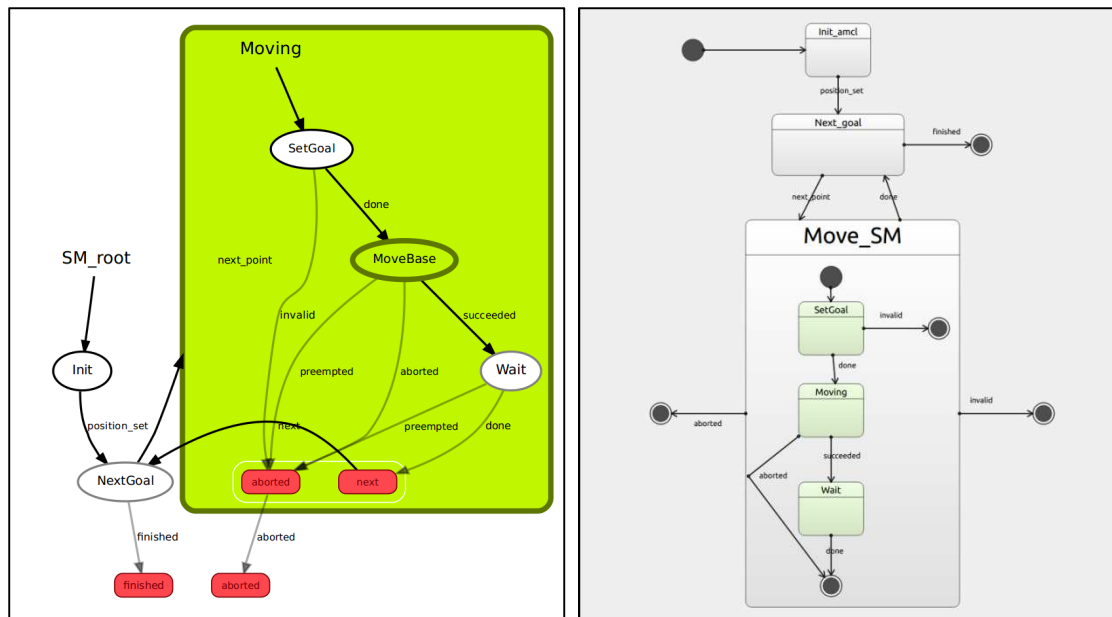
3 SCXML State Machine

The SCXML State Machine (SSM) framework is an SMACH overhaul to improve the reusability, the flexibility and accessibility of SMACH. It converts SCXML files into SMACH compatible state machine and provides some ROS interfaces to access some functions (like pre-emption) or start/pause.

SSM is part of metapackage. You can retrieve the last version here: https://github.com/ipa-led/airbus_coop.git
Note: The kinetic version will be release soon.

The objective of this tutorial is to create a state machine to make a Turtlebot run a full simulation.

3.1 SCXML Files

The SCXML format is an XML-based mark-up language use to create state machine. It can describe complex finite state machine.
The standard description can be found here: https://www.w3.org/TR/scxml/

As there is no (not yet at least) dedicated SSM editor, we will use the QtCreator plugin SCXML Editor.

QtCreator should already be installed on the USB-Stick provide. You can also download it from here: https://www.qt.io/ (the plugin is only provided since QtCreator 4.2).
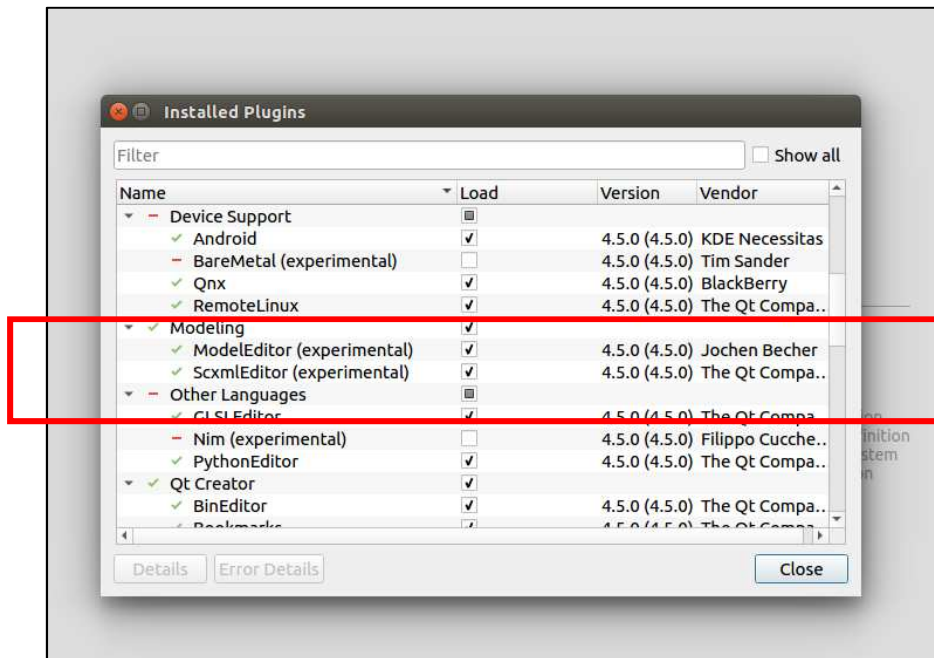**To start it:**

```
$ qtcreator-ros
```

As the plugin is not enabled by default, you should first enable it. To do so in QtCreator select:
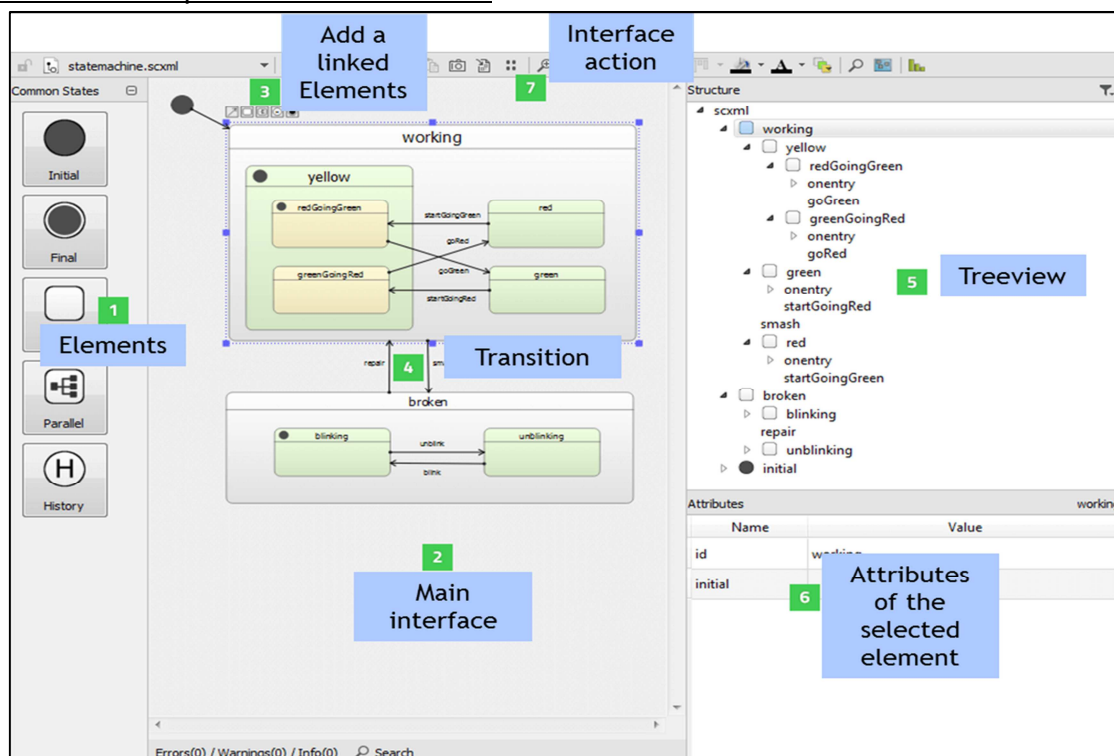
Help > **About Plugins** > **Modeling** > **ScxmlEditor**

You have to restart QtCreator.

Tips: To start a new SCXML file: **File (top tool bar)** > **New File or Project** > **Files and Classes** > **Modeling** > **State Chart.**

Here is a description of the interface:

To add an element:
- Drag and drop from the left side to the main interface.
- To nest, drag inside another element.
- To create a transition **(arrow symbol)**, select the element from which you want to transition then **right click** on the element you want to transition to.

EXERCISE:

Open the SCXML File **Moving.scxml** from *state_machine_tutorial* in the resources/scxml/ folder using QtCreator.

Design this simple state machine. It will be the core function in order to make the Turtlebot move.
- The first state (SetGoal) will convert the goal from a list into a Move_base compatible goal message. It can return "**done**" or "**invalid**"
- The second state (Moving) will send the goal to Move_base action server and wait for an answer. It can return "**aborted**" or "**succeeded**".
- The third (Wait) is a simple wait state which can only return "**done**".

Note:
- The transitions are named after the states outcome.
- The initial state have to added but the transition name isn't needed
- Final states are just target for transitions. They don't have to be renamed.
- **All outcomes have to be linked to something.**
- States IDs (SetGoal, Moving and Wait) have to be unique but can be set as you wish (for example you can change the ID Moving by anything you wish).
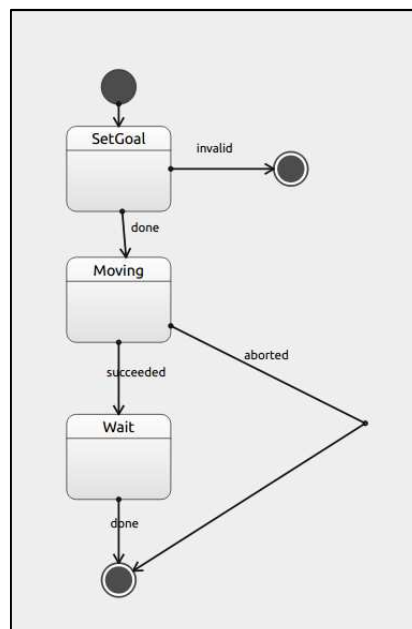
Save you SCXML.

Now, we have to link this SCXML to some python code.

Open the **states.py** from *state_machine_tutorial* in the src/scxml/ folder using your python code editor.

You can see that most of the states needed are already created. You just have to complete the Wait state. This state simply sleep for the duration set inside the userdata "**sleep_time**" then return "**done**" (outcome: "**done**", userdata : "**sleep_time**").
Don't forget:
- **To use SSM it's execution not execute.**
- Use ssm_states.SSMState to have access to the interfaces from SSM.
- Access userdata using only io_keys.
- As the data set inside the SCXML will be string. You need to convert them into python structure (like floats or lists) by using (replace ud.data by any userdata key you need):

```
if(isinstance(ud.data,basestring)): ##Check if the userdata is a string or not
    ud.data = ast.literal_eval(ud.data) #convert the string into a python structure
```

Now that you have edited the state, you have to add it into the Register file in order to make it findable by the SSM interpreter.
To do that; open **register.xml** from *state_machine_tutorial* in the resources/scxml/ folder.

Add your wait state inside the register.
- name - The reference to call this state inside the SCXML (Can be anything but, must be unique inside the register).
- pkg - The ROS package where the classes can be found. (Here "state_machine_tutorial"
- module – The Python path to the file in which the class is ("ssm_tutorial.states").
- class – The name of the Python class.

You can also see inside the register that the states SetGoal and MoveBase_ac are already registered.
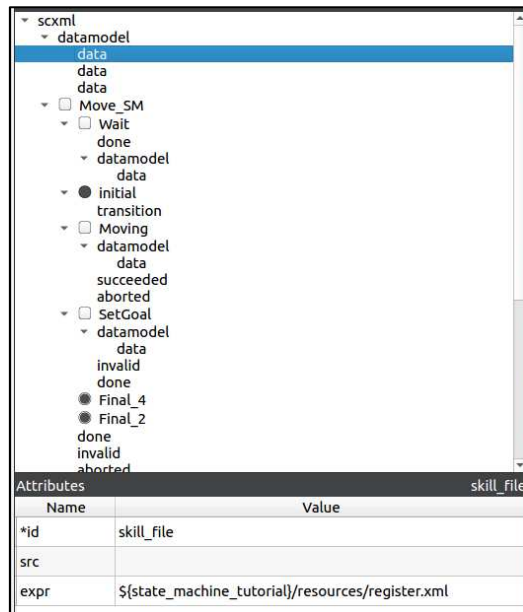
It's now time to link the SCXML file to the register file and the state to the python class.

Open again the **Moving.scxml** file using QtCreator.
Add to the root scxml a datamodel by right-clicking on the treeview and selecting datamodel.
Add a data inside this datamodel by right-clicking the new datamodel and selecting data.
Select the new data and edit its attribute:

- Id=**skill_file** - This is meant for the interpreter to know that we are linking the register file.
- expr=${state_machine_tutorial}/resources/register.xml – We are giving the path to the register.xml file we previously set. The ${state_machine_tutorial} mean that we are referring to the state_machine_tutorial ROS packages.

Using this, the interpreter will find the register.xml file and will be able to find the Python Classes we set up before.

We still have to tell inside the SCXML which state (block) corresponding to which Python Class.

Add a datamodel to the Wait block by right-clicking it in the tree-view and selecting datamodel then add a data.
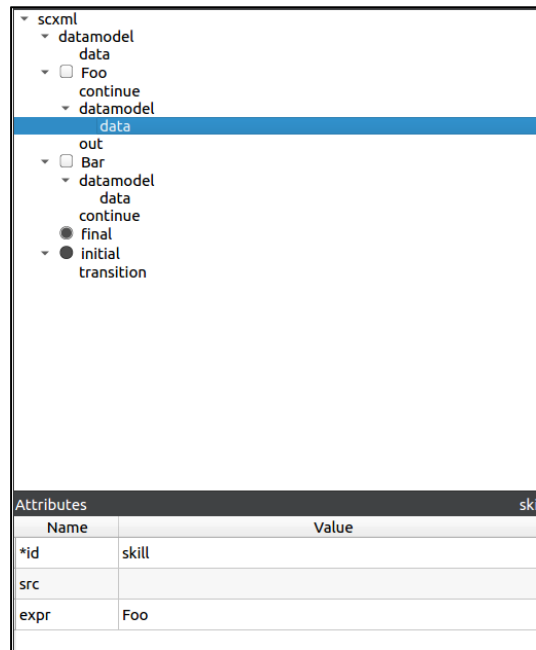
Here we will specify that we want to use the skill "Wait".

- Id="skill" – This is use to tell the interpreter that the value is a skill **name** referenced in the register xml.
- Expr="Wait" (or the name you used inside the register) – This is used to retrieve the skill information inside the register file.

Add the same information for the two other states:

- SetGoal related skill is SetGoal in the register.
- Moving related skill is Movebase_ac in the register.

We need now to define our userdata. There are two userdata used in this state machine:

- "**goal**" – used by SetGoal, it's a list containing the following information [x,y,theta]. Where x and y are the position in the map and theta the orientation in radian. These are the goal information. One possible goal can be : [0.592, -0.553,0.0]
- "**sleep_time**" – used by Wait. It's the time in second to be waited after reaching a goal.

There are two possible way to add this:

- Inside the scxml's datamodel. This will be set at the beginning of the execution of the whole state machine
- Inside the related state datamodel. Here, the data will be set everytime the state is executed overriding the existing previous value (if it existed).

Here, we want to be sure that the sleep time is set to 2.0 seconds for the wait. Add a data inside the Wait datamodel.
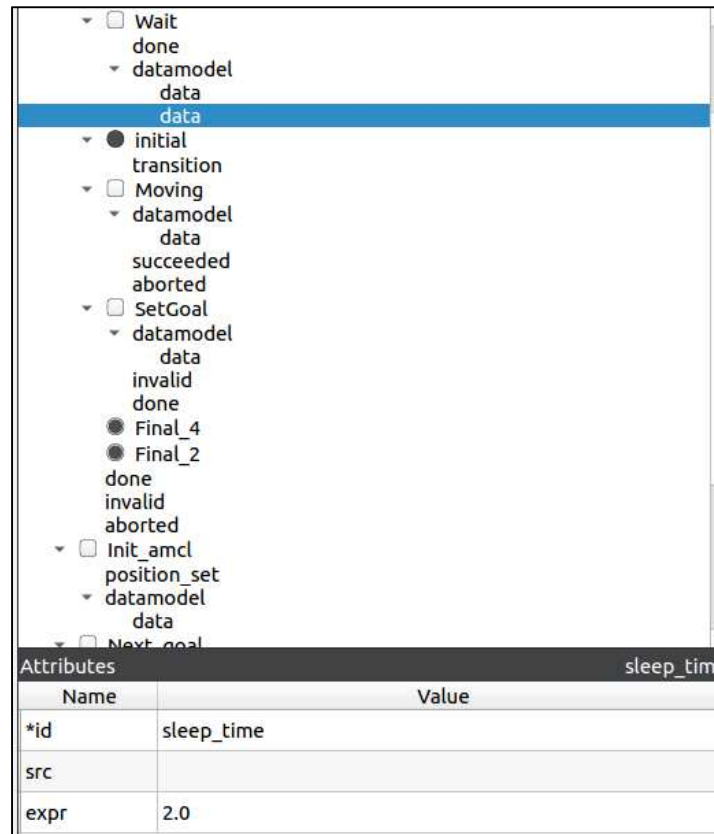
- Id=sleep_time – We want to set the value of the "**sleep_time**" userdata.
- Expr=2.0 – The value is set to "**2.0**".

For the "goal" userdata, we doesn't want to it be set every time the state is executed (the final objective is to use a list of goal). So we set up the data inside the scxml data model.

- Id=goal – We want to set the value of the "**goal**" userdata.
- Expr= – The value use for this userdata.

The first StateMachine is now set. Open the Turtlebot simulation:

```
$ roslaunch state_machine_tutorial sm_turtlebot3_navigation_gazebo.launch
```

Localize the robot inside the map (use either rViz or some other tools).

When the robot is localized you can test the state machine with the GUI:

```
$ roslaunch airbus_ssm_plugin ssm_plugin_standalone.launch
```
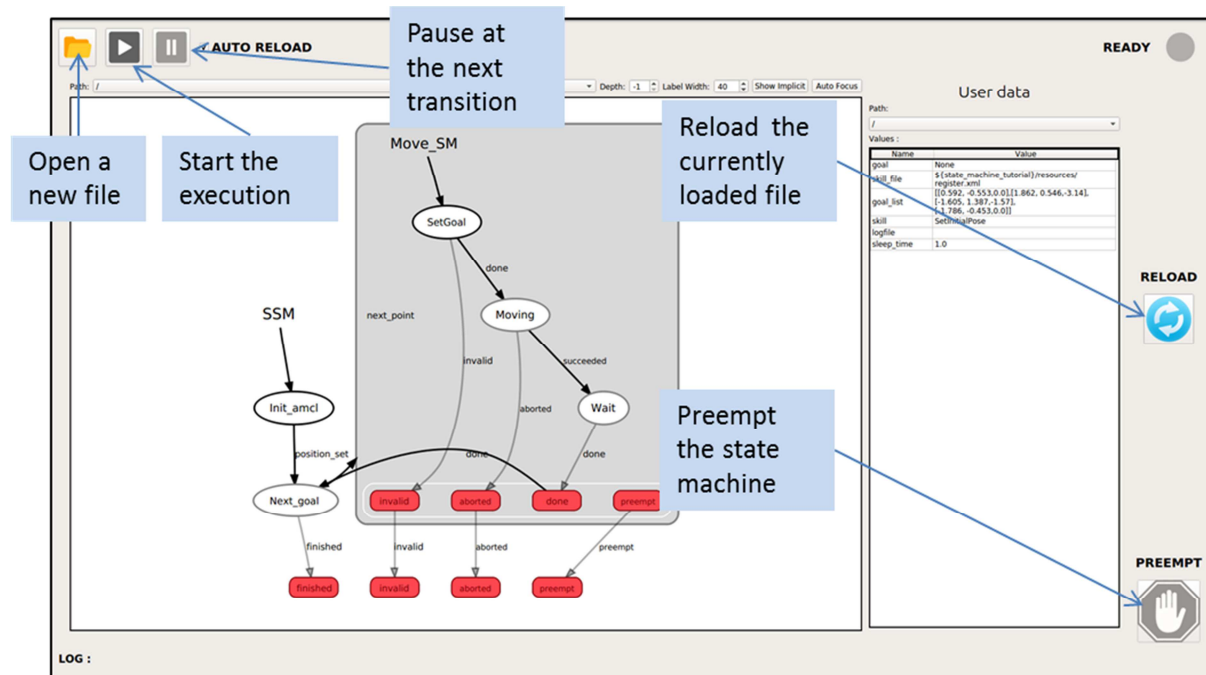
Use the GUI to open your SCXML file. If the state machine doesn't load, please check the error message from the terminal.

You can run the state machine by pressing the play button.
Change the goal destination, save the SCXML and reload it. Try again.

3.2 Extending the State Machine

Now that our first state machine is working, we can extend it to make it run a full trajectory.

To extend it, we will need two more states:
- NextGoal - Will find the next goal inside a list of goal (so a list of list) from the userdata "**goal_list**". It can return "**next_goal**" and "**finished**".
- Init_amcl - In order to localize the robot at the beginning. (So we don't have to it manually).

As all the states are already created (inside the states.py file) and linked inside the register.xml (you can look at them if you are interested) we just have to setup the rest of the state machine inside the SCXML.

Let's start by nesting our already existing state machine. Drag and drop a new state inside the main interface select your whole state machine (minus the newly added state) and drag and drop them inside the new state.
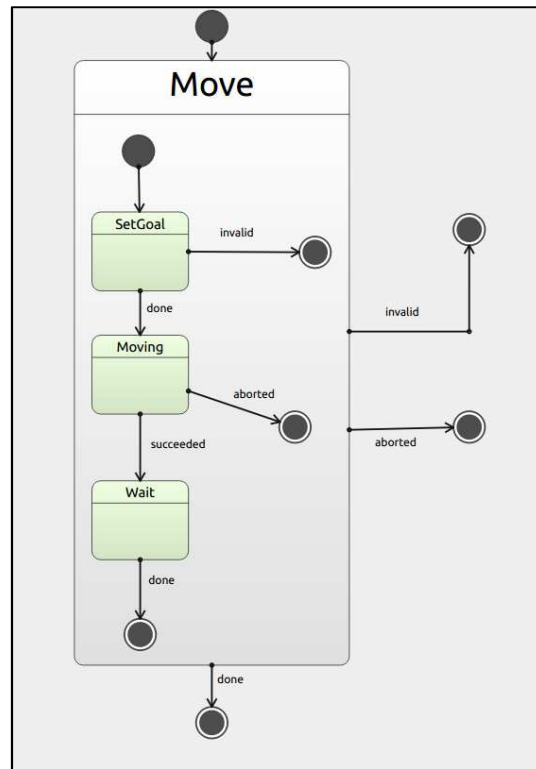
Now plug the outcome of this state. As it is a state machine, the possible outcome of this state are the one coming from the states inside (the one plugged on the final state).

There are 3 outcomes:

- "invalid" : we want to stop the state machine if this occurs. This means we need to create a new transition named "invalid" going from the parent state to a final state.
- "aborted" : Same as "invalid"
- "done" : Will be plugged to the NextGoal State.

Now add the two other states and make them usable by the SSM interpreter using the datamodel:

- SetInitialPose to Init_amcl
- NextGoal to NextGoal
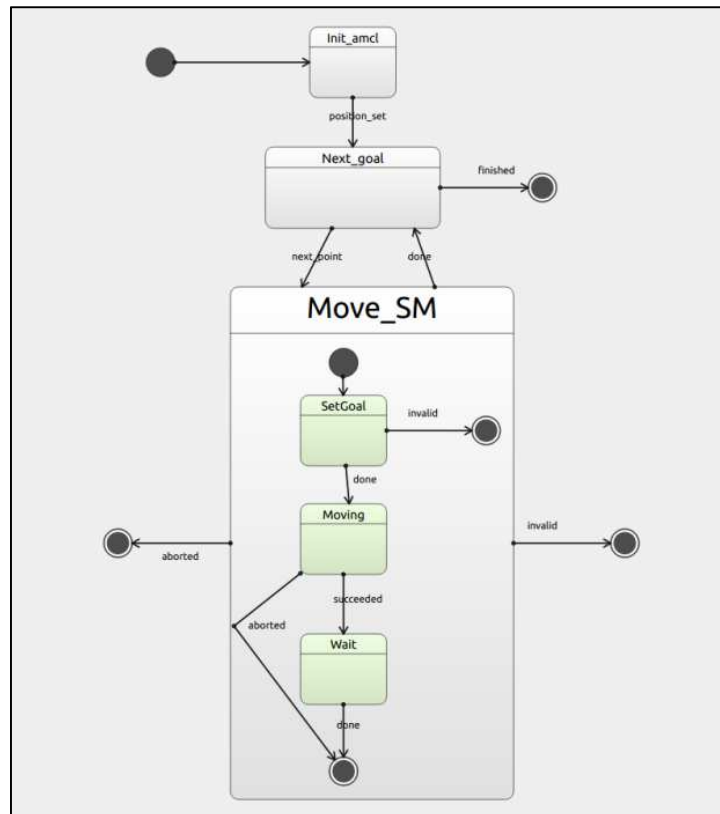
Also change the goal data by goal_list with these values:
[[0.592,-0.553,0.0],[1.862,0.546,-3.14],[-1.605,1.387,-1.57],[-1.786,-0.453,0.0]]

Relaunch the simulation and try the new state machine.

Congratulations, you have finished the tutorial!

4. Advanced Topics

Advanced States:

1. Create a complex state that will wait to receive a message on a define topic **("/message")** of type **"std_msgs/String"** then store the message data in the **"msg".** This state take into account: possible SMACH pre-emption and timeout (time limit define as a parameter for the initialisation).

- Parameter : "**time_out**" (float that define the timeout limit)
- Outcomes : **"timeout", "message_received", "preempted"**
- Userdata : **"msg"**

To try out:

```
$ roslaunch state_machine_tutorial exo_advancedstates.launch exercise:=0
```

Then you can either use Rqt or a Ros command line to publish on the topic.
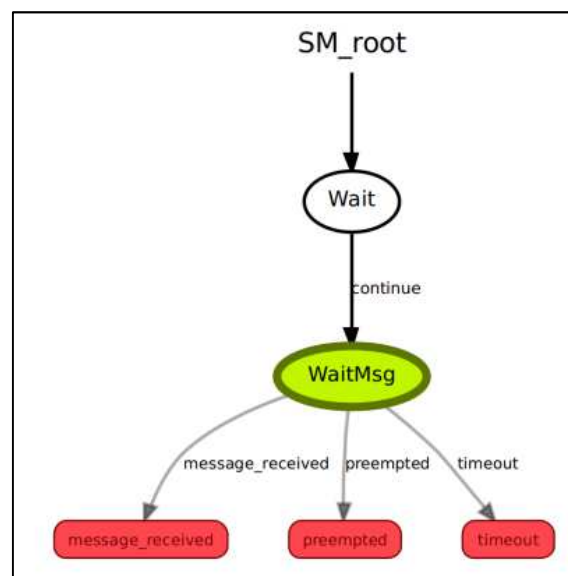
In the solution you also have a possible action client for move base with the same waiting functionalities. It's a good way to be sure that you don't block your Python thread and are still able to stop / cancel the process while executing.

With long blocking API, it may be hard to stop the process.

It also possible to setup an action client using the SMACH template, more details can be found here: http://wiki.ros.org/smach/Tutorials/SimpleActionState

Advanced State Machine (SMACH):
Open the file ***AdvancedStateMachine.py*** from the package *state_machine_tutorial* in the src/smach_tutorial/

The first part is creating a state machine that is able to make a platform move using move_base.
We have 3 states:

- SetGoal – This state takes a list of 3 floats as input and converts it into a MoveBaseGoal.
- MoveBase_ac – This state is an action client for move_base with request the goal and wait for the result. (NOTE: There is no result analyse, so if the action fail, it's still return "succeeded").
- Wait – A simple wait state.

```python
Moving_sm = smach.StateMachine(outcomes=["exit"])
Moving_sm.userdata.goal = [0.592, -0.553,0.0]

with Moving_sm:
    Moving_sm.add('SetGoal', SetGoal(),
                                    transitions={"done" : 'MoveBase',
                                                  "invalid" : "exit"},
                                    remapping={"goal_in"  : "goal",
                                               "goal_out" : "goal"})


    Moving_sm.add('MoveBase', MoveBase_ac(),
                                transitions={"succeeded"  : 'Wait',
                                             "aborted"   : "exit",
                                             "preempted" : "exit"},
                                remapping={"goal"      :"goal"})


    Moving_sm.add('Wait',  Wait(1.0), transitions={"done" : "exit",
                                             "preempted" : "exit"})
```

We map the outcomes and the userdata the same way we add the state inside the state machine before.

It's possible to test this state machine. First we launch the turtlebot3 gazebo simulation.

```
$ export TURTLEBOT3_MODEL=waffle
$ roslaunch state_machine_tutorial sm_turtlebot3_navigation_gazebo.launch
```

If you don't have the turtlebot3 simulation installed:

```
$ sudo apt-get install ros-kinetic-turtlebot3 ros-kinetic-turtlebot3-gazebo
```

Then localize the robot (using either rviz or teleoperation).

When the robot is localized:

```
$ roslaunch state_machine_tutorial exo_advanced_statemachine.launch
exercise:=0
```

This will connect to the turtlebot move_base actions server and send the goal on it then wait for a result and return.
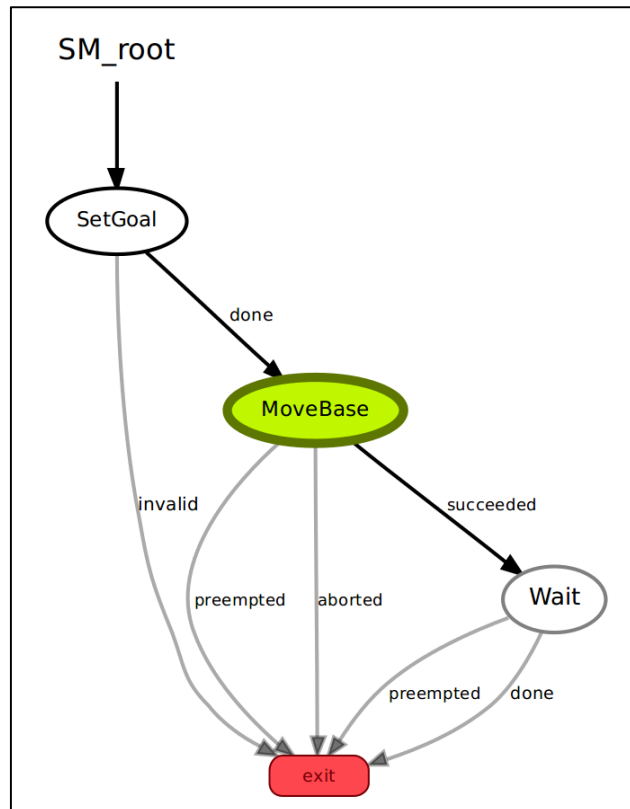
Now that the moving state machine is setup, we can setup a more complex state machine to execute a list of goal.

First we need to convert a bit the previous state machine, in order to make it works as a loop:

```
    Moving_sm =
smach.StateMachine(outcomes=["next","aborted"],input_keys=["goal"])

    with Moving_sm:
        Moving_sm.add('SetGoal', SetGoal(),
                                    transitions={"done" : 'MoveBase',
                                                "invalid" : "aborted"},
                                    remapping={"goal_in"  : "goal",
                                                "goal_out" : "goal"})

        Moving_sm.add('MoveBase', MoveBase_ac(),
                                    transitions={"succeeded"  : 'Wait',
                                                "aborted"    : "aborted",
                                                "preempted" : "aborted"},
                                remapping={"goal"        : "goal"})
        Moving_sm.add('Wait',  Wait(2.0),
                                    transitions={"done"        : "next",
                                                "preempted" : "aborted"})
```

We add a second outcome in case of failure and we map the "**goal**" from the upper state machine.

We also create two new states:
- SetInitialPose – This state set the initial pose of the robot (before the robot move).
- NextGoal – read the list of points and extract the next point. Also check if we have finished the state machine.

Then we define the parent state machine:

```
FullTrajectory_sm = smach.StateMachine(outcomes=["aborted", "finished"])
#the goal list
FullTrajectory_sm.userdata.goal_list = [[0.592, -0.553,0.0],
                                        [1.862, 0.546,-3.14],
                                        [-1.605, 1.387,-1.57],
                                        [-1.786, -0.453,0.0]]

with FullTrajectory_sm:
    FullTrajectory_sm.add('Init', SetInitialPose(), transitions={"position_set":'NextGoal'})

    FullTrajectory_sm.add('NextGoal', NextGoal(), transitions={"next_point":'Moving',
                                                    "finished" : "finished"},
                                        remapping={"goal_list" : "goal_list",
                                                   "next_goal" : "goal"})

    FullTrajectory_sm.add('Moving', MovingSMNested(), transitions={"next" : 'NextGoal',
                                                    "aborted" : "aborted"},
                                        remapping={"goal" :  "goal"})
```
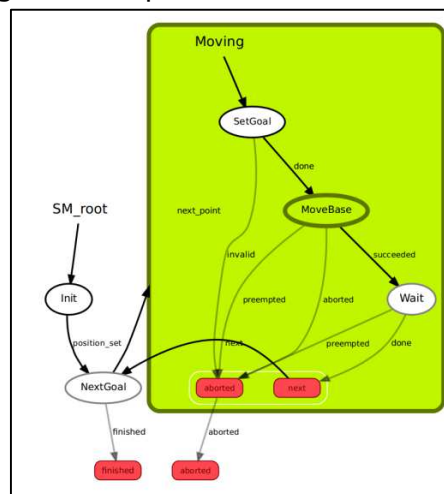
Restart the turtlebot simulation:

```
$ roslaunch state_machine_tutorial sm_turtlebot3_navigation_gazebo.launch
```

And then start the full state machine:

```
$ roslaunch state_machine_tutorial exo_advanced_statemachine.launch
exercise:=1
```

You see the turtlebot going to the 4 points defined in the "**goal_list**" userdata.

This covers the basics of SMACH and some SMACH/ROS interactions.

Advanced State Machine Concurrence (SMACH):

The concurrence is a state container (like a state machine) that executes all his child states simultaneously (in different Python Threads). This means you have to define a mapping of the different child possible outcomes. Every time a child returns an outcome, it will test the outcome map. If one is found, it will return the corresponding outcome and, may, pre-empt the other child if they are still running or wait until all the child are finished.

It's possible to have a custom made testing after each child return. I refer you to this tutorial: http://wiki.ros.org/smach/Tutorials/Concurrence%20container (Part 1.2)

The concurrence has also the same interfaces than a state so it can be use inside other containers (other concurrences or inside a state machine).

WARNING: Python Multithreading is no real multithreading. Don't use blocking API or you will block your entire state machine.

Open the file **Concurrence.py** from the package *state_machine_tutorial* in the src/smach_tutorial/

The state ConcurrenceState will generate a random number between -50 and 50 and return "**positiv**" or "**negativ**".

```
FooBar_cc = smach.Concurrence(outcomes = ["positiv","negativ"],
                               default_outcome = "positiv",
                     outcome_map = {"positiv" : {'Foo':"positiv",'Bar':"positiv"},
                                    "negativ" : {'Foo':"negativ",'Bar':"negativ"}})
with FooBar_cc:
    FooBar_cc.add('Foo',ConcurrenceState())
    FooBar_cc.add('Bar',ConcurrenceState())
```

Line by Line explanation:

```
FooBar_cc = smach.Concurrence(outcomes = ["positiv","negativ"],
                               default_outcome = "positiv",
                     outcome_map = {"positiv" : {'Foo':"positiv",'Bar':"positiv"},
                                    "negativ" : {'Foo':"negativ",'Bar':"negativ"}})
```

As a StateMachine we don't reimplement the concurrence interfaces. We create a new instance of the class.
- We set two outcomes "**positiv**" and "**negativ**".
- The *default_outcome* is set to "**positiv**", this means if will return "**positiv**" if the child outcome combination never met any define in the outcome map.
- The *outcome_map* is the different possible child outcomes combination mapped to a concurrence's outcome. Here we define, if the two states return "**positiv**" then the concurrence return "**positiv**". The same for "**negative**"

```
with FooBar_cc:
    FooBar_cc.add('Foo',ConcurrenceState())
    FooBar_cc.add('Bar',ConcurrenceState())
```
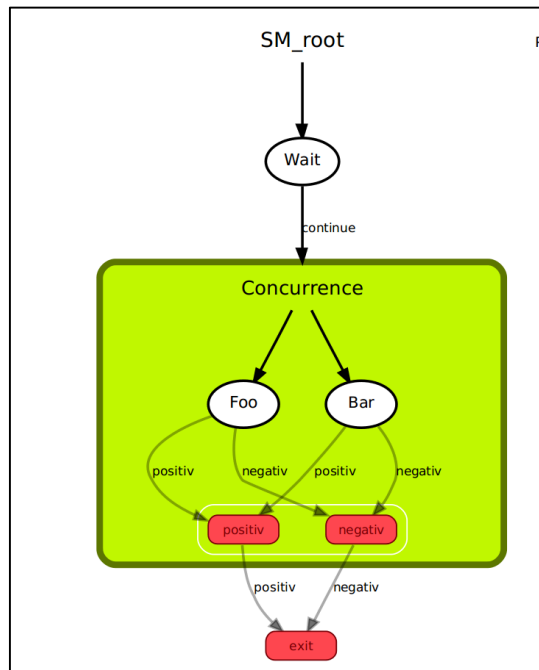
We open the concurrence as a State Machine.
We simply add two instances of the sample state with different labels. Note that we don't define any transition here.

You can try the concurrence by calling:

```
$ roslaunch state_machine_tutorial exo_concurrence.launch exercise:=0
```


The second example is a way to "synchronize" (it's not real time synchronization) two states inside a concurrence.
Pong wait until the userdata "**ping**" is created.
The state Ping create "**ping**" in the userdata dictionary after 3seconds and return.
When ping return, it updates the userdata dictionary so Pong can see it.

You can try it by calling:

```
$ roslaunch state_machine_tutorial exo_concurrence.launch exercise:=1
```