

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 System variables 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 Hierichal 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 that just wait for 1 second and then return.

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:

```
#!/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 inherit from "smach.State" to have access to the core interface from SMACH framework, including outcomes for transitioning, consistency checking, 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 :

```
$ rosrn smach_viewer smach_viewer.py
```

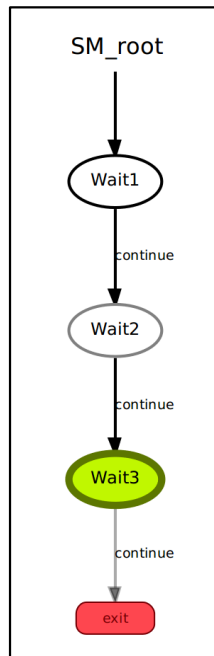
ROS-I Training by ROSIN

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 its value to "") then return the outcome.

- Outcome : **"continue"**
- Userdata : **"msg"**

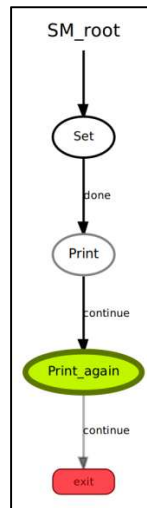
General info :

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 that :

- 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"**

General info :

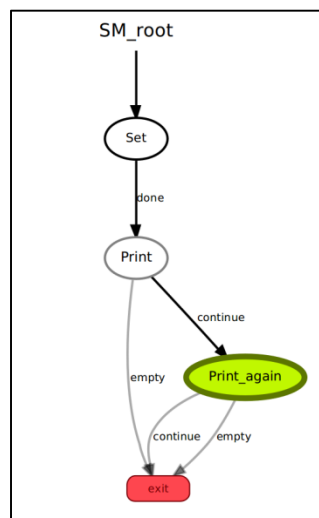
To print on the roslog system an error, you can use *rospy.logerr* ("THIS IS AN ERROR !!!")

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

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

If you feel confident enough you can try to complete the AdvancedStates.py, otherwise copy the advancedstates.py from the solution folder. The code is breakdown inside the solution file.

Exercise Advanced:

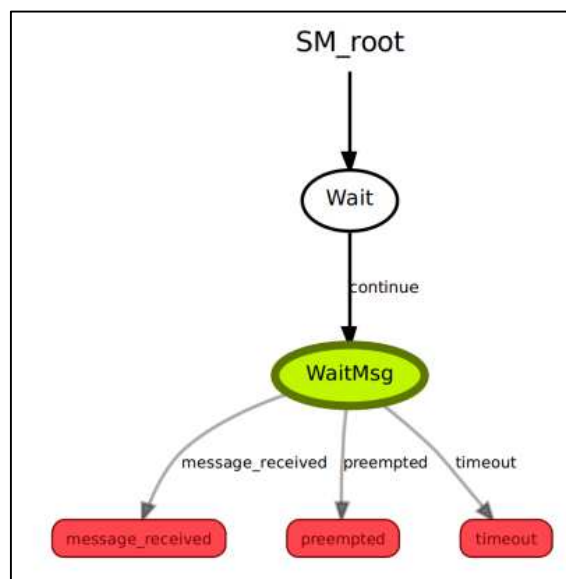
1. Create a complex state that will wait to receive a message on a define topic (**"/message"**) of type **"std_msgs/String"** then store the message data in the **"msg"**. This state take into account: possible SMACH pre-emption and timeout (time limit define as a parameter for the initialisation).

- 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>

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:

```
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:

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

We don't reimplement 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 upperlevel 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).

```
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)

```
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.

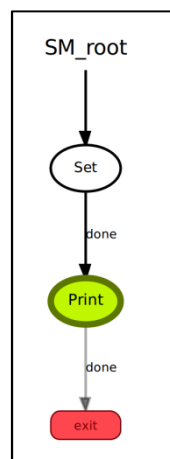
EXERCISES:

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:

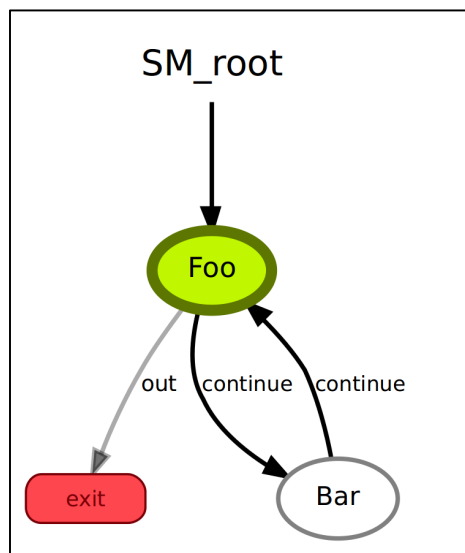
- Foo – A state that increase his inside counter by one every time it's called and if the counter is 3 or higher return with **"out"**, otherwise it return **"continue"**
- Bar – A state that wait 2 seconds, before returning **"continue"**

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

Try out:

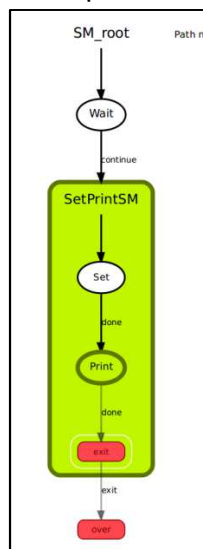
```
$ roslaunch state_machine_tutorial exo_basic_statemachine.launch  
exercise:=1
```

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
```

Example Advanced:

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.

```
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:

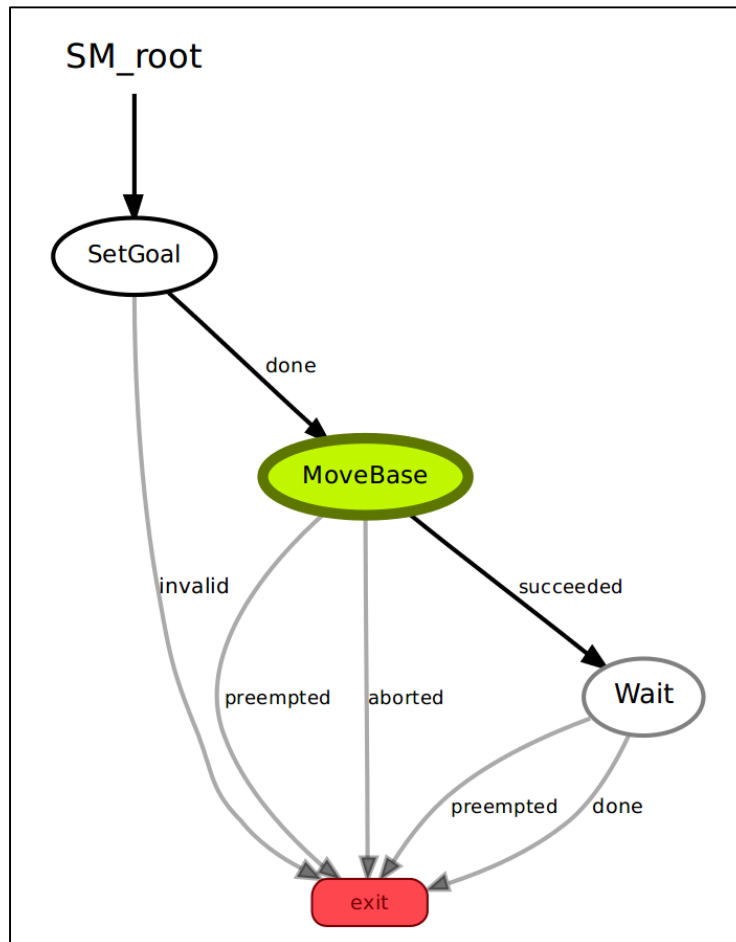
```
$ sudo apt-get install ros-kinetic-turtlebot3 ros-kinetic-turtlebot3-gazebo
```

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"})
```

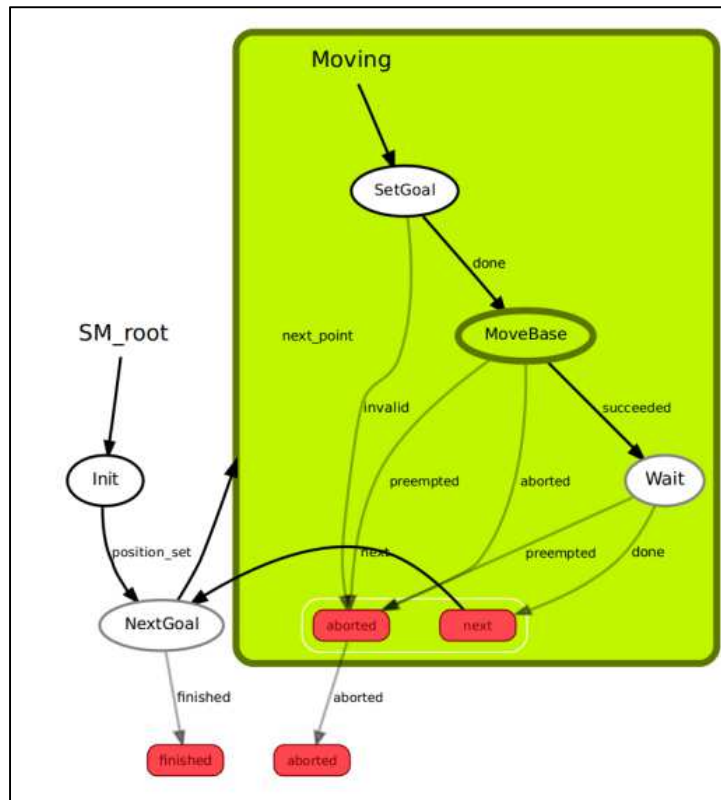
Restart the turtlebot simulation:

```
$ 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.

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 Concurrency

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/Concurrency%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 childs 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 returns "**positiv**" then the concurrence return "**positiv**". The same for "**negative**"

```
with FooBar_cc:
```

ROS-I Training by ROSIN

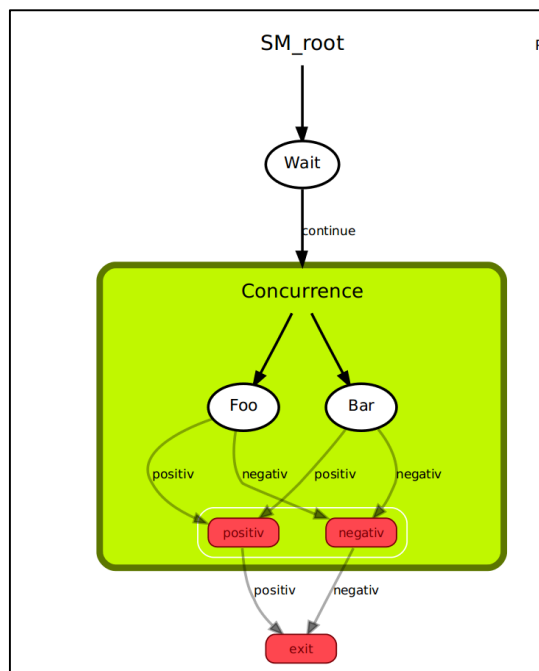
```
FooBar_cc.add('Foo',ConcurrenceState())  
FooBar_cc.add('Bar',ConcurrenceState())
```

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
```

2.4 SMACH Introspection

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.

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.

3.1 SCXML Files

The SCXML format is an XML-based markup language use to create state machine. It can describe complex finite state machine.

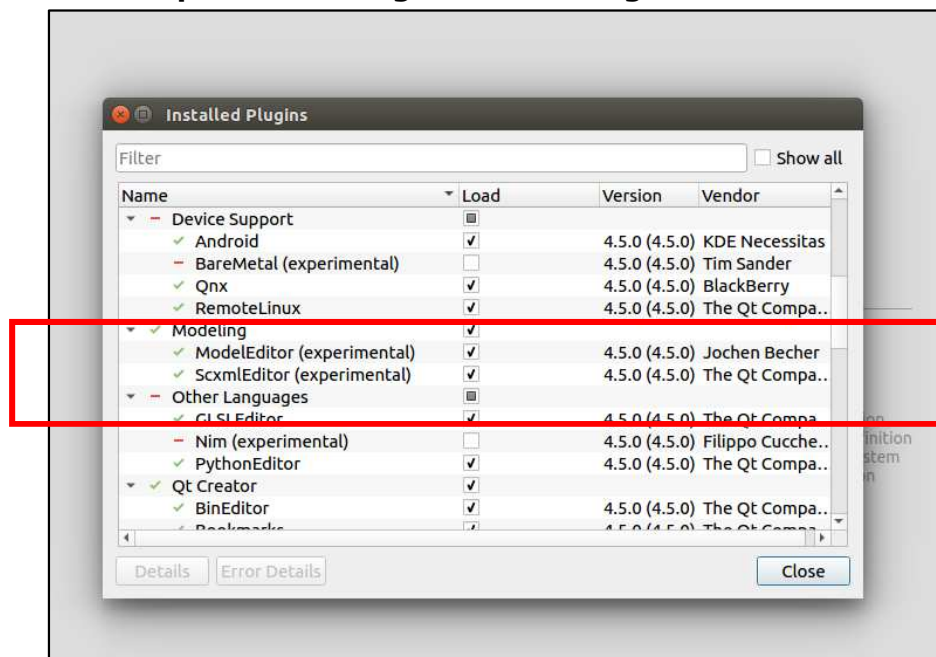
The standard description can be found here: <https://www.w3.org/TR/scxml/>

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).

As the plugin is not enabled by default, you should first enable it. To do so in QtCreator select :

Help > About Plugins > Modeling > ScxmlEditor

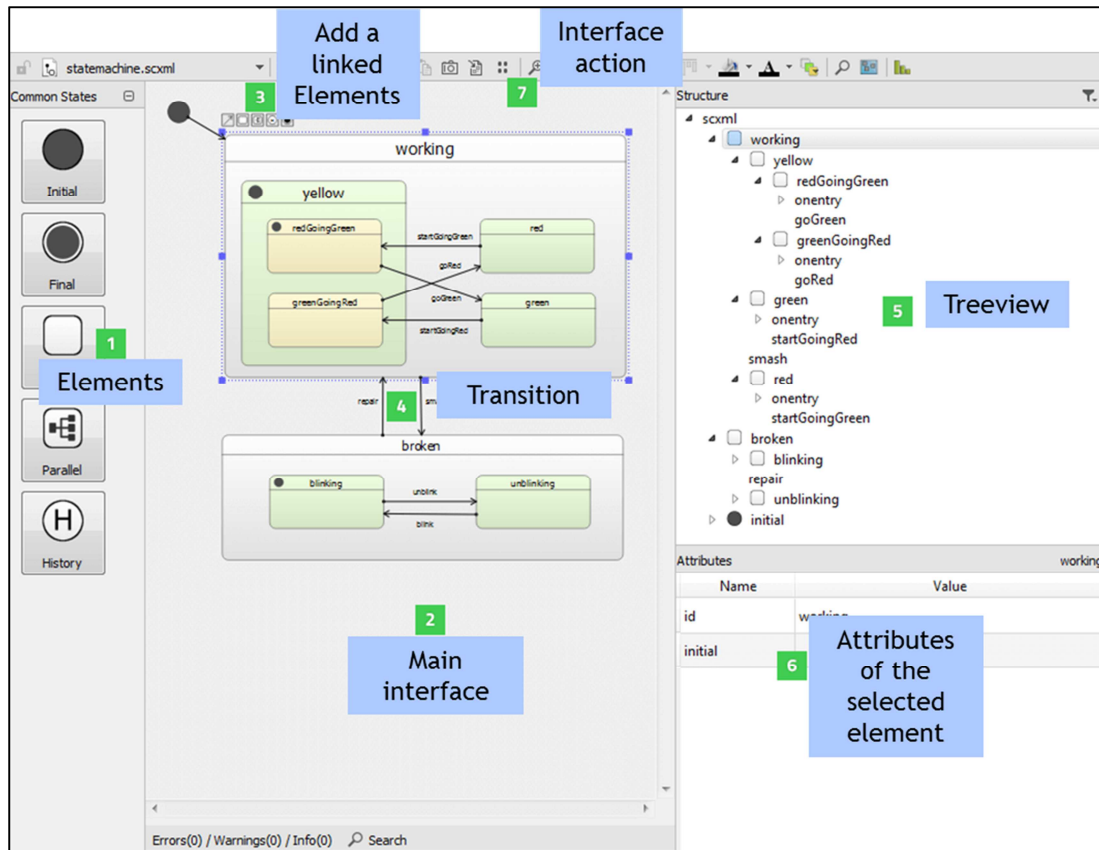


You have to restart QtCreator.

To start a new SCXML file: **File > New File or Project > Files and Classes > Modeling > State Chart.**

Give it a name and a location. For this tutorial, all premade SCXMLs have been saved inside the package *state_machine_tutorial* in resources/scxml folder.

Here is a description of the interface:

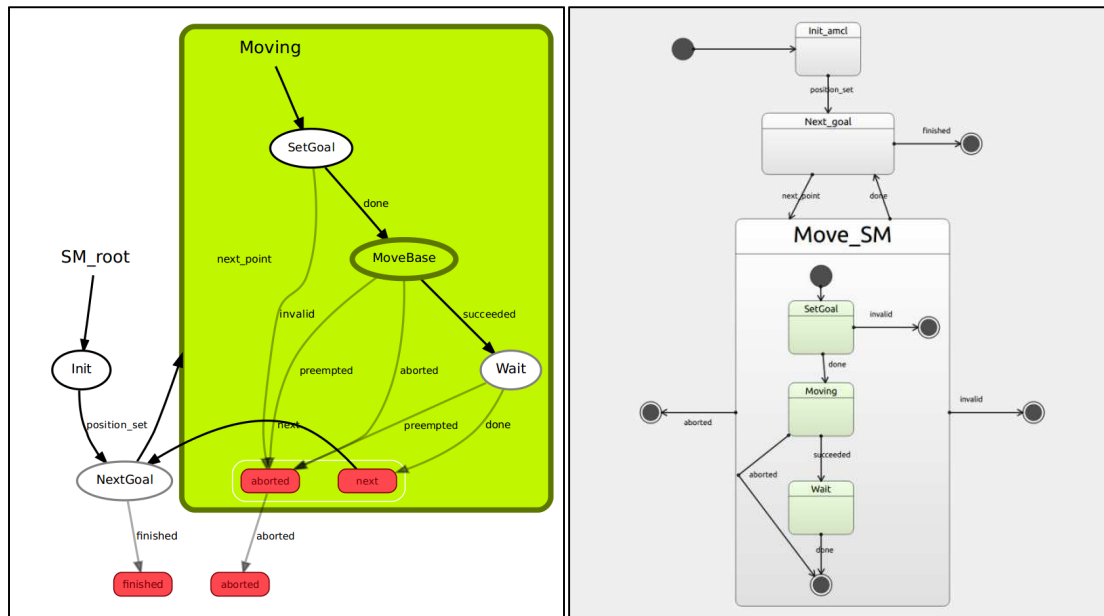


To add an element:

- Drag and drop from the left side to the main interface.
- To nest, drag inside another element.
- To create a transition, select the element from which you want to transition then right click on the element you want to transition to.

EXERCISES:

Try to create one of the previously seen state machines for example:



For userdata, you can define them directly from the datamodel.

To add a datamodel at a state, right click on the state in the treeview, select datamodel, then right click on the newly added datamodel and select data. It's also possible to add datamodel to the scxml the same way.

Note:

- You cannot copy/paste a datamodel from a state to another state. You can copy/paste states, or data from datamodel.
- When interpreted in the SSM, data are actually string. In order to use them as python structure like list use ***ast.literal_eval***, maybe coupled with a type checking if you go through the state multiple time.

For example:

```
if(isinstance(ud.goal, basestring)):
    ud.goal = ast.literal_eval(ud.goal) #convert the string into a list
```

You can see SCXML file in a SCXML format by going into "DESIGN" but you cannot it here.

Save your SCXML files.

3.2 From SMACH to SSM

You can use directly the SMACH state but you need to define all your userdata as **io_keys** (input_keys and output_keys are not retrieved from the SCXML interpreter and can generate some consistency error) and all the SSM provided functions may not be working. To avoid problem, I suggest convert the SMACH state into a SSM state that provide all the interfaces to SSM.

To do so:

- 1 Add this line to import the ssmState interface inside your python script.

```
from airbus_ssm_core import ssm_state
```

- 2 Convert smach.State (the class you inherited and the initialisation of the interface) to ssm_state.ssmState. This will provide all the SSM interfaces on top of SMACH.
- 3 Change the input_keys and output_keys into io_keys (be sure to check you execute function if you change some names)
- 4 Change the execute(self,ud) function to execution(self,ud)

And with these 4 steps, your SMACH State is now a SSM compatible State.

EXAMPLE:

This is the SMACH State:

```
class WaitState(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=["continue"], input_keys=["sleep_time"])

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

And the conversion in a SSM state:

```
class WaitState(ssm_state.ssmState):
    def __init__(self):
        ssm_state.ssmState.__init__(self, outcomes=["continue"], io_keys=["sleep_time"])

    def execution(self, ud):
        rospy.sleep(ud.sleep_time)
        return "continue"
```

EXERCISES:

Try to convert some previously created / used SMACH state into a SSM state.

Most of them can be found inside the **states.py** file that can be found in the package *state_machine_tutorial* into the *src/ssm_tutorial/* folder.

3.3 SSM Usage

In order to make the SCXML a support for executing Python State machine you need to provide two things to the SSM interpreter:

- 1- A way to retrieve the Python code by the interpreter.
- 2- A link between the SCXML state (inside the SCXML file) and the class containing the Python code (the SSM state class)

1. There is an XML File that provides all the information for the interpreter to retrieve the Python classes.

In the tutorial it's located in the package *state_machine_tutorial*, resources folder: **register.xml**

```
<?xml version="1.0"?>
<skills>
```

```
<?xml version="1.0"?>
<skills>
  <skill name="Foo"          pkg="state_machine_tutorial" module="ssm_tutorial.states" class="FooState" />
  <skill name="Bar"         pkg="state_machine_tutorial" module="ssm_tutorial.states" class="BarState" />
  <skill name="SetGoal"     pkg="state_machine_tutorial" module="ssm_tutorial.states" class="SetGoal" />
  <skill name="Wait"        pkg="state_machine_tutorial" module="ssm_tutorial.states" class="Wait_ssm" />
  <skill name="MoveBase_ac" pkg="state_machine_tutorial" module="ssm_tutorial.states" class="MoveBase_ac" />
  <skill name="SetInitialPose" pkg="state_machine_tutorial" module="ssm_tutorial.states" class="SetInitialPose" />
  <skill name="NextGoal"    pkg="state_machine_tutorial" module="ssm_tutorial.states" class="NextGoal" />
</skills>
```

Skill attributes:

- name="Foo" - The reference to call this state inside the the SCXML
- pkg="state_machine_tutorial" - The ROS package where the classes can be found
- module="ssm_tutorial.states" - Which Python file to use
- class="FooState" - The name of the Python class

This file is linked in the SCXML with a data inside the SCXML's datamodel thought the skill_file data like this:

The screenshot shows a GUI with a tree view of the SCXML datamodel and an attributes table below it.

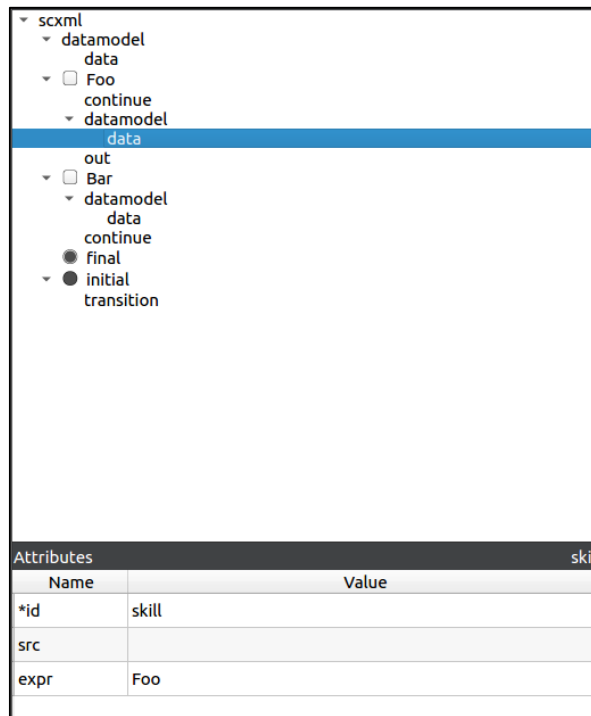
Tree View:

- scxml
 - datamodel
 - data
 - data
 - data
 - Move_SM
 - Wait
 - done
 - datamodel
 - data
 - initial
 - transition
 - Moving
 - datamodel
 - data
 - succeeded
 - aborted
 - SetGoal
 - datamodel
 - data
 - invalid
 - done
 - Final_4
 - Final_2
 - done
 - invalid
 - aborted

Attributes Table:

Attributes		skill_file
Name	Value	
*id	skill_file	
src		
expr	\${state_machine_tutorial}/resources/register.xml	

- The link between a SCXML state and the Python class is then done by adding a data for a SCXML with **"id"** set to **"skill"** and the **"expr"** set to the xml skill name (the one given in in the xml file) :



The screenshot shows a GUI with a tree view on the left and an 'Attributes' table on the right.

Tree View:

- scxml
 - datamodel
 - data
 - Foo
 - continue
 - datamodel
 - data (highlighted)
 - out
 - Bar
 - datamodel
 - data
 - continue
 - final
 - initial
 - transition

Attributes Table:

| Name | Value |
|------|-------|
| *id | skill |
| src | |
| expr | Foo |

There are also some rules for the interpreter:

- Only atomic state can have skill associate. Compound state and parallel will not be executed
- Transition's **event** refers to the outcome and **target** is the next target id.
- All outcomes must be linked to something (final state or other state)
- When exiting a compound state, the event of the transition targeting the final state must be the same than the one from the compound state to the next state.
- All compound state must have an initial state declared.

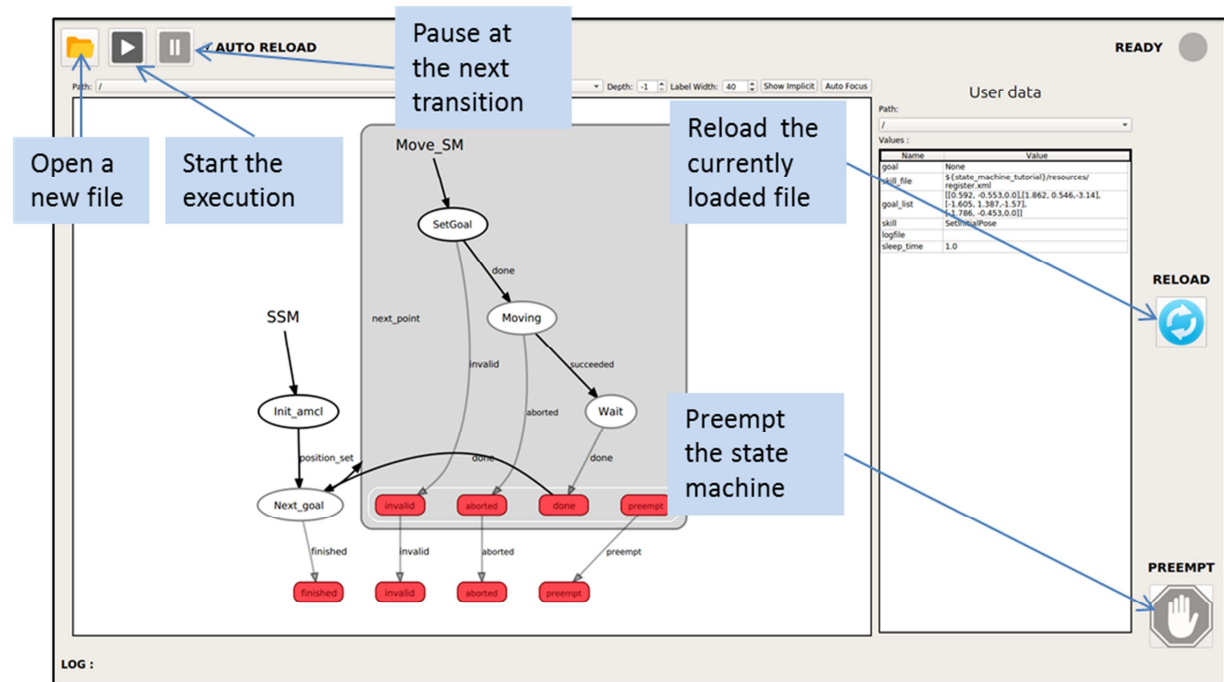
Now you can make an SCXML file compatible with the SSM framework.

In order to launch an SCXML file you can either call the launch file:

```
$ roslaunch airbus_ssm_core ssm.launch scxml:=path/to/the/file
```

Or use the dedicated GUI:

```
$ roslaunch airbus_ssm_plugin ssm_plugin_sstandalone.launch
```



EXERCISES:

You can try the already existing SCXML file or create your own.