

ROS-I Academy Training

State Machine Based Programming

Ludovic DELVAL
Fraunhofer IPA
2018

Overview

- ▶ State Machine Generalities
- ▶ SMACH
- ▶ SCXML State Machine framework

Before Starting

Go into your catkin workspace.

```
$ cd src
$ git clone https://github.com/ipa-led/
  state_machine_tutorial.git
$ cd ..
$ catkin_make
$ roslaunch state_machine_tutorial exo_basicstates.
  launch exercise:=0
```

The slides and the tutorial can be found in the docs folder.

Learning Objectives

You will

- ▶ get to know the basic of **state machine** functionment to prototype a sequence of action
- ▶ learn how to **program using SMACH** to build a state machine
- ▶ see **how to use a GUI** to create your state machine

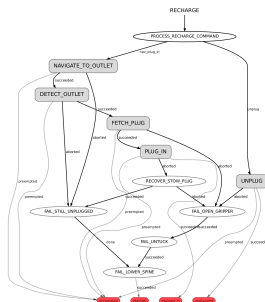
Outline

- ▶ State Machine Generalities
- ▶ SMACH
- ▶ SCXML State Machine (SSM)

Outline

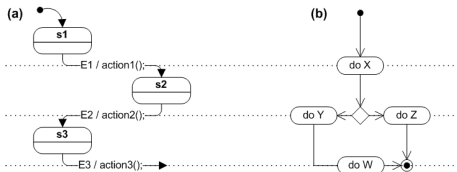
- ▶ State Machine Generalities
- ▶ SMACH
- ▶ SCXML State Machine (SSM)

State Machine



A Finite State Machine is a list of a finite number of states. The Finite State Machine change from a state to another state by a transition. The Finite State Machine is defined by its states, its initial state and its transitions.

State diagrams vs flowcharts



A State Diagram (State Machine) perform an action in response of an event then it may change its state.

A Flowcharts don't need event to transition from state to state. It transitions upon completion of the action.

State Machine Application

Airbus Floor Screwing

<https://drive.google.com/file/d/0B3PdNHHyyoTFY3pPemdJMkxOdXM/view?usp=sharing>

Denso Cell Pick and Place

<https://drive.google.com/file/d/0B3PdNHHyyoTFelFFenJkcEQxbWM/view?usp=sharing>

TiaGo Pick and Place

<https://drive.google.com/file/d/0B3PdNHHyyoTFcldBwm5ibzFsU28/view?usp=sharing>

State Machine

When use a State Machine ?

- ▶ You can describe your task as a sequence of actions and transitions.
- ▶ You want to synchronize different high/middle level modules.
- ▶ You need to have a clear view of what your robot is currently doing.

When not use a State Machine ?

- ▶ You have list of unordered tasks.
- ▶ You want to do low-level efficient tasks.
- ▶ You need to create a very complex behavior sequence.

Glossary

State

An irreducible action.

State machine

A finite number of states or state machines linked by transitions.

Transition

A link between states or state machines

Outline

- ▶ State Machine Generalities
- ▶ SMACH
- ▶ SCXML State Machine (SSM)

SMACH

What is SMACH ?

SMACH is a Python State Machine Framework. SMACH core libraries are ROS-independent but it also provides some ROS interaction. The SMACH programation is oriented to be like a flow chart. Meaning the states are ACTION and not CONFIGURATION.

What SMACH provides ?

- ▶ Fast prototyping: The straightforward Python-based SMACH syntax makes it easy to quickly prototype a state machine and start running it.
- ▶ Complex state machines: SMACH allows you to design, maintain and debug large, complex hierarchical state machines.
- ▶ Introspection: SMACH gives you full introspection in your state machines, state transitions, data flow, etc.

SMACH ROS Wiki : <http://wiki.ros.org/smach>

SMACH basic structures

smach.State

This is the core class of SMACH. It's the basic building bricks to setup your state machine. You have to make your classes inherit this one.

Important functions :

```
__init__(outcomes=[], input_keys=[], output_keys=[], io_keys=[])
```

outcomes List of strings that describes the possible results of the execution functions

input_keys List of strings that give you read access on the userdata

output_keys List of strings that give you write access on the userdata

io_keys List of strings that give you read and write access on the userdata

```
execute(self, ud)
```

Function to reimplement in your class. It's the code that is executed when the state become active.

ud Userdata for the scope in which this state is executing

SMACH basic structures

Simple State class example :

```
class Publish_msg(smach.State):
    #This class publish the message then reset the message and return "ok".
    #If the message is not set, it fails.
    def __init__(self):
        #Init of the smach class with the outcomes and the io_keys
        smach.State.__init__(self,outcomes=["ok","fail"],io_keys=["msg"])
        #init of the publisher
        self.msg_publisher = rospy.Publisher('/msg_topic',String)

    def execute(self, ud): #Execute function reimplemented
        if ud.msg is not None:
            self.msg_publisher.publish(ud.msg)
            return "ok"
        else:
            return "fail"
```

SMACH basic structures

smach.StateMachine

A container which can include other states or containers and execute them one at a time. It has the same *init* function than the State class and should be opened to add states or containers in it then closed (or use the keyword "**with**"). The first state added will be the initial (or you can specify it by using the **set_initial_state** function).

Important function :

```
add(label, state, transitions=None, remapping=None)
```

This function add a state to the state machine.

label The name to reference the state that you add.

state The class instance you want to add.

transitions A dictionary mapping state outcomes to other state labels or container outcomes.

remapping A dictionary mapping local userdata keys to userdata keys in the container.

Other functions can be found here : http://docs.ros.org/groovy/api/smach/html/python/smach.state_machine.StateMachine-class.html

SMACH basic structures

Simple State Machine class example :

```
class Message_set_and_publish(smach.StateMachine):
    #This class setup a string message and publish it.
    def __init__(self):
        smach.StateMachine.__init__(self, outcomes=["ok","fail"],
                                     input_keys=["message"],
                                     output_keys=["message"])
        self.userdata.message = None #initialisation of the message userdata

    with self: #We use the keyword with to open and close the container
        #add the setup message state
        self.add('Setup', #label
                 SetupMsg(), #class instance
                 {"ok":"'Publish','fail':'fail'}, #transitions mapping
                 {"msg":"message"}) #userdata remapping
        #add the Publish message state
        self.add('Publish',Publish_msg(),
                 {"ok" : "ok",
                  "fail": 'Setup'}
                 {"msg":"message"})
```

SMACH basic structures

smach.Concurrence

A container which can include other states or containers and execute them in parallel (using threads). It works quite similarly to a State Machine but you need to define the outcome maps (Which combinations of states outcomes will trigger which concurrence outcome) and a default outcome (in case none of the combinations are met).

Example :

```
sm_con = smach.Concurrence(outcomes=['outcome4','outcome5'],
                           default_outcome='outcome4',
                           outcome_map={'outcome5':
                                         { 'FOO': 'outcome2',
                                           'BAR': 'outcome1' }})
```

```
# Open the container
with sm_con:
    # Add states to the container
    smach.Concurrence.add('FOO', Foo())
    smach.Concurrence.add('BAR', Bar())
```

SMACH usefull tools

Preemption

There is a built-in preemption propagation system in SMACH. You can check for preemption signal or either trigger it. It will be propagated to the whole state machine (e.g. in a concurrence, all the state running will trigger the preempt signal if one is triggered. You have to manually code the behavior of states in case a preempt signal is received but for containers, it will trigger the termination of the whole State Machine.

Functions :

```
self.request_preempt() #trigger the preempt signal
```

```
self.preempt_requested() #Return True if a preempt has been requested
```

```
self.recall_preempt() #set the preempt signal to False
```

SMACH usefull tools

Introspection

SMACH containers can provide a debugging interface (over ROS) which allows a developer to get full introspection into a state machine. Using the `smach_viewer` package, you can visualize the flow of your state machine and the user data during runtime.

Functions :

```
sis = smach_ros.IntrospectionServer('server_name', sm, 'SM_ROOT')
sis.start() #start the server
sm.execute() #start the state machine
rospy.spin()
sis.stop() #stop the server
```

This create an instance of the SMACH Introspection server. The parameters are :

'server_name' A name that you give as a namespace for introspection topics.

sm The state machine you want to inspect To have access to your whole state machine, give your highest state machine instance.

'SM_ROOT' The name that you give at your top level state machine.

To run SMACH Viewer : ***roslaunch smach_viewer smach_viewer.py***

SMACH ROS template

CBState

An function wrapper into an SMACH state.

<http://wiki.ros.org/smach/Tutorials/CBState>

ServiceState

A service client conversion into an SMACH state.

<http://wiki.ros.org/smach/Tutorials/ServiceState>

SimpleActionState

An action client wrapper into an SMACH state.

<http://wiki.ros.org/smach/Tutorials/SimpleActionState>

SMACH tutorial

Start the SMACH Tutorial !

Outline

- ▶ State Machine Generalities
- ▶ SMACH
- ▶ SCXML State Machine (SSM)

State Chart XML Standard

SCXML is a XML-based markup language use to describe state-machine. Is possible to describe nested states or state machines, parallel executions and data model.

Example :

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initial="Foo">
  <datamodel>
    <data id="skill_file" expr="${airbus_ssm_core}/resources/empty_register.xml"/>
  </datamodel>
  <state id="Foo">
    <transition type="external" target="Bar" event="next">
    </transition>
  </state>
  <state id="Bar">
    <transition type="external" event="next" target="final"/>
  </state>
  <final id="final">
  </final>
</scxml>
```

The SCXML standard web site : <https://www.w3.org/TR/scxml/>

SCXML Core Elements

Brief description of every elements and their interactions.

<scxml>

This is the root of the SCXML file.

Attribute :

initial The initial state's id.

Childs :

<datamodel> One datamodel that contains the data.

<state> Atomic or compound states.

<parallel> Parallel elements.

<final> One or more final outcome.

SCXML Core Elements

<state>

It can be considered as a compound state if it contains another state or parallel elements, or atomic state if not.

Attribute :

id An unique identifier.

Childs (Atomic and Compound):

<datamodel> One datamodel that contains the data.

<onentry> Elements to be executed before entering the state.

<onexit> Elements to be executed after the exiting the state.

<transition> The event that trigger the exit of the state and the target associate.

Childs (Compound):

<state> Child states.

<parallel> Child parallels elements.

<initial> The initial element (must target the initial state).

<final> One or more final outcome.

SCXML Core Elements

<parallel>

Start every child states simultaneously.

Attribute :

id An unique identifier.

Childs :

<datamodel> One datamodel that contains the data.

<onentry> Elements to be executed before entering the state.

<onexit> Elements to be executed after the exiting the state.

<transition> The event that trigger the exit of the parallel and the target associate.

<state> Child states.

<parallel> Child parallels elements.

SCXML Core Elements

<transition>

Define the trigger event, eventually the condition that exit the state and the next state to be executed.

Attribute :

- event** The trigger event for the transition.
- cond** (optional) Additional condition for the transition.
- target** Next state's id.

<datamodel/data>

A list of data that can be transmit between states.

Attribute (of data elements):

- id** A unique identifier.
- expr** (optional) A value associate with the data that is set before entering the state.

SCXML State Machine Framework

In order to make an SCXML file working as a SMACH state machine, there is a SCXML interpreter. This framework also extends SMACH at different levels.

Mapping between SCXML and SMACH

SCXML	SMACH
<SCXML /> Root	Main State Machine
<State /> Atomic State	State
<State / State /> Compound State	State Machine
<Parallel /> Parallel	Concurrence
<State / Transition /> State Transition <ul style="list-style-type: none"> Event Target 	Transitions <ul style="list-style-type: none"> Outcome Outcome's target
<Parallel / Transition /> Parallel Transition <ul style="list-style-type: none"> Cond 	Outcomes_map <ul style="list-style-type: none"> Logic combination of outcomes
<Datamodel / Data />	Userdata
<Initial />	Initial
<Final />	State Machine (self outcomes)

SSM Framework

SSM extend the SMACH framework with new functionalities or ease the access to some. To have access to most of them you have to use the SSM derivate classes (like ssmState or ssmStateMachine).

- ▶ `"/preempt"` topic that trigger the preempt flag in every states.
 - ▶ `"/pause"` topic that pause the state machine at the next transition (still finishing the current running atomic state).
 - ▶ `"onEntry"` and `"onExit"` to save logs and execute python script before or after executing a state.
-
- ▶ **Release Source Code** : https://github.com/ipa320/airbus_coop
 - ▶ **Devel Source Code** : https://github.com/ipa-led/airbus_coop
 - ▶ **ROS Wiki Page** : http://wiki.ros.org/airbus_coop

From SMACH to SSM

You can use SMACH inside the SSM Framework, but you will lose most of the extension provided. Here are the steps to convert from SMACH to SSM:

1. 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.
2. Change the `input_keys` and `output_keys` into `io_keys` (be sure to check your `execute` function if you change some names)
3. Change the `execute(self,ud)` function to `execution(self,ud)`

SMACH

```
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"
```

SSM

```
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"
```

SSM Framework Rules

1. The path of an XML file is put in the <scxml> datamodel with the id **skill_file**. This file refers to all the SMACH (or SSM) states python class and where to find them.
2. The atomic state are linked with an instance of a SMACH (or SSM) thought a data inside the atomic state datamodel. The "id" has to be set as **skill** and the "expr" the name which refer to the python class inside the XML skill_file.
3. Only atomic state can have skill associate. Compound state and parallel will not be executed.
4. Transition's "event" refer to the linked python class outcome and "target" to the next target state id.
When exiting a compound state, the target should be one of the final state of the compound state, and the compound state must have a transition with the same "event" and the next target after exiting the compound state.
5. Data inside a state datamodel are automatically added to the state "io_keys" with the key declared in the "id". If an "expr" is associate, then it will be reimplacing the current userdata value (**it will considered as a String !!**). If there is no "expr", the value is unchanged or initialize to "" (empty String).
6. All outcomes need to be connected.
7. All compound states need to have one initial state.

SSM Framework Rules

Example (rule 1) :

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initial="Foo">
  <datamodel>
    <data id="skill_file" expr="${airbus_ssm_core}/resources/empty_register.xml"/>
  </datamodel>
  ....
</scxml>
```

XML Skill file :

```
<?xml version="1.0"?>
<skills>
  <skill name="Input" pkg="airbus_ssm_tutorial" module="airbus_ssm_tutorial1.skills"
    class="Input"/>
  <skill name="isPrime" pkg="airbus_ssm_tutorial" module="airbus_ssm_tutorial1.skills"
    class="isPrime"/>
  <skill name="Primes" pkg="airbus_ssm_tutorial" module="airbus_ssm_tutorial1.skills"
    class="Primes"/>
</skills>
```

SSM Framework Rules

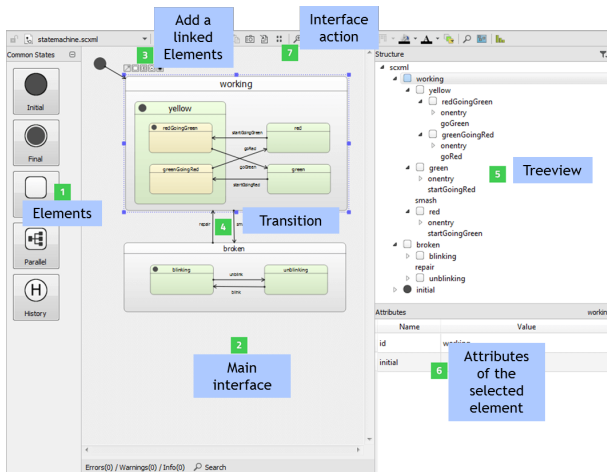
Example (rule 2) :

```
<state id="IsPrime">
  <datamodel>
    <data id="skill" expr="isPrime"/>
  </datamodel>
  ...
</state>
```

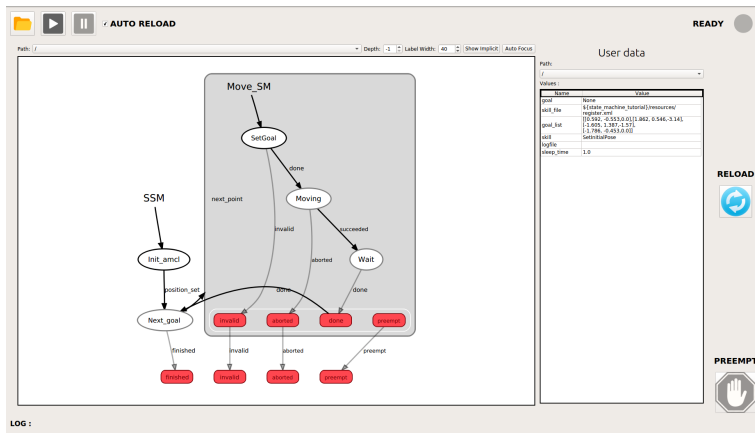
Example (rule 4) :

```
<state id="Parent">
  <initial>
    <transition type="external" target="foo"/>
  </initial>
  <state id="foo">
    <transition event="to_bar1" target="Final"/>
    <transition event="to_bar2" target="Final"/>
  </state>
  <final id="Final"/>
  <transition event="to_bar1" target="bar1"/>
  <transition event="to_bar2" target="bar2"/>
</state>
```

Usefull tools : Qt SCXML editor



Usefull tools : SSM GUI



SSM tutorial

Start the SSM Tutorial !