Finite state machine description in Robotflow

Introduction

A finite state machine (FSM) is a useful data structure to express actions with a given sequence of events. In a robotic application, a finite state machine can be used to activate or deactivate behaviors in time. This document defines how to describe a finite state machine from an XML file into a Robotflow node.

Creating an XML finite state machine description

The first step is to write an XML document containing the information about the FSM. XML was chosen for the description language because of its widespread use, its good readability and its ease of use. A user does not need to know all the details of XML to define a FSM. One only needs to understand the usage of tags and their hierarchies. For more information on XML, the reader is referred to the following web site:

http://www.ibiblio.org/xml/books/xmljava/chapters/ch01.html

Defining a FSM document

This first part of the XML file must first indicate that it is in XML format. Then, it must indicate that the document is a FSM with the <FSM> tag.

<FSM> tag definition

Attribute "name": this attribute must contain a string with the name for the given FSM. This name will be used as the Robotflow node name.

Children: all the FSM elements.

Example:

```
<?xmlversion="1.0"?>
<FSM name="ForagerCoordination">
        <!-- Description of the FSM! -->
</FSM>
```

The FSM elements can be either: *Input*, *Output*, *StartState*, *State* or *Transition*. The following text defines these different elements, their associated tags, their usage and their possible children.

- ***Input***

  An *Input* usually represents the actual value of a given trigger. These triggers are use to make transition between states. But fundamentally, an Input can be used to store and process information about external events.

<Input> tag definition

Attribute "name": this attribute must contain a string with the name for the given Input. The name must be unique in the FSM.

Attribute "type": this attribute must contain a string with the type for the given Input. The types currently supported are: int, float and bool.

Children: none.

Example:
<Input name="MyInput1" type="bool"/>

- ***Output***
  An *Output* is usually a consequence of the current state. It gives an action or information about what should be done according to the current events for a given time. For example, *Output*s can be used to coordinate behaviors.

<Output> tag definition

Attribute "name": this attribute must contain a string with the name for the given Output. The name must be unique in the FSM.

Attribute "type": this attribute must contain a string with the type for the given Output. The types currently supported are: int, float and bool.

Children: none.

Example:
<Output name="MyOutput1" type="bool"/>

- ***StartState***
  Like its name indicates, this tag defines the starting state of the FSM. This state can be used to only initialize the FSM or it can be a state like all the other ones. This tag is essential because without a starting state, no transition could occur.

<StartState> tag definition

Attribute "name": this attribute must contain a string with the name of a valid State.

Children: none.

Example:
<StartState name="MyState"/>

- ***State***
  A *State* is used to make actions and drive outputs accordingly to the current events for a given time. This can be achieved using the tag *SetOutput*.

<State> tag definition

Attribute "name": this attribute must contain a string with the name for the state. The name must be unique in the FSM.

Children: SetOutput.

Example:
<State name="MyState"/>
<!-- Actions for this state! -->
</ State >

  - ***SetOutput***
    This tag defines a value for a given output. The value will be effective if the current state is the one using this tag.

<SetOutput> tag definition

Attribute "name": this attribute must contain a string with the name of a valid Output.

Attribute "value": this attribute must contain a string with the value for the given Output. The values can be of one of the following supported types: int, float and bool.

Children: none.

Example:
<SetOutput name="MyOutput" value="true"/>

- ***Transition***

  A *Transition* responds to a trigger or a new event and results in the change of the current state. A *Transition* must contain one or more *Condition* tags to indicate what triggers the transition. If there is more than one *Condition*, the one or more *Operator* tags must be used to define the logic to apply between the *Condition*s.

<Transition> tag definition

Attribute "from": this attribute must contain a string with the name of a valid State that was used before the trigger(s).

Attribute "to": this attribute must contain a string with the name of a valid State that will become the current state after the trigger(s).

Children: Condition, Operator.

Example:
<Transition from="MyState" to="MyNextState">
<!-- Conditions for this transition! -->
</ Transition >

  - Operator

    An *Operator* is used to apply a certain logic between two or more elements. Typically, an *Operator* would be used to indicate a certain logic between the *Left* and *Right* part of a *Condition* or to group two or more *Condition*s.

<Operator> tag definition

Attribute "type": this attribute must contain a string with the name of a valid operator. The operators currently supported are: and, or, equals, greater than, less than, greater than or equal, less than or equal.

Children: Condition or none.

Example:

```
<Operator type="and" >
        <!-- Condition1! -->
        <!-- Condition2! -->
</ Operator  >
```

- ***Condition***

  A *Condition* is typically used in a FSM to define when to make transitions. *Input*s are compared to constants or other variables to trigger an event (a change of state). A *Condition* must have three children: a *Left*, an *Operator* and a *Right*. The order is not important.

<Condition> tag definition

Attribute : none

Children: Left, Operator and Right.

Example:

```
<Condition>
        <!-- Left! -->
        <!-- Operator! -->
        <!-- Right! -->
</ Condition   >
```

  - ***Left***

    This tag defines the left part of the condition. It can use a constant or a variable.

<Left> tag definition

Attribute "variable" OR "constant": this attribute must contain a string with the name of a valid Input or of a constant value.

Attribute "type": this attribute must contain a string with the type for the given variable or constant. The types currently supported are: int, float and bool.

Children: none.

Example:
<Left variable="MyIntVar" type="int"/>

- ***Operator***
  This tag is very similar to the other *Operator* tag used to combine more than one *Conditions*. The only difference is it does not have children since the left and right part of the *Condition* are always required.

<Operator> tag definition

Attribute "type": this attribute must contain a string with the name of a valid operator. The operators currently supported are: and, or, equals, greater than, less than, greater than or equal, less than or equal.

Children: none.

Example:
<Operator type="equals" >

- ***Right***
  This tag defines the right part of the condition. It can use a constant or a variable.

<Right> tag definition

Attribute "variable" OR "constant": this attribute must contain a string with the name of a valid Input or of a constant value.

Attribute "type": this attribute must contain a string with the type for the given variable or constant. The types currently supported are: int, float and bool.

Children: none.

Example:
<Right variable="MyFloatVar" type="float"/>

Generating a FSM node in Robotflow

Once an XML description is available, one can generate a Robotflow node. The user should try to look at the example in $ROBOTFLOW/FSM/examples/generate_v2.n. In this file, double click on the FSMGenerator node and change the name of the FSM_FILE parameter. Remember, the name must also include th path if it is not in the current directory. Then, one only has to click on “Execute”. This will generate and install the node. But to be visible, the user must quit and restart vflow. The new node will be in the menu RobotFlow->FSM. You can always make changes to your XML file and regenerate your node whenever necessary.

A complete exemple

This section describes a FSM used to coordinate behaviors of a forager robot.

```
<?xml version="1.0"?>
<FSM name="ForagerCoordination">
   <Input name="FoundPuck" type="bool"/>
   <Input name="HasPuck" type="bool"/>
   <Input name="IsAtBase" type="bool"/>
   <Input name="FoundBase" type="bool"/>
   <Input name="HasLeftBase" type="bool"/>

   <Output name="AvoidNear" type="float"/>
   <Output name="AvoidFar" type="float"/>
   <Output name="SafeVelocity" type="float"/>
   <Output name="Grasp" type="float"/>
   <Output name="FindPuck" type="float"/>
   <Output name="FindBase" type="float"/>
   <Output name="LeaveBase" type="float"/>
   <Output name="Noise" type="float"/>

   <StartState name="Wander"/>

   <State name="Wander">
      <SetOutput name="AvoidNear" value="1.0"/>
      <SetOutput name="AvoidFar" value="1.0"/>
      <SetOutput name="SafeVelocity" value="1.0"/>
      <SetOutput name="Grasp" value="0.0"/>
      <SetOutput name="FindPuck" value="1.0"/>
      <SetOutput name="FindBase" value="0.0"/>
      <SetOutput name="LeaveBase" value="0.0"/>
      <SetOutput name="Noise" value="0.5"/>
   </State>

   <State name="GetPuck">
      <SetOutput name="AvoidNear" value="1.0"/>
      <SetOutput name="AvoidFar" value="0.0"/>
      <SetOutput name="SafeVelocity" value="0.2"/>
```

```
    <SetOutput name="Grasp" value="1.0"/>
    <SetOutput name="FindPuck" value="1.0"/>
    <SetOutput name="FindBase" value="0.0"/>
    <SetOutput name="LeaveBase" value="0.0"/>
    <SetOutput name="Noise" value="0.0"/>
</State>

<State name="FindBaseLocation">
    <SetOutput name="AvoidNear" value="1.0"/>
    <SetOutput name="AvoidFar" value="1.0"/>
    <SetOutput name="SafeVelocity" value="1.0"/>
    <SetOutput name="Grasp" value="1.0"/>
    <SetOutput name="FindPuck" value="0.0"/>
    <SetOutput name="FindBase" value="1.0"/>
    <SetOutput name="LeaveBase" value="0.0"/>
    <SetOutput name="Noise" value="0.5"/>
</State>

<State name="GoToBase">
    <SetOutput name="AvoidNear" value="1.0"/>
    <SetOutput name="AvoidFar" value="0.0"/>
    <SetOutput name="SafeVelocity" value="0.7"/>
    <SetOutput name="Grasp" value="1.0"/>
    <SetOutput name="FindPuck" value="0.0"/>
    <SetOutput name="FindBase" value="1.0"/>
    <SetOutput name="LeaveBase" value="0.0"/>
    <SetOutput name="Noise" value="0.0"/>
</State>

<State name="LeaveBase">
    <SetOutput name="AvoidNear" value="1.0"/>
    <SetOutput name="AvoidFar" value="0.0"/>
    <SetOutput name="SafeVelocity" value="0.2"/>
    <SetOutput name="Grasp" value="0.0"/>
    <SetOutput name="FindPuck" value="0.0"/>
    <SetOutput name="FindBase" value="0.0"/>
    <SetOutput name="LeaveBase" value="1.0"/>
    <SetOutput name="Noise" value="0.0"/>
</State>

<Transition from="Wander" to="GetPuck">
    <Condition>
        <Left variable="FoundPuck" type="bool"/>
        <Operator type="equals"/>
        <Right constant="true" type="bool"/>
    </Condition>
</Transition>

<Transition from="GetPuck" to="FindBaseLocation">
```

```
    <Condition>
      <Left variable="HasPuck" type="bool"/>
      <Operator type="equals"/>
      <Right constant="true" type="bool"/>
    </Condition>
  </Transition>

  <Transition from="GetPuck" to="Wander">
    <Condition>
      <Left variable="FoundPuck" type="bool"/>
      <Operator type="equals"/>
      <Right constant="false" type="bool"/>
    </Condition>
  </Transition>

  <Transition from="FindBaseLocation" to="GoToBase">
    <Condition>
      <Left variable="FoundBase" type="bool"/>
      <Operator type="equals"/>
      <Right constant="true" type="bool"/>
    </Condition>
  </Transition>

  <Transition from="FindBaseLocation" to="Wander">
    <Condition>
      <Left variable="HasPuck" type="bool"/>
      <Operator type="equals"/>
      <Right constant="false" type="bool"/>
    </Condition>
  </Transition>

  <Transition from="GoToBase" to="FindBaseLocation">
    <Condition>
      <Left variable="FoundBase" type="bool"/>
      <Operator type="equals"/>
      <Right constant="false" type="bool"/>
    </Condition>
  </Transition>

  <Transition from="GoToBase" to="Wander">
    <Condition>
      <Left variable="HasPuck" type="bool"/>
      <Operator type="equals"/>
      <Right constant="false" type="bool"/>
    </Condition>
  </Transition>

  <Transition from="GoToBase" to="LeaveBase">
    <Condition>
```

```
        <Left variable="IsAtBase" type="bool"/>
        <Operator type="equals"/>
        <Right constant="true" type="bool"/>
      </Condition>
    </Transition>

    <Transition from="LeaveBase" to="Wander">
      <Operator type="and">
        <Condition>
          <Left variable="HasLeftBase" type="bool"/>
          <Operator type="equals"/>
          <Right constant="true" type="bool"/>
        </Condition>
        <Condition>
          <Left variable="IsAtBase" type="bool"/>
          <Operator type="equals"/>
          <Right constant="false" type="bool"/>
        </Condition>
      </Operator>
    </Transition>
  </FSM>
```

The code is equivalent to the following diagram.