

Proyecto IROBOT

Introducción a FlexBE – Octubre de 2019

Autor: Álvaro Fernández García

Contacto: fernandezgalvaro@uniovi.es

Fecha: 9 de octubre de 2019

La herramienta FlexBE ayuda a desarrollar máquinas de estados totalmente integrada en ROS. Además de proporcionar muchos estados básicos ya implementados (imprimir mensajes por pantalla, leer un mensaje de un tópico, esperar un cierto tiempo etc.) proporciona también una interfaz de usuario que facilita la creación y monitorización del autómata.

1 Acciones. Servidores y clientes

El concepto de acciones en ROS es muy importante a la hora de implementar máquinas de estado pues permite adaptar de manera rápida código en C++ para ser ejecutado en un estado de FlexBE, programado en Python.

Las acciones son una interfaz de comunicación implementada por el paquete *actionlib*[1]. Su funcionamiento es similar al de los servicios de ROS pero estos están pensados para operaciones breves mientras que las acciones presentan mejores herramientas para ejecuciones más largas y complejas.

Como se puede ver en la Figura 1, el esquema general es muy similar al de los servicios. La aplicación cliente realiza una llamada a la función con unos determinados parámetros. La ejecución de esta función se lleva a cabo en el servidor donde se realizan las operaciones pertinentes enviando a la aplicación cliente el resultado o resultados correspondientes.

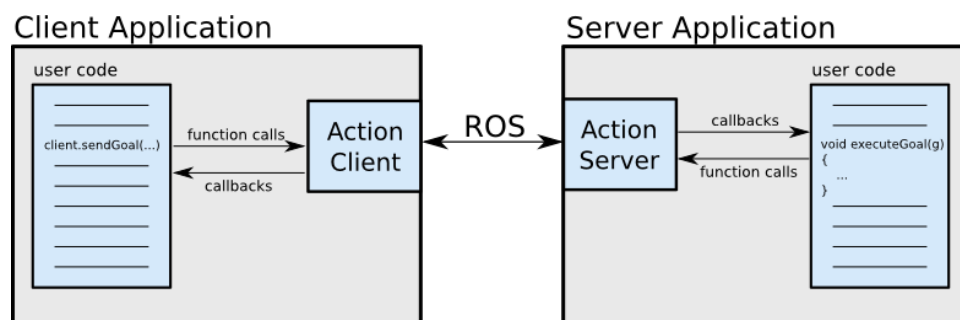
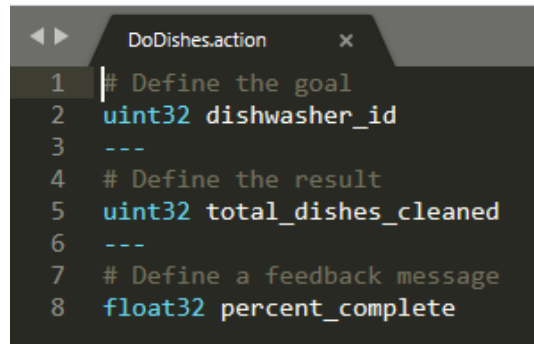


Figura 1: Esquema de funcionamiento de un cliente y un servidor que usan una acción. Figura extraída de [1]

Para establecer este tipo de comunicaciones en ROS serán necesarios tres archivos: el archivo *.action*, el servidor y el cliente.

1.1 Archivo *.action*

El archivo *action* es un archivo similar a los *.msg* o *.srv*. En él se define el *goal*, u objetivo, el *result*, o resultado tras haber realizado la acción, y el *feedback*, o la información que se puede enviar al cliente antes de la finalización de la acción.



```

1 | # Define the goal
2 | uint32 dishwasher_id
3 | ---
4 | # Define the result
5 | uint32 total_dishes_cleaned
6 | ---
7 | # Define a feedback message
8 | float32 percent_complete

```

Figura 2: Ejemplo de archivo *.action*

En la Figura 2 puede verse un ejemplo de archivo de acción. En este caso, el parámetro de llamada a la función será la *id* del lavaplatos. Durante la ejecución de la función se irá informando del porcentaje de platos lavados. Finalizada la acción se devolverá el número de platos lavados.

Los campos de la acción deben declararse en el orden descrito y separados por ---. En el ejemplo se presenta una variable para para cada campo, pero pueden componerse por múltiples variables o dejarse vacíos.

A partir de estos archivos simples, ROS es capaz de generar estructuras de clases C++ y mensajes para cada acción. Siguiendo con el ejemplo de *DoDishes.action*, para compilar la acción se debe añadir, antes de la función *catkin_package()*, en el *CMakeList.txt* del paquete en el que se encuentra:

```

find_package(catkin REQUIRED genmsg actionlib_msgs)
add_action_files(DIRECTORY action FILES DoDishes.action)
generate_messages(DEPENDENCIES actionlib_msgs)

```

En el archivo *package.xml* se deben añadir también las siguientes dependencias:

```

<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib_msgs</exec_depend>

```

1.2 Servidor

El servidor es el programa que deberá estar corriendo en paralelo en todo momento y ejecutará la acción cuando se lo ordene otro nodo.

Un ejemplo de este tipo de programas puede encontrarse en la Figura 3. Como puede verse contiene un *callback* (línea 6), en el que se ejecutaría la acción así. En el *main* se lanza el servidor y se mantiene un bucle de ejecución a la espera de que se realice una llamada

```

1 #include <chores/DoDishesAction.h> // Note: "Action" is appended
2 #include <actionlib/server/simple_action_server.h>
3
4 typedef actionlib::SimpleActionServer<chores::DoDishesAction> Server;
5
6 void execute(const chores::DoDishesGoalConstPtr& goal, Server* as) // Note: "Action" is not appended to DoDishes here
7 {
8     // Do lots of awesome groundbreaking robot stuff here
9     as->setSucceeded();
10 }
11
12 int main(int argc, char** argv)
13 {
14     ros::init(argc, argv, "do_dishes_server");
15     ros::NodeHandle n;
16     Server server(n, "do_dishes", boost::bind(&execute, _1, &server), false);
17     server.start();
18     ros::spin();
19     return 0;
20 }

```

Figura 3: Ejemplo de servidor de la acción *DoDishes*, en C++

1.3 Cliente

Por último, el cliente es el encargado de enviar la orden para que se ejecute la acción. En este caso, como lo integraremos dentro de un estado *FlexBE*, se mostrará un ejemplo en lenguaje *Python*. Basta con declarar un cliente y pasarle los correspondientes parámetros, es decir el o los *goals* que debe alcanzar la acción. Es en este mismo código en el que se espera a la finalización de la acción y puede leerse su resultado, o el *feedback* si esta aún no ha terminado.

Un ejemplo de estado *FlexBE* en el que se implementa un cliente para acciones se puede encontrar la sección 4.

```

1 #!/usr/bin/env python
2
3 import roslib
4 roslib.load_manifest('my_pkg_name')
5 import rospy
6 import actionlib
7
8 from chores.msg import DoDishesAction, DoDishesGoal
9
10 if __name__ == '__main__':
11     rospy.init_node('do_dishes_client')
12     client = actionlib.SimpleActionClient('do_dishes', DoDishesAction)
13     client.wait_for_server()
14
15     goal = DoDishesGoal()
16     # Fill in the goal here
17     client.send_goal(goal)
18     client.wait_for_result(rospy.Duration.from_sec(5.0))

```

Figura 4: Ejemplo de cliente de la acción *DoDishes*, en Python

Los temas tratados en esta sección pueden encontrarse, de manera más detallada, en la documentación del paquete *actionlib* [1].

2 Instalación

Para la instalación lo más cómodo resulta seguir los pasos descritos en la página de *FlexBE* [2]. Aunque existen disponibles más paquetes que implementan estados extra, para el trabajo que he realizado me ha bastado con la instalación de los dos metapaquetes más básicos. Debajo de este párrafo se puede ver las carpetas contenidas por los mismos.

- flexbe_behavior_engine:

behaviors

flexbe_core

flexbe_input

flexbe_mirror
flexbe_states

flexbe_msgs
flexbe_testing

flexbe_onboard
flexbe_widget

- generic_flexbe_states:

flexbe_manipulation_states flexbe_navigation_states flexbe_utility_states

Una vez se tiene FlexBE el siguiente paso es instalar FlexBEApp que proporcionará la interfaz gráfica. Las instrucciones también se encuentran en la referencia anteriormente aportada.

El siguiente paso puede ser crear carpeta que almacenará nuestro código (los estados implementados y la máquina de estados en sí). Para ello basta con desplazarse hasta la carpeta deseada en el terminal y ejecutar:

```
roslaunch flexbe_widget create_repo nombre_de_la_maquina(_behaviors)
```

Una vez hecho esto el último paso se debe acometer la configuración. Para ello se debe abrir la FlexBEApp y acudir a la pestaña *Configuration*. Una vez en ella, en el recuadro *StateLibrary* se debe especificar la ruta en la que se encuentra el código de los estados. Por comodidad, yo he decidido extraer el contenido de los paquetes *flexbe_behavior_engine* y *generic_flexbe_states* y guardarlos en un nuevo metapackage llamado *flexbe_things*, Figura 5.



Figura 5: Contenido de mi carpeta *flexbe_things*.

En el recuadro *Workspace* se deben incluir las rutas de la carpeta *behaviors* y *flexbe_behaviors* que se han creado automáticamente en nuestra carpeta personal. En la Figura 6 se puede observar el aspecto de la pestaña de configuración tras estos pasos.

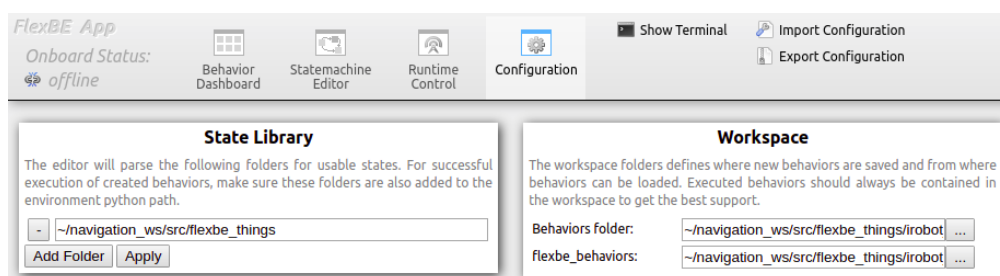


Figura 6: Configuración de la pestaña *Configuration*.

3 Interfaz FlexBeApp

En la siguiente sección se intentará introducir al manejo de la interfaz de la FlexBEApp intentando destacar lo más importante de su manejo.

3.1 Behavior Dashboard

Una vez instalado y configurado se puede pasar a crear el primer autómata de estados o *behavior*, para lo que será necesario visitar las tres pestañas restantes de la aplicación. Se comenzará por la pestaña *Behavior Dashboard*.

El primer paso será darle a nuestro nuevo *behavior* un nombre, usando una nomenclatura válida para código Python, y una descripción, unos tags y el nombre del autor, con fines de documentación.

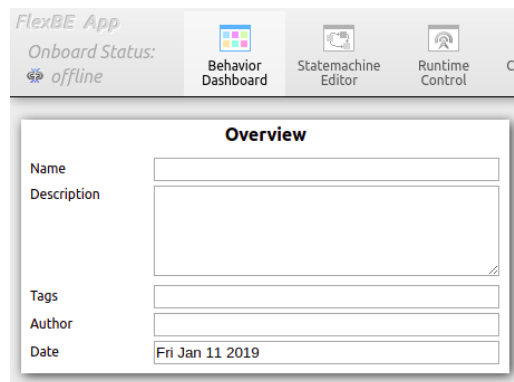


Figura 7: Pestaña *Overview* en la que se da nombre y se documenta nuestra máquina de estados.

Una vez descrito el *behavior*, la pestaña *Private Configuration* permite crear valores constantes que podremos pasar luego como entradas a nuestro estado. En *State Machine Userdata* se pueden declarar variables que, en este caso, sí podrán ser modificadas en los estados y solo recibirán un valor por defecto.

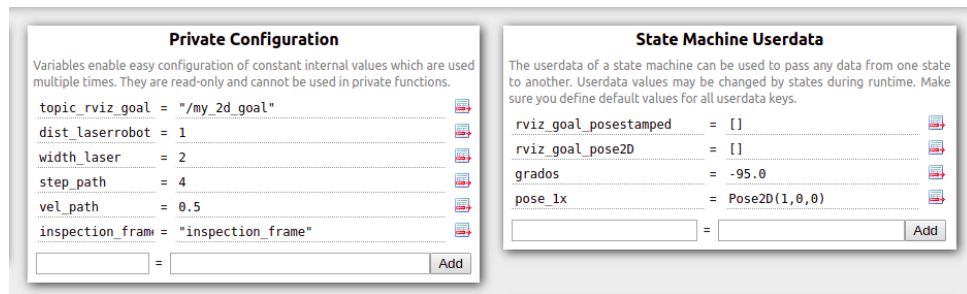


Figura 8: Pestañas *Private Configuration* y *State Machine Userdata*

Nota:

Si se desea añadir una variable que pertenece a un paquete, por ejemplo *geometry_msgs::Pose2D* como en la Figura 9, deberemos acudir al código Python de la máquina de estados situado en:

nombre_carpeta_behaviors\

behaviors\behavior_nombre_maquina_estados\src\behavior_nombre_maquina_estados\nombre_maquina_estados_sm.py


E importar el tipo de mensaje:

```

21 from flexbe_states.log_key_state import LogKeyState
22 from irobot_flexbe_states.getsimpleinspectionposes_state import GetSimpleInspectionPoses_State
23 from irobot_flexbe_states.moverobotcoord_action_state import MoveRobotCoordActionState
24 # Additional imports can be added inside the following tags
25 # [MANUAL_IMPORT]
26 from geometry_msgs.msg import Pose2D
27 # [/MANUAL_IMPORT]
28

```

Figura 9: Inclusión del tipo de mensaje *geometry_msgs::Pose2D*

El último tipo de variable son los parámetros. En el recuadro *Behavior Parameters*, Figura 10, se pueden añadir variables que serán fácilmente modificables por el operario en la pestaña de *Runtime Control*. Se pueden configurar los límites, el valor por defecto y otros aspectos del parámetro haciendo clic en el  junto a su definición, Figura 11.

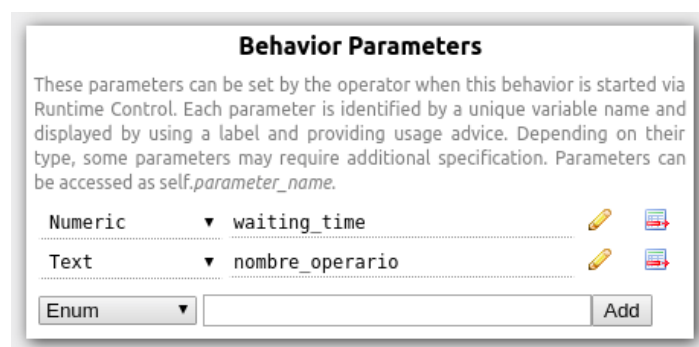


Figura 10: Recuadro de Behavior Parameters

Behavior Parameters

Numeric Label: **Waiting Time**

Advice for the operator: **Sets the time to wait after logging.**

waiting_time = **3**

Minimum: **1** Maximum: **5**

Figura 11: Configuración del parámetro waiting_time

Por último, en la parte inferior derecha encontramos el recuadro *State Machine Interface*. En él podemos definir las salidas de nuestra máquina de estado que finalizarán su ejecución. También es posible definir variables de entrada y de salida pues el software permite la opción de implementar superestados a partir de autómatas ya creados.

State Machine Interface

Defines how the state machine of this behavior can be accessed when embedded in another behavior.

Outcomes **finished** **failed**

Input Keys

Output Keys

Add

Figura 12: State Machine Interface

State Machine Editor

Esta pestaña es quizás la que más ventajas supone respecto a crear un autómata sin interfaz gráfica. Será en esta pantalla en la que podremos insertar estados en nuestro autómata, conectarlos entre sí mediante sus salidas y organizar la transmisión de datos y variables. Muchas funcionalidades básicas ya están implementadas en estados, pero otras deberán ser programadas por cada usuario.

El editor dispone de una herramienta para copiar, cortar y pegar y para deshacer y rehacer que puede desplegarse presionando **CTRL+ESPACIO**, Figura 13.

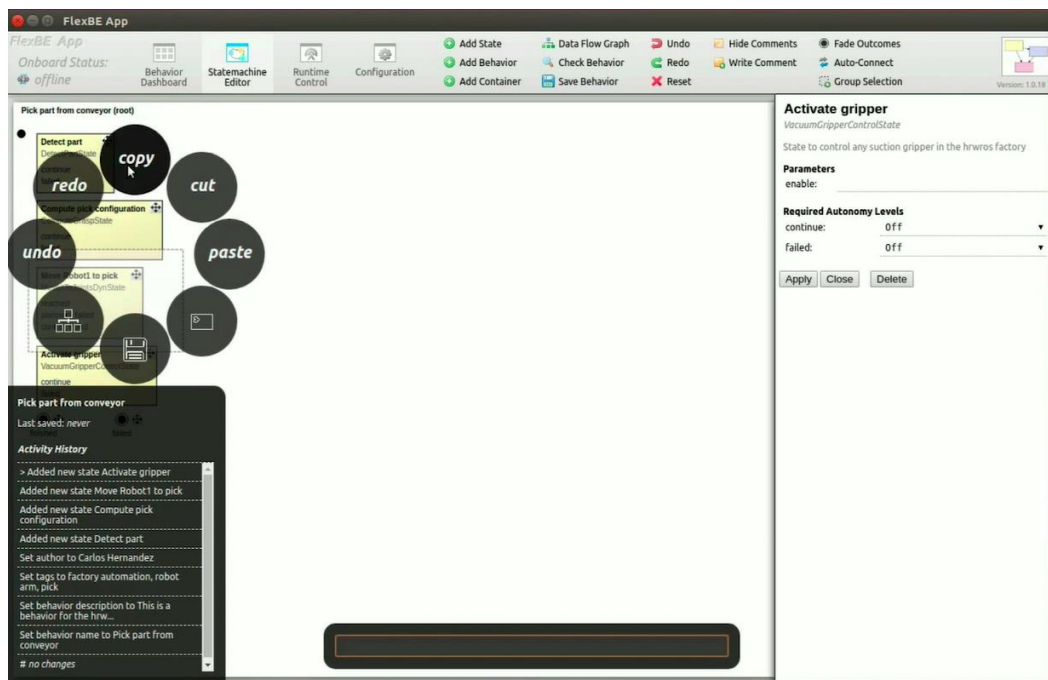


Figura 13: Herramienta desplegada en el editor de máquinas de estado

Nota:

Conviene destacar que la máquina de estados no permite guardar el progreso realizado hasta que su compilación es correcta. Por este motivo, resulta adecuado no realizar grandes avances sin haber ido guardando.

Además de estados también podremos añadir lo que FlexBE denomina como *containers*. Estos pueden ser de tres tipos:

- **StateMachine**
Funciona como el superestado clásico. Dentro de él pueden añadirse estados e incluso un nuevo contenedor.
- **Concurrency**
Estado que permite la ejecución simultánea de varios estados y que este no se abandone hasta que todos los estados en paralelo no hayan finalizado, si así se desea. De momento, no ha sido necesaria su uso, pero se ha probado una implementación simple y parecen funcionar. Puede ser útil cuando se realizan dos actividades a la vez, por ejemplo, en nuestro proyecto, recorrer la chapa e inspeccionarla registrando un mapa de defectos. Documentación sobre este tipo de contenedores puede encontrarse en [3].
- **Priority**
Superestado especial pues goza de prioridad respecto al resto del autómatas. Cuando este se activa la ejecución del resto de estados se detiene.

El editor también permite que se agreguen *behaviors* y que funcionen como *containers StateMachine*.

3.2 RunTimeControl

En esta pestaña se lanza el código creado y permite además la supervisión de la ejecución y la interacción con las transiciones entre los estados. Para ello, no es suficiente con abrir la FlexBEapp normal habrá que hacerlo con el comando

```
roslaunch flexbe_app flexbe_full.launch
```

y esperar a que se conecte con ROS, Figura 14.

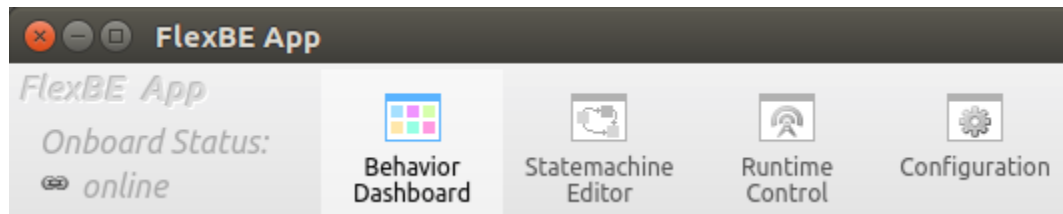


Figura 14: Interfaz de la FlexBEApp cuando ha establecido conexión con ROS.

En el editor, previamente, puede establecerse para cada estado el nivel de interactividad de sus salidas, Figura 15. Se puede definir la cota de autonomía de nuestra máquina en esta pestaña de ejecución, Figura 16. Una vez establecida la cota en la pestaña de ejecución, todas las transacciones cuyo nivel sea inferior a ella deberán ser activados de manera manual para el usuario para llevarse a cabo. La interfaz informa, en cualquier caso, de si la salida habría sido activada por el estado o no.

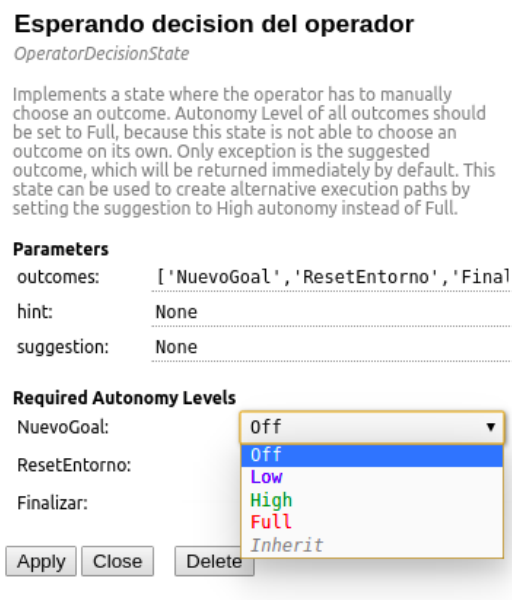


Figura 15: Selección del nivel de autonomía de cada salida en el StateMachineEditor

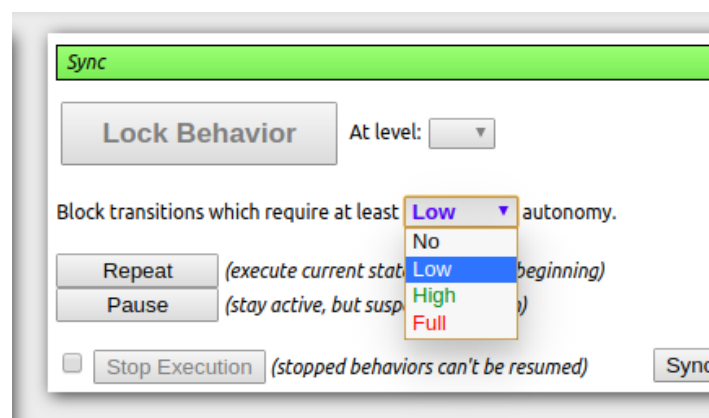


Figura 16: Selección de la cota mínima de autonomía para no bloquear las transiciones en la pestaña RunTimeControl

De manera independiente, cualquier transición puede forzarse en cualquier momento desde esta pestaña. Estas características permiten en cierta medida establecer una interfaz de usuario. Esto es aprovechado por estados ya implementados como, por ejemplo, el *OperatorDecisionState* que permite escoger entre las salidas que se definan en el editor, presente en la Figura 15.

Nota:

Errores en el código de los estados pueden dar error al ejecutar el autómata a pesar de que el comportamiento había compilado sin problemas. En este caso es necesario cerrar la aplicación, modificar el archivo `.py`, y lanzar el siguiente comando en la ruta del código antes de reabrir FlexBEApp:

```
chmod +x mi_archivo.py
```

De lo contrario, la máquina seguirá funcionando con la versión del estado con la que se mandó ejecutar.

También se debe tener en cuenta que para el correcto funcionamiento de estados que hacen uso

4 Clase EventState

Por último, se realizará un pequeño seguimiento de las funciones presentes en las clases que implementan los estados de FlexBE, que deben heredar de *EventState*. Para ello se analizará el *example_action_state.py*, que se añade automáticamente en la carpeta `xxxx_flexbe_states` de nuestro código, ligeramente modificado.

Tanto este ejemplo, como *example_state.py* y los estados incluidos en los paquetes FlexBE, se encuentran bien documentados son una buena manera de familiarizarse con la programación de estas clases.

Más información sobre el ciclo de vida de los estados FlexBE puede encontrarse en [4].

Se deben incluir las librerías necesarias para el correcto funcionamiento, como puede verse en la Figura 17. *EventState* y *Logger* son imprescindibles para el funcionamiento de cualquier estado FlexBE. Por su parte, el comentario

```
#!/usr/bin/env Python
```

es necesario para la ejecución de cualquier código Python en ROS.

En la declaración de la clase se le otorgará nombre y se hará que herede de *EventState*. A continuación, es habitual documentar el estado para reconocerlo en la FlexBEApp y hacer el código más amigable para otros usuarios. La nomenclatura empleada para identificar las entradas y salidas del estado es la que puede verse en la Tabla 1.

```

1  #!/usr/bin/env python
2  from flexbe_core import EventState, Logger
3  from flexbe_core.proxy import ProxyActionClient
4
5  # example import of required action
6  from chores.msg import DoDishesAction, DoDishesGoal
7
8
9  class ExampleActionState(EventState):
10     """
11     Actionlib actions are the most common basis for state implementations
12     since they provide a non-blocking, high-level interface for robot capabilities.
13     The example is based on the DoDishes-example of actionlib (see http://wiki.ros.org/actionlib).
14     This time we have input and output keys in order to specify the goal and possibly further evaluate the result in a later state.
15
16     -- dishes_to_do int    Expected amount of dishes to be cleaned.
17
18     ># dishwasher int    ID of the dishwasher to be used.
19
20     #> cleaned int        Amount of cleaned dishes.
21
22     <= cleaned_some       Only a few dishes have been cleaned.
23     <= cleaned_enough     Cleaned a lot of dishes.
24     <= command_error      Cannot send the action goal.
25
26     """

```

Figura 17: Inclusión de librerías, declaración de la clase y documentación de la misma.

Parámetros	--
Variables de entrada	>#
Variables de salida	#>
Salidas del estado	<=

Tabla 1: Nomenclatura empleada en la documentación de los estados

A continuación, en el constructor `_init_` se declaran las entradas y salidas recién comentadas. Los parámetros se deben indicar como un argumento del constructor (línea 28), mientras que el resto de las variables se declaran en el superconstructor (línea 30), Figura 18.

```

28  def __init__(self, dishes_to_do):
29      # See example_state.py for basic explanations.
30      super(ExampleActionState, self).__init__(outcomes = ['cleaned_some', 'cleaned_enough', 'command_error'],
31                                              input_keys = ['dishwasher'],
32                                              output_keys = ['cleaned'])
33
34      self.dishes_to_do = dishes_to_do
35
36      # Create the action client when building the behavior.
37      # This will cause the behavior to wait for the client before starting execution
38      # and will trigger a timeout error if it is not available.
39      # Using the proxy client provides asynchronous access to the result and status
40      # and makes sure only one client is used, no matter how often this state is used in a behavior.
41      self._topic = 'do_dishes'
42      self._client = ProxyActionClient({self._topic: DoDishesAction}) # pass required clients as dict (topic: type)
43
44      # It may happen that the action client fails to send the action goal.
45      self._error = False
46

```

Figura 18: Función `_init_` en `example_action_state.py`

Es en esta función `_init_` donde podrán definirse atributos que luego podrán emplearse en el resto de los métodos como es el caso del cliente de la acción (línea 42). También es una práctica habitual y recomendable crear una variable para los parámetros o variables de entrada por si el valor original no quiere modificarse por error.

La función `execute`, Figura 19, se ejecuta de manera cíclica mientras el estado se encuentre activo. En el ejemplo de la figura se chequea si la acción ha concluido o a dado error. Cuando concluye es este el método que asocia a las salidas los valores correspondientes (línea 61) y activa la salida del estado (línea 65), rompiendo el bucle de ejecución.

```

48 def execute(self, userdata):
49     # While this state is active, check if the action has been finished and evaluate the result.
50
51     # Check if the client failed to send the goal.
52     if self._error:
53         return 'command_error'
54
55     # Check if the action has been finished
56     if self._client.has_result(self._topic):
57         result = self._client.get_result(self._topic)
58         dishes_cleaned = result.total_dishes_cleaned
59
60         # In this example, we also provide the amount of cleaned dishes as output key.
61         userdata.cleaned = dishes_cleaned
62
63         # Based on the result, decide which outcome to trigger.
64         if dishes_cleaned > self._dishes_to_do:
65             return 'cleaned_enough'
66         else:
67             return 'cleaned_some'
68
69     # If the action has not yet finished, no outcome will be returned and the state stays active.

```

Figura 19: Función `execute` en `example_action_state.py`

Previamente al bucle de `execute`, se ha ejecutado, aunque en este caso una única vez, el código contenido en `on_enter`, Figura 20. En este ejemplo, se emplea para leer la variable de entrada (línea 77) y para realizar la llamada a la acción, cuyo avance se comprueba en `execute`.

```

72 def on_enter(self, userdata):
73     # When entering this state, we send the action goal once to let the robot start its work.
74
75     # As documented above, we get the specification of which dishwasher to use as input key.
76     # This enables a previous state to make this decision during runtime and provide the ID as its own output key.
77     dishwasher_id = userdata.dishwasher
78
79     # Create the goal.
80     goal = DoDishesGoal()
81     goal.dishwasher_id = dishwasher_id
82
83     # Send the goal.
84     self._error = False # make sure to reset the error state since a previous state execution might have failed
85     try:
86         self._client.send_goal(self._topic, goal)
87     except Exception as e:
88         # Since a state failure not necessarily causes a behavior failure, it is recommended to only print warnings, not errors.
89         # Using a linebreak before appending the error log enables the operator to collapse details in the GUI.
90         logger.logwarn('Failed to send the DoDishes command:\n%s' % str(e))
91     self._error = True

```

Figura 20: Función `on_enter` en `example_action_state.py`

El método `on_exit` se lleva a cabo también una sola vez, una vez se ha activado una de las salidas y se ha pasado a un nuevo estado. En el ejemplo, Figura 21, se emplea para asegurarse de que la acción deja de ejecutarse.

```

94 def on_exit(self, userdata):
95     # Make sure that the action is not running when leaving this state.
96     # A situation where the action would still be active is for example when the operator manually triggers an outcome.
97
98     if not self._client.has_result(self._topic):
99         self._client.cancel(self._topic)
100         logger.loginfo('Cancelled active action goal.')

```

Figura 21: Función `on_exit` en `example_action_state.py`

Por último, y aunque estos no se encuentran definidos en `example_action_state` (y por tanto no se tienen en cuenta), cabe mencionar la existencia de `on_start` y `on_stop`, Figura 22. El primer método permite realizar alguna acción cuando se inicia la máquina de estados, independientemente de si el estado se encuentra activo o no. El segundo tiene un comportamiento análogo, pero este entra en funcionamiento cuando la máquina de estados para su ejecución.

```

def on_start(self):
    # This method is called when the behavior is started.
    # If possible, it is generally better to initialize used resources in the constructor
    # because if anything failed, the behavior would not even be started.

    # In this example, we use this event to set the correct start time.
    self._start_time = rospy.Time.now()

def on_stop(self):
    # This method is called whenever the behavior stops execution, also if it is cancelled.
    # Use this event to clean up things like claimed resources.

```

Figura 22: Funciones on_start y on_enter extraídas de example_state.py

5 Referencias

- [1] «actionlib - ROS Wiki». [En línea]. Disponible en: <http://wiki.ros.org/actionlib>. [Accedido: 11-ene-2019].
- [2] «FlexBE Behavior Engine». [En línea]. Disponible en: <http://philserver.bplaced.net/fbe/download.php#states>. [Accedido: 14-ene-2019].
- [3] «flexbe/Tutorials/Parallel State Execution - ROS Wiki». [En línea]. Disponible en: <http://wiki.ros.org/flexbe/Tutorials/Parallel%20State%20Execution>. [Accedido: 16-ene-2019].
- [4] «flexbe/Tutorials/The State Lifecycle - ROS Wiki». [En línea]. Disponible en: <http://wiki.ros.org/flexbe/Tutorials/The%20State%20Lifecycle>. [Accedido: 15-ene-2019].