Proyecto IROBOT

Navegación autónoma, generación de trayectorias y coordinación de comportamientos

Manual de usuario – Octubre de 2019

Autor: Álvaro Fernández García

Contacto: fernandezgalvaro@uniovi.es

Fecha: 9 de octubre de 2019

1 Resumen del conjunto de workspaces de ROS

El entorno ROS se organiza principalmente mediante espacios de trabajo o *workspaces*. Estos contendrán el código fuente organizado en paquetes, para más información consultar el anexo dedicado a conceptos básicos de ROS, y, una vez compilado, los ejecutables y librerías generados.

El resultado final del trabajo ha sido, por comodidad, la creación de dos espacios de trabajo: *ws* e *irobot_qt_ws*. El primero de ellos engloba principalmente las rutinas de navegación autónoma y el segundo, más orientado a poder ser editado, compilado y ejecutado con la ayuda de QtCreator, los de generación de trayectorias.

En este apartado se dará una breve descripción de los paquetes, nodos y librerías más reseñables del proyecto.

1.1 Espacio de trabajo ws

El workspace ws está compuesto por los siguientes paquetes:

1.1.1 action_server_package

Este paquete contiene los servicios de creación propia a los que se llama durante el comportamiento de navegación.

1.1.2 aux_controllers

Paquete que contiene librerías orientadas, principalmente dedicadas al envío de mensajes al robot y al acople de herramientas en la simulación.

1.1.3 flexbe_ap, flexbe_behavior_engine y generic_flexbe_states

Estos paquetes son los encargados de hacer funcionar la herramienta de coordinación FlexBe. Para su descarga e instalación puede consultarse su página web REF.

1.1.4 irobot_move_base

Este paquete contiene los archivos de configuración .yaml que ajustan la navegación autónoma del robot.

1.1.5 irobot_sm_behaviors

Contiene el código generado por FlexBE derivado de la creación de la máquina de estados y los estados de creación propia en Python.

1.1.6 my_rviz_tools

Este paquete contiene las librerías creadas para nuestras propias herramientas en Rviz. De manera resumida, *my_goal_tool* implementa la herramienta que permite clicar en el mapa de Rviz y publicar esa posición en el topic *my_2d_goal* que puede leerse desde cualquier nodo. La clase *reparation_tool* tiene una funcionalidad similar publicando en el *topic* rviz_defect, para una funcionalidad temporal que permitía simular de manera muy básica una reparación.

1.1.7 slam_launchers

Paquete que contiene. launch para ejecutar de manera conjunta y con los adecuados parámetros el mapeo 3D y el algortimo de SLAM.

1.1.8 spawn_package

Paquete que múltiples. launch empleados de manera interna para invocar modelos en distintos mundos de Gazebo.

1.1.9 summitxl_metapackage

Paquete que contiene los paquetes necesarios para el uso del SummitXL en el simulador Gazebo. Se han incluido archivos adicionales como un modelo que contiene el sensor RGBD.

1.1.10 velodyne_simulator

Este paquete, disponible en [1], contiene los archivos necesarios para la simulación del sensor 3D. Los cambios en el código original se encuentran en las carpetas velodyne_description/urdf y velodyne_description/launch. En el primero se encuentran los parámetros que permiten modificar el sensor 3D (número de láseres, rango, apertura, etc.). En esta carpeta pueden encontrarse simulaciones de sensores comerciales como el MRS1000, MRS6000, IntelRealSense_D435 etc.

Usando como referencia ARCHIVO, Los parámetros de interés a modificar en los urdf pueden encontrarse en la línea 4. En ella se puede modificar el nombre del *topic* en el que publica, el número de láseres, la frecuencia, el ruido y el rango mínimo y máximo.

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="VLP-16">
3    <xacro:property name="M_PI" value="3.1415926535897931" />
4    <xacro:macro    name="VLP-16"    params="*origin    parent:=base_link
   name:=velodyne  topic:=/velodyne_points  hz:=10  lasers:=16  samples:=1875
   collision_range:=0.3   min_range:=0.9   max_range:=130.0   noise:=0.008
   min_angle:=-${M_PI} max_angle:=${M_PI} gpu:=false">
5
```

En la parte baja del archivo se puede modificar los ángulos máximos y mínimos tanto horizontales (línea 110 y 111) como verticales (línea 116 y 117).

```
100       <xacro:unless value="${gpu}">
101         <sensor type="ray" name="${name}-VLP16">
102           <pose>0 0 0 0 0 0</pose>
103           <visualize>false</visualize>
104           <update_rate>${hz}</update_rate>
105           <ray>
106             <scan>
107               <horizontal>
108                 <samples>${samples}</samples>
109                 <resolution>1</resolution>
110                 <min_angle>${min_angle}</min_angle>
111                 <max_angle>${max_angle}</max_angle>
112               </horizontal>
113               <vertical>
114                 <samples>${lasers}</samples>
115                 <resolution>1</resolution>
116                 <min_angle>-${15.0*M_PI/180.0}</min_angle>
117                 <max_angle> ${15.0*M_PI/180.0}</max_angle>
118               </vertical>
119             </scan>
120             <range>
121               <min>${collision_range}</min>
```

```
122                <max>${max_range+1}</max>
123                <resolution>0.001</resolution>
124              </range>
125              <noise>
126                <type>gaussian</type>
127                <mean>0.0</mean>
128                <stddev>0.0</stddev>
129              </noise>
130            </ray>
131   <plugin name="gazebo_ros_laser_controller"
      filename="libgazebo_ros_velodyne_laser.so">
132              <topicName>${topic}</topicName>
133              <frameName>${name}</frameName>
134              <min_range>${min_range}</min_range>
135              <max_range>${max_range}</max_range>
136              <gaussianNoise>${noise}</gaussianNoise>
137            </plugin>
138          </sensor>
139        </xacro:unless>
```

1.2 Espacio de trabajo irobot_qt_ws

1.2.1 action_server_package_qt

Paquete que contiene los servicios y servidores de las acciones de movimiento del robot y creación de *frames*.

1.2.2 gazebo_sim_movement

Paquete empleado para simular trayectorias en Gazebo sin envío de mensajes cinéticos.

1.2.3 irobot_fcpp_pckg

Contiene las librerías de descomposición trapezoidal, generación de trayectorias y otras funcionalidades auxiliares.

1.2.4 irobot_fcpp_simul

Paquete que contiene los servicios y servidores de las acciones de descomposición trapezoidal, cálculo de trayectorias y el servicio principal del autómata.

1.2.5 irobot_inspection_pckg

Paquete que contiene los servicios y servidores de las acciones de inspección.

2 Entorno de navegación autónoma

En este apartado se explicarán los pasos necesarios para poner en marcha el comportamiento de navegación autónoma.

2.1 Configuración de la navegación autónoma

Como ya se ha comentado los archivos configuración de la navegación autónoma se pueden encontrar en el paquete *irobot_move_base*.

Los parámetros para el planificador TEB se encuentran en el archivo *base_local_params.yaml.* Estos valores son leídos por el paquete instalado *teb_local_planner,* la explicación de los mas importantes puede encontrarse en la wiki de ROS [2]. El archivo para el MPO700 se basa en gran medida en este archivo presente en un repositorio oficial de Neobotix [3].

El resto de los archivos contienen los parámetros relativos al mapa de coste local, el global y el estático.

Para la navegación se utiliza la librería de ROS *navigation* [4]. Se trata del paquete más extendido en el control y la navegación de robots móviles. Se encargará de coordinar la información recibida del mapa y el planificador de trayectorias para enviar comandos a la base móvil. La explicación se puede dividir en las características de los mapas de coste y la configuración del planificador en sí.

Los mapas de coste son estructuras proporcionadas por la librería *costmap_2d* [5] que permiten trabajar con las zonas ocupadas, libres o desconocidas.

Los mapas de coste de este paquete pueden organizarse en distintas capas siendo las más comunes la capa del mapa estático, la capa de obstáculos y la capa de inflación. Una solución habitual es emplear un mapa estático para el mapa global y una capa de obstáculos construida de manera dinámica a partir de los sensores para el mapa local.

Gracias a *Octomap* el mapa recibido en nuestro trabajo ya tendrá un comportamiento dinámico y que garantiza al robot un entorno seguro para la navegación. De esta manera, el mapa local y global estará compuesta por los mismos tipos de capas: una capa *staticmap*[6] y, por encima, una capa *inflation*[7].

La capa estática será simplemente la lectura del mapa 3D proyectado configurada para atender a actualizaciones de este.

La capa de inflación, por su parte, permite aumentar la seguridad de la navegación dándole al mapa valores de peligro alrededor de los obstáculos detectados. En la implementación se establece un radio de inflación de 0.75 metros. Este valor es modificable en la última línea del archivo irobot_move_base/yaml/costmap_common_params.

La principal diferencia entre el mapa local y el global estará en su tamaño. El mapa global recibe un tamaño máximo de 100x100 metros (valor modificable en irobot_move_base/yaml/global_costmap), en este caso, y se irá completando según se va construyendo el mapa 3D. Servirá para planificar la trayectoria inicial completa. El mapa local, sin embargo, será una ventana de 6x6 metros (valor modificable en irobot_move_base/yaml/local_costmap),sobre el mapa que se irá desplazando junto al robot, Figura 1. Sobre este mapa se irán introduciendo las distintas poses intermedias de manera que respeten la trayectoria global lo máximo posible evitando la colisión con obstáculos.

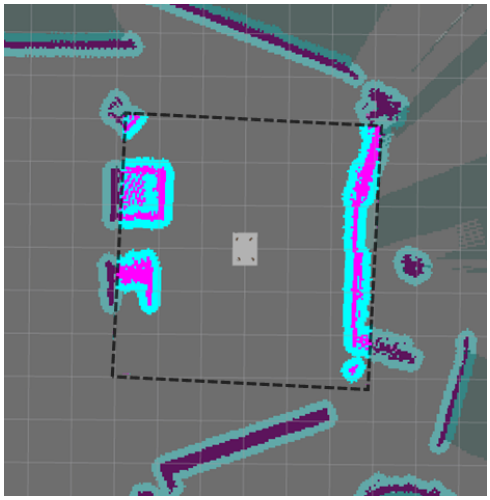

Figura 1: Representación del mapa de coste local. El mapa de coste global se extiende más y, en la imagen, puede verse con una menor opacidad.

2.2 Acciones y servidores

El código de FlexBE se encuentra escrito en lenguaje Python en contraposición con el C++ de los paquetes de desarrollo propio. Para poder aprovechar ese código se hace uso de la comunicación mediante acciones, explicada en el Anexo I. De esta manera, los estados de la máquina estarán escritos en Python y contendrán los clientes que realizan las llamadas a las acciones. El código C++ implementará los servidores que corren de manera constante y que ejecutará la función asociada a la acción cuando el cliente lo solicite.

Los servidores que ejecutan acciones implementados específicamente para la simulación de la navegación autónoma son los que se ven en la Tabla 1. Los movimientos del robot se realizan con los métodos de la clase *RobotController.*

Servidor - .action	**Descripción**
baserotation_server.cpp – BaseRotation.action	Ejecución de una rotación de la base un número especificado de grados.
moverobotcoord_server.cpp – MoveRobotCoord.action	Movimiento recto del robot hacia una posición referida en un *frame* especificado.
completerectanglecoverage_server.cpp – CompleteRectangleCoverage.action	Planificación y ejecución de la trayectoria compleja de inspección basada en los parámetros de la chapa y del robot.

Tabla 1: Conjunto de acciones empleadas en el autómata

Otras operaciones se realizan de manera mucho más rápida por lo que se implementarán mediante servicios en lugar de mediante acciones, Tabla 2. Es el caso de la publicación de un sistema de coordenadas auxiliar y la consulta sobre la posición de las chapas.

Servidor - .srv	Descripción
inspectionframe_server.cpp – SetFrame.srv	Publicación de un *frame* de *tf* a una distancia y orientación de un sistema de referencia. El servicio se implementa como el método de una clase por lo que se le dota de memoria respecto a unas cuantas variables.
chapastorage_server.cpp – ChapaStorage.srv	En función de la solicitud devuelve el número de chapas almacenadas o las dimensiones y las coordenadas de las esquinas de una lámina seleccionada.

Tabla 2: Conjunto de servicios empleadas en el autómata

2.3 Implementación de la máquina de estados y parámetros

Explicados ya los conceptos se pasará a detallar la máquina implementada. En primer lugar, es necesario comentar que los servidores de las acciones y servicios deberán estar ejecutándose en paralelo cuando se inicie la máquina de estados.

Además de estos programas de desarrollo propio también deberán estar en ejecución los nodos de los paquetes encargados del SLAM, el mapeo 3D y el de navegación, además de, obviamente, los que proporcionan la simulación del robot y los sensores.

Finalmente, antes de la ejecución de la máquina pueden ajustarse las variables que configuran su comportamiento, Tabla 3.

Nombre		Valor		Descripción
topic_rviz_goal	=	"/my_2d_goal"	<cte.>	*Topic* del que se recibe posiciones objetivo.
inspection_frame	=	"inspection_frame"	<cte.>	Sistema de coordenadas auxiliar para la inspección.
grados	=	180.0	<cte.>	Número de grados para la primera exploración.
dist_laserobot	=	1	<cte.>	Distancia entre la base del robot y haz de inspección.
width_laser	=	2	<cte.>	Amplitud del haz de inspección.
step_path	=	4	<cte.>	Distancia máxima entre dos posiciones intermedias de un recorrido de chapa.
vel_path	=	0.5	<cte.>	Velocidad para el recorrido encima de la chapa.
rviz_goal_posestamped	=	[ ]	<vble.>	Variable auxiliar que guarda un mensaje de tipo *PoseStamped.*

rviz_goal_pose2d = [ ]	<vble.> Variable auxiliar que guarda un mensaje de tipo *Pose2D.*

Tabla 3: Constantes y variables de la máquina en FlexBE

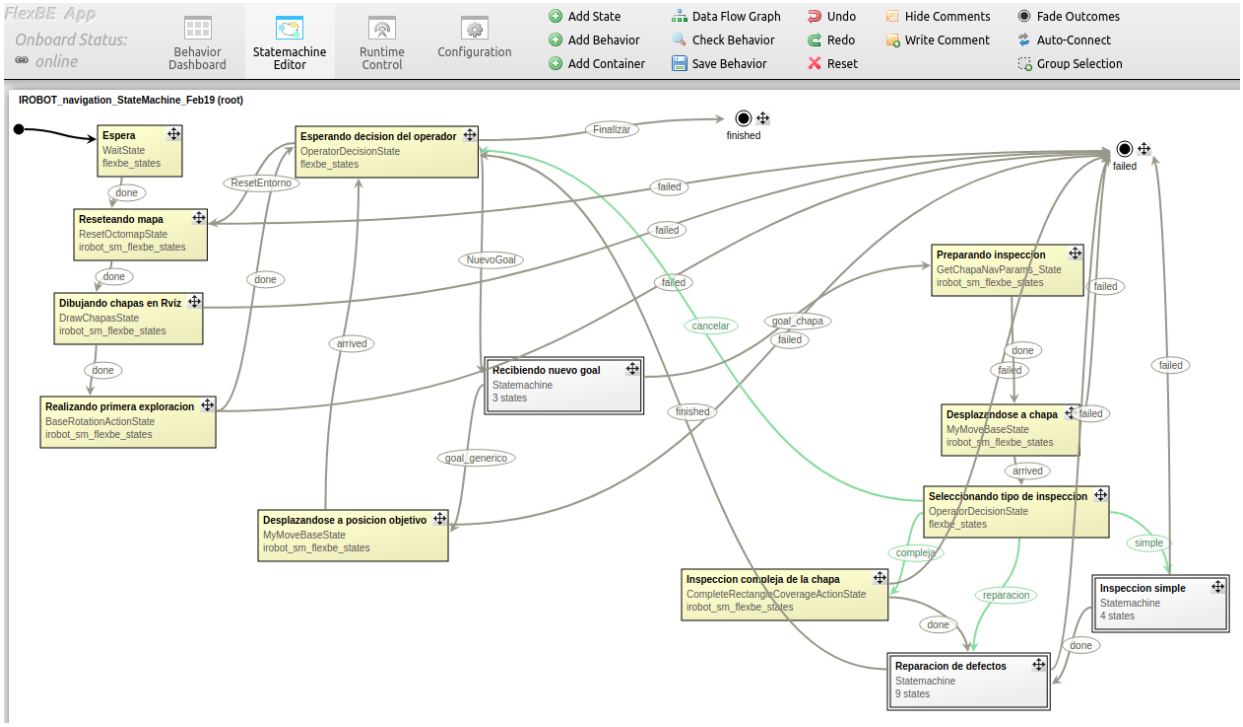

Figura 2: Vista general de la máquina de estados en FlexBE

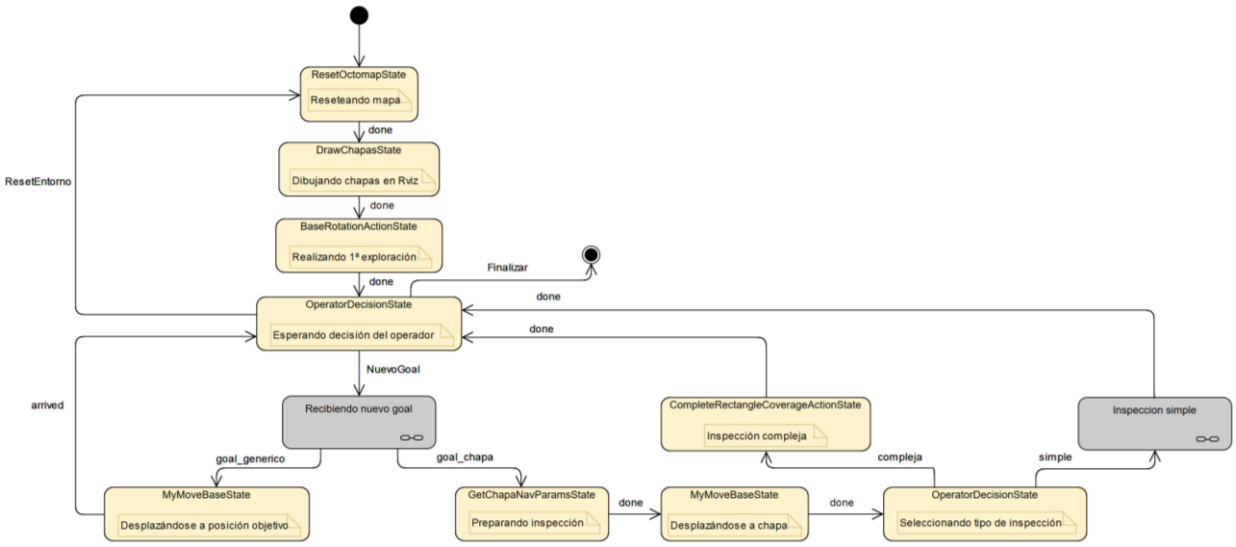

Figura 3: Vista general de una representación de la máquina de estados.

Centrándonos en la lógica implementada en el autómata, Figura 2 y Figura 3Figura 3: Vista general de una representación de la máquina de estados.

, el primer estado en activarse será el de la re-inicialización del *octomap.* Para el borrado de todos los nodos que componen la estructura de *octrees* se invoca el servicio *reset* del nodo *octomap_server.*

El siguiente estado se empleará para representar la posición de las chapas en *Rviz*. Para ello, se accede al servidor que guarda las coordenadas de las chapas y se publica un mensaje del tipo *geometry_msgs::PolygonStamped* [8] creando para cada una el *topic chapa_idX,* en función del identificador de la chapa dibujada.

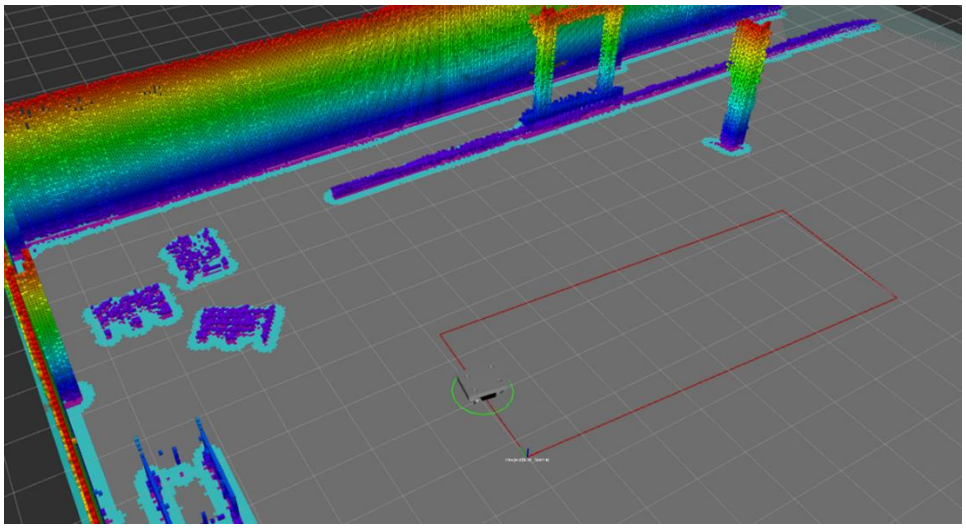

Figura 4: Imagen con robot en chapa dibujada

En la siguiente etapa se realizará una rotación de la base del robot sobre sí misma. De esta manera antes de inicializar el comportamiento el robot habrá obtenido un mapa completo de su entorno más cercano. Para realizar la rotación se hace una llamada a la acción del servidor *baserotation_server* en la que el número de grados viene definido por una variable configurada en la creación del autómata. La acción hace uso del método *turn_rel* de la clase *RobotController*.

Realizadas estas etapas iniciales se llega al *OperatorDecisionState,* una estancia desarrollada por *FlexBE* que permite al usuario seleccionar el siguiente paso. Se establecen tres posibles salidas: *Finalizar, ResetEntorno* y *NuevoGoal*. La primera permitirá finalizar el autómata y la segunda volver a los estados iniciales, por si se ha producido algún error en el mapa. Por último, *NuevoGoal* permitirá acceder a los estados relativos al envío de posiciones objetivo.

Esta última salida lleva al superestado *Recibiendo nuevo goal* compuesto internamente por tres estados que se ejecutan de manera secuencial, Figura 5. El primero de ellos es un *SubscriberState* de la propia librería *FlexBE* que espera a la publicación de un mensaje en un determinado *topic.* El estado se configura para leer el *topic* que publica la herramienta de selección de posiciones en el visualizador y para tener una variable de salida con la posición clicada. El siguiente estado, *PoseStamped_Pose2D_ConversionState*, sirve para cambiar el tipo de mensaje de *PoseStamped* a *Pose2D*. Esto permite continuar al estado que evalúa si la posición seleccionada se encuentra dentro o fuera de una chapa. La comprobación se realiza accediendo a las coordenadas almacenadas en el servicio de *chapastorage_server.* El estado tiene dos salidas que comparte con el superestado: *goal_generico*, en el caso de que la posición no esté dentro de una lámina de acero y *goal_chapa* en el caso de que sí.

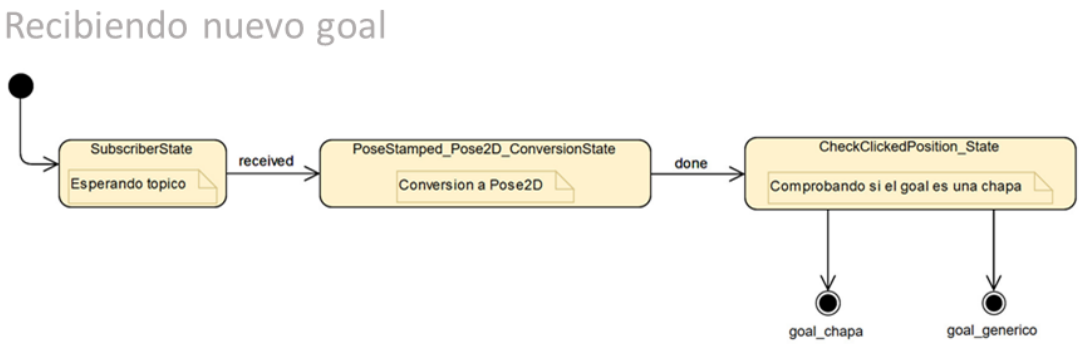

Figura 5: Interior del superestado "Recibiendo nuevo goal"

En el caso de haberse seleccionado una posición cualquiera, el autómata avanzará hacia el estado *Desplazándose a posición objetivo.* Este está implementado por *MyMoveBaseState*, que supone una pequeña variación respecto a *MoveBaseState* ya incluido en la librería. La modificación principal hace referencia al *frame* de referencia que viene por defecto. El autómata se mantendrá en este estado hasta que el robot alcance la posición deseada y se active la salida *arrived* volviendo al estado controlado por la decisión del operador.

Si la pose clicada se encuentra dentro de una chapa se accede al estado que prepara la inspección. Esta toma como entrada la *id* de la chapa que proporciona el anterior superestado. Con esa información *GetChapaNavParamsState* puede obtener del servicio las coordenadas y dimensiones de las hojas de acero. Con ello, puede variar la posición del *inspection_frame* a partir del servicio *SetFrame* que se ejecuta como un método de la clase *FrameService*. El estado también es capaz de seleccionar la pose intermedia entre las esquinas *A* y *B* de la chapa. Esta pose se envía a un nuevo estado que emplea el paquete *move_base* para que se lleve a cabo el desplazamiento.

Una vez el robot está correctamente situado en una posición inicial para la inspección se llega a un nuevo estado en el que las posibles salidas son cancelar la inspección, una inspección compleja y una inspección simple de la chapa. La opción de la cancelación hace que se vuelva, una vez más, al estado *"Esperando decisión del operador"* en el caso de que no se desee realizar ninguna inspección.

La inspección compleja se implementa en un estado que recibe las dimensiones de la chapa a inspeccionar. Con esos parámetros, junto a los que componen la configuración del robot se puede llamar a la acción *CompleteRectangleCoverage.* Esta acción incluye la planificación de las distintas posiciones a alcanzar y el envío de comandos al robot para que los realice. Una vez finalizada la inspección se retorna al estado en el que se decide la siguiente operación.

En el caso de que se seleccione una inspección simple se accederá al contenedor *Inspección simple*, compuesto por cuatro estados secuenciales, Figura 6. El primero de ellos permite obtener las tres poses necesarias para esta inspección: una pose a cierta distancia de la chapa para cubrirla al completo, una vuelta a una posición entre las coordenadas *A* y *B* y, por último, una posición entre las coordenadas *C* y *D* que finalizará la inspección como se representa en la **¡Error! No se encuentra el origen de la referencia.**. Los siguientes estados se encargan de enviar comandos simples al robot mediante la acción *MoveRobotCoord*. Para ello la acción recibe como objetivos el vector que se ejecuta por una función *move* de la clase *RobotController.* Cuando esta inspección finaliza se retorna al estado de decisión del operador.

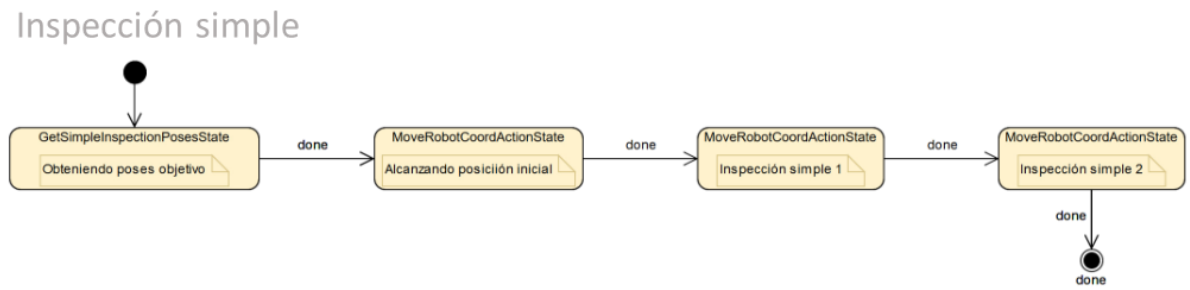

Figura 6: Interior del superestado “Inspección simple”

Finalizada la explicación cabe comentar que la inmensa mayoría de los estados cuentan con una salida *failed* que desembocan en el final homónimo del autómata y que se activa si algún elemento en el estado desencadena en algún tipo de error. Se ha obviado en la explicación general para mayor claridad.

Por último, en las Figura 7, Figura 8, y Figura 9 se pueden ver un diagrama del envío de datos a través de la máquina de estados.

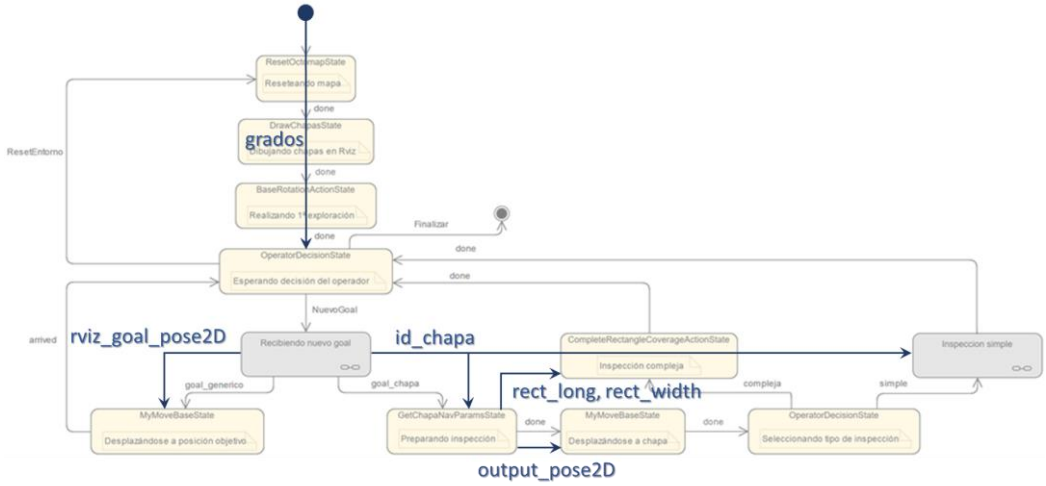

Figura 7: Flujo de datos general en el autómata

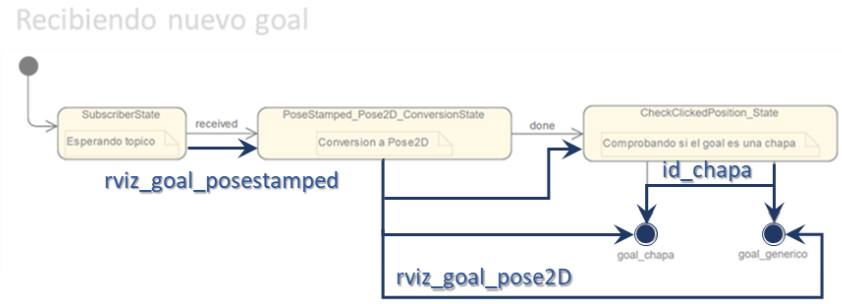

Figura 8: Flujo de datos dentro del superestado “Recibiendo nuevo goal”

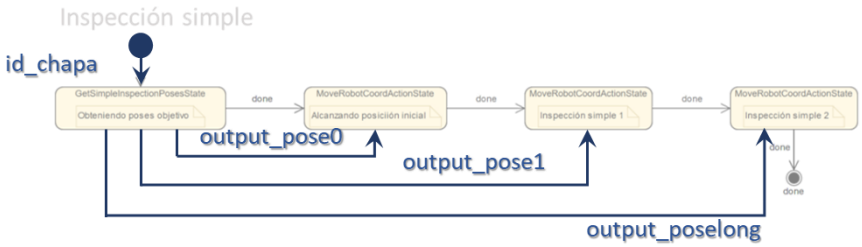

Figura 9: Flujo de datos dentro del superestado "Inspección simple"

2.4 Ejecución

Para el lanzamiento del entorno y todos los servidores basta con ejecutar en una terminal

```
roslaunch spawn_package mpo700_complete_slam_*
```

Donde * puede ser `epilab`, `example` o `factory`, en función del entorno deseado.

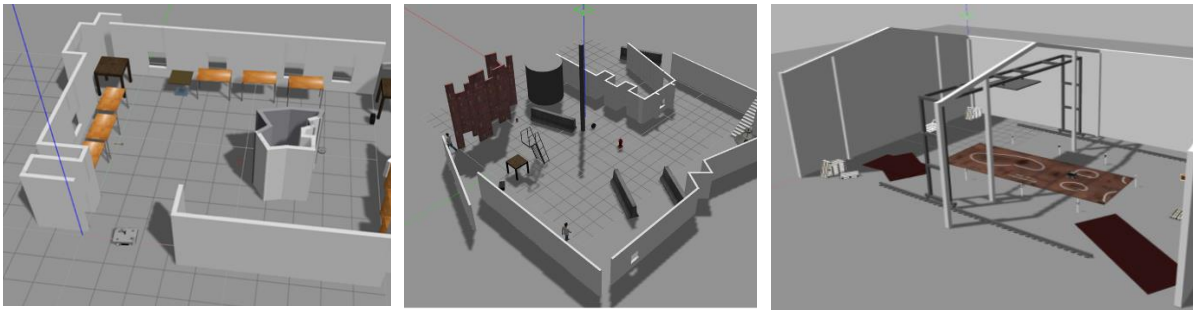

Figura 10:

Para ejecutar la máquina de estados debe ejecutarse la FlexBE app

```
roslaunch flexbe_app flexbe_full
```

Y seleccionar el *behaviour* llamado `IROBOT_navigation_StateMachine_Feb19`.S

2.5 Estructura en ROS

Como conclusión del capítulo y a modo de resumen general se incluye la Figura 11. En ella pueden observarse los principales nodos y *topics* involucrados en el proyecto, excluyendo por claridad de la explicación alguno de estos elementos.

En primer lugar, se pueden apreciar en la parte derecha los nodos que comunican a *tf* los eslabones del robot. En la figura también pueden verse los nodos que actúan sobre Gazebo para el acople de los sensores y como los *topics* que estos generan derivan en los algoritmos de SLAM y mapeo 3D. En la parte inferior también se observa el nodo *movebase* con sus *topics* más importantes y conectado al mapa, a los sistemas de coordenadas y al *behavior*. Con el autómata también se relaciona la herramienta de *Rviz* y los servidores de acciones.

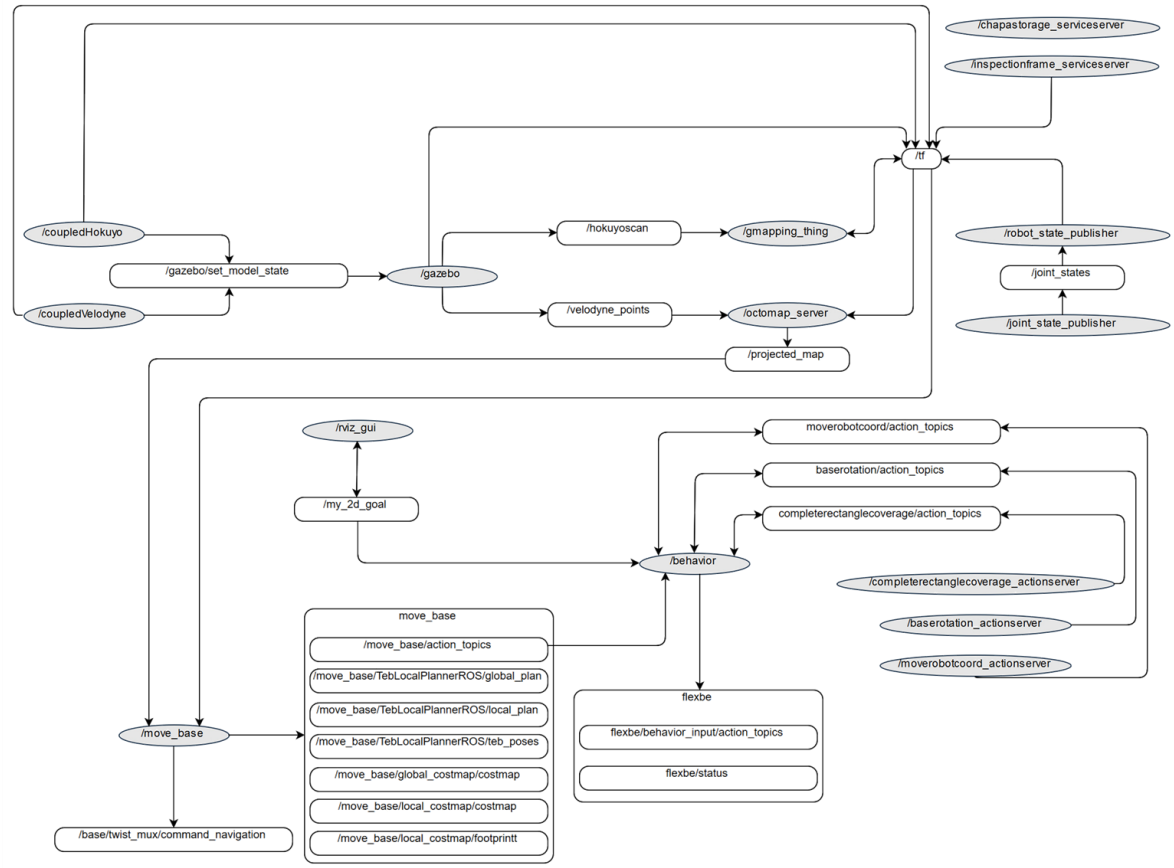

Figura 11: Representación gráfica de los principales nodos, representados por elipses, y *topics* involucrados en el trabajo

3 Entorno de generación de trayectorias de inspección y de reparación

Como se ha comentado en otros documentos, la implementación de la generación y ejecución de las trayectorias de inspección y reparación se lleva a cabo a través de una máquina de estados. Se empleará este apartado para explicar el proceso de poner en ejecución el comportamiento y los parámetros de interés.

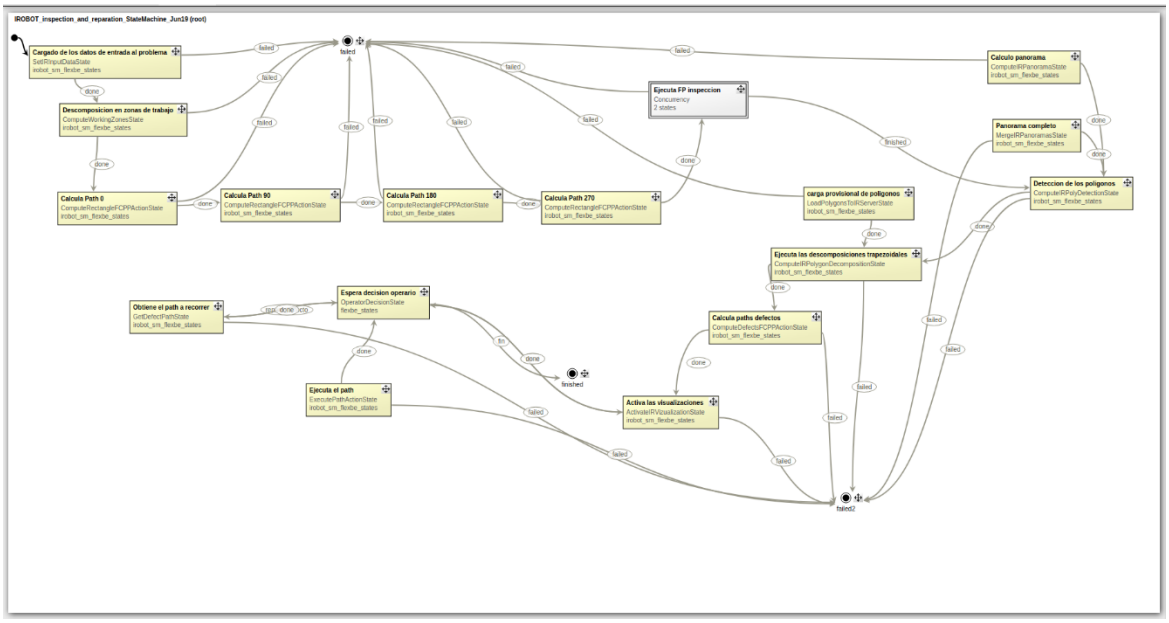

Figura 12: Máquina de estados relativa a la generación de trayectorias de inspección y reparación.

3.1 Ejecución

Para la ejecución de este entorno se incluye un archivo .sh que puede ser ejecutado desde la terminal con bash lanzaentornosummit_ir_flexbe.sh.

La posición simulada de la herramienta simulada puede modificarse en aux_controllers/main_coupledtool:

```
19
20  dist.x=0.70;
21  dist.y=0.60;
22  dist.z=0.25;
23
24  CoupledRobot coupled(SUMMITXL,dist, 0, -pi/2, 0,
    "monkey_wrench_tool","coupled_tool", true);
25
```

En las líneas 20,21 y 22 pueden modificarse los valores en metro de la traslación. Si se desea modificar la orientación de la herramienta, su roll, pitch y yaw puede modificarse respectivamente en el tercer, cuarto y quinto parámetro del constructor (línea 24).

Los parámetros de la máquina de estados pueden modificarse en la pestaña Behavior Dashboard y siempre deberán mantener la coherencia con los valores simulados.

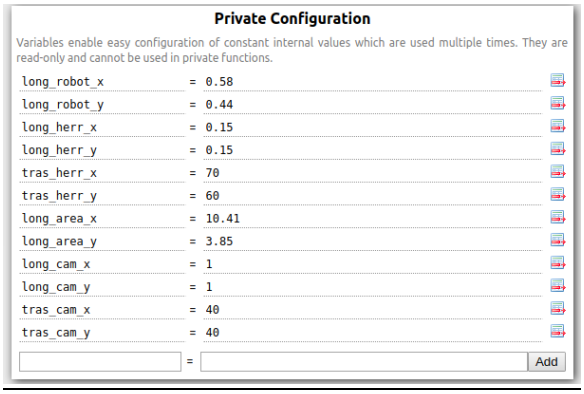

Figura 13: Parámetros configurables de la máquina de estados en la interfaz de FlexBE.

Los parámetros *long* hacen referencia al tamaño, en x e y, del elemento en cuestión y *tras* a la traslación, también en ambas coordenadas, del mismo respecto al centro del robot. La abreviatura *herr* hace referencia a la herramienta reparadora, *cam* a la cubierta con el sensor de visión, *area* al área de la chapa y *robot* a la base del robot móvil. Valores contradictorios para los parámetros pueden conllevar un comportamiento erróneo de la simulación.

La ejecución de la simulación es automática hasta la selección del defecto a reparar. Para seleccionar el defecto se debe a hacer una llamada al principal servicio de este entorno.

```
1   ~$ rosservice call /irobot_ir_data "load_inputdata: false
2                                       load_polygon: false
3                                     get_input_data: false
4                                  get_working_zonesh: false
5                                  get_working_zonesc: false
6                                        get_rot_zone: false
7                                  compute_adquireimgs: false
8                                     compute_panorama: false
9                                      merge_panoramas: false
10                                 compute_detectpolys: false
11                              compute_fcpp_defect_i: false
12                                compute_fcpp_zone_i: false
13                              execute_fcpp_defect_i: false
14                                execute_fcpp_zone_i: false
15                       change_defect_reparation_goal: false
16                                          get_poly_i: false
17                                        get_defect_i: false
18                                   get_defect_path: false
19                                     activate_vizs: false
20                                         long_robotx: 0.0
21                                         long_roboty: 0.0
22                                          long_herrx: 0.0
23                                          long_herry: 0.0
24                              tras_herrx: 0.0 0.0, z: 0.0}
25                                         tras_herry: 0.0
26                                          long_camx: 0.0
```

```
27                                                long_camy: 0.0
28                                                tras_camx: 0.0
29                                                tras_camy: 0.0
30                                               long_areax: 0.0
31                                               long_areay: 0.0
32                                                   input_poly:
33                                                       points:
34                                   - {x: 0.0, y: 0.0, z: 0.0}
35                                                    id_poly: 0
36                                                 id_defecto: 0
37                                                   id_zona: 0"
```

Como puede verse la llamada contiene muchos parámetros que se emplean de manera interna. Para llamarlo basta con escribir en una terminal “rosservice call irobot_ir_data” y pulsar dos veces el tabulador para que autocomplete el comando con los valores por defecto. Para seleccionar el desperfecto a reparar simplemente bastará con indicar su número en el *id_defecto* y activar el cambio del polígono objetivo en el parámetro *change_defect_reparation_goal*.

4 Paquetes adicionales

Además de los mencionados, el código necesita de la instalación en el sistema de otros múltiples paquetes. Estos se pueden instalar directamente mediante la instrucción de Linux:

Sudo apt-get install ros-kinetics-*

Siendo * el nombre del paquete. En ocasiones la instalación de varios paquetes se realiza por el mismo comando por lo que se recomienda la consulta de la página de la wiki de Ros de cada uno de los paquetes. También es recomendable el uso del tabulador en la terminal en Ubuntu para ver las posibles opciones de autocompletado.

```
actionlib
actionlib_enhanced
actionlib_lisp
actionlib_msgs
actionlib_tutorials
amcl
angles
aruco
aruco_detect
aruco_msgs
aruco_ros
audio_common_msgs
base_local_planner
bond
bondcpp
bondpy
calibration_msgs
camera_calibration
camera_calibration_parsers
camera_info_manager
catkin
cl_utils
class_loader
clear_costmap_recovery
cmake_modules
collada_parser
collada_urdf
compressed_depth_image_transport
compressed_image_transport
control_msgs
control_toolbox
controller_interface
controller_manager
controller_manager_msgs
costmap_2d
costmap_converter
costmap_cspace
costmap_cspace_msgs
costmap_prohibition_layer
costmap_queue
cpp_common
cv_bridge
depth_image_proc
diagnostic_aggregator
diagnostic_analysis
diagnostic_common_diagnostics
diagnostic_msgs
diagnostic_updater
```

```
dynamic_reconfigure
dynamic_tf_publisher
eigen_conversions
eigen_stl_containers
fiducial_msgs
fiducial_slam
filters
forward_command_controller
gazebo_dev
gazebo_msgs
gazebo_plugins
gazebo_ros
gazebo_ros_control
gencpp
geneus
genlisp
genmsg
gennodejs
genpy
geographic_msgs
geometric_shapes
geometry_msgs
gl_dependency
gmapping
graph_msgs
hardware_interface
hector_gazebo_plugins
image_geometry
image_proc
image_publisher
image_rotate
image_transport
image_view
interactive_marker_tutorials
interactive_markers
joint_limits_interface
joint_state_controller
joint_state_publisher
joint_states_settler
jsk_footstep_msgs
jsk_gui_msgs
jsk_hark_msgs
jsk_recognition_msgs
jsk_recognition_utils
jsk_rviz_plugins
jsk_topic_tools
kdl_conversions
kdl_parser
laser_assembler
laser_filters
laser_geometry
libg2o
libmavconn
librealsense
librviz_tutorial
map_aruco
map_msgs
map_server
mavlink
mavros
mavros_extras
mavros_msgs
media_export
```

5 Bibliografía

[1] «DataspeedInc / velodyne_simulator». [En línea]. Disponible en: https://bitbucket.org/DataspeedInc/velodyne_simulator. [Accedido: 20-ene-2019].

[2] «teb_local_planner - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/teb_local_planner. [Accedido: 21-nov-2018].

[3] «neobotix/neo_mpo_700», *GitHub*. [En línea]. Disponible en: https://github.com/neobotix/neo_mpo_700. [Accedido: 05-oct-2019].

[4] «navigation - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/navigation. [Accedido: 21-nov-2018].

[5] «costmap_2d - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/costmap_2d. [Accedido: 25-ene-2019].

[6] «costmap_2d/hydro/staticmap - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/costmap_2d/hydro/staticmap. [Accedido: 25-ene-2019].

[7] «costmap_2d/hydro/inflation - ROS Wiki». [En línea]. Disponible en: http://wiki.ros.org/costmap_2d/hydro/inflation. [Accedido: 25-ene-2019].

[8] «geometry_msgs/PolygonStamped Documentation». [En línea]. Disponible en: http://docs.ros.org/lunar/api/geometry_msgs/html/msg/PolygonStamped.html. [Accedido: 26-ene-2019].