Universidad Zaragoza
1542

Proyecto Fin de Carrera

Diseño, implementación y validación de un sistema de traducción semántica para el sistema operativo de robots ROS

Design, implementation and validation of semantic translation tool for Robots Operating System ROS

Autora

Sara Ruiz Álvarez

Director
Pablo Quílez Velilla

Ponente
Raquel Trillo Lado

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2017

Fraunhofer IPA

Diseño, implementación y validación de un sistema de traducción semántica para el sistema operativo de robots ROS

Resumen

El coste de integración en tiempo y dinero es una de las dificultades que se presenta hoy en día para la robotización a gran escala de pequeñas y medianas empresas. Con demasiada frecuencia, es necesario reprogramar código fuente existente por actualizaciones o cambios en piezas de hardware o software, porque los nuevos elementos no siguen los formatos de datos utilizados por sus antecesores por motivos técnicos o comerciales. Semánticamente esos datos representan la misma información, pero las salidas y entradas de los nuevos módulos no son compatibles.

En este proyecto se diseña una solución alternativa a la estandarización tradicional, para el sistema operativo de robots ROS, cumpliendo las premisas de creación de una herramienta de traducción semántica (STT) robusta y extensible, que funciona en un entorno industrial ROS, y es integrable con la plataforma drag&bot. Dicha plataforma es una herramienta creada para facilitar la programación gráfica y modular de robots industriales, orientada a trabajadores sin conocimientos previos de informática, con el objetivo de reducir costes en las fábricas que utilizan entornos de ROS-Industrial, el *framework* de ROS para aplicaciones industriales.

Se ha realizado siguiendo la metodología de desarrollo incremental, llegando a la implementación y validación de una herramienta práctica, que permite añadir en tiempo de ejecución nuevas transformaciones e información semántica para datos del sistema, y une de forma robusta diversos componentes a priori incompatibles por formato de datos, pero con entradas y salidas de datos que semánticamente representan la misma información. Al finalizar el prototipo, este ha sido integrado por el equipo de drag&bot en su herramienta.

El proyecto se ha realizado para el grupo de Automatización de Montaje del departamento de Robótica y Sistemas de Asistencia del instituto Fraunhofer IPA para la Automatización de la Producción en Stuttgart.

Índice

Índice de figuras

1. Introducción

La sección Introducción describe el proyecto dentro del contexto en el que se ha llevado a cabo, especificando el objetivo y alcance del mismo, así como añadiendo una descripción de cómo se ha realizado. Finalmente se encuentra una breve explicación de la estructura de esta memoria.

1.1 Objetivo y alcance del proyecto

Uno de los principales problemas presentes a día de hoy en robótica es la falta de compatibilidad entre diversos componentes de *software* y, por extensión, de *hardware*. Por ejemplo, un mismo sistema puede usar módulos de visión por computador, control de fuerza, drivers para robots, sensores y actuadores, código antiguo que debe ser mantenido y otras herramientas y algoritmos de tan diverso origen, que pueden presentar problemas al integrarlos. Los datos que necesitan compartir los diferentes componentes tienden a estar representados en formatos distintos, no siempre compatibles, aunque semánticamente se refieran a la misma información.

Esto nos lleva a la necesidad de interconectar un amplio espectro de dispositivos con diferentes características, protocolos y funcionalidades que suelen hacer la integración demasiado costosa en tiempo y dinero [1]. Mientras tanto, la industria 4.0, y las nuevas fábricas inteligentes, requieren que la programación de las máquinas sea flexible y modular para ahorrar esos costes, por lo que se necesita acelerar y simplificar los métodos de programación e integración de la robótica industrial [2].

El sistema operativo distribuido de código abierto para robótica ROS, no es ajeno a esta problemática. La solución tradicional adoptada en el mismo, es la estandarización de los canales y formatos de comunicación entre numerosos nodos y componentes que integran el sistema. Esto dificulta la reutilización de software externo o antiguo no estandarizado. Como desventaja añadida, se obtiene una limitación en cuanto a prestaciones que un componente puede dar, en comparación con las prestaciones de su diseño original. Por otro lado, aunque es deseable, la estandarización de software no es siempre fácil de llevar a cabo debido a la variedad de lenguajes de programación, librerías, calidad del código y especificaciones de las aplicaciones, lo que suele hacer más rápido y simple la reprogramación de módulos existentes en vez de la integración dentro de nuevas arquitecturas de software [3].

La herramienta *drag&bot*, es un software con interfaz web para programación fácil de robots industriales. Su interfaz gráfica de usuario permite a personas no expertas generar bloques de secuencias de código reutilizables sin necesidad de conocimientos de programación previos. Incluye un conjunto de funciones independientes del robot, como moverlo, abrir una pinza o localizar objetos con una cámara, escondiendo la complejidad de la programación de robots. Estos bloques pueden ser parametrizados de forma rápida e intuitiva gracias a los asistentes que provee. [4] Está programada para funcionar en ROS, por lo que sufre los mismos problemas de compatibilidad entre componentes descritos anteriormente, aunque se pretende evitar al usuario, en la medida de lo posible, los problemas derivados dicha incompatibilidad.

El objetivo de este proyecto ha sido evaluar la posibilidad de integrar una herramienta semántica en ROS para interconectar diferentes componentes, desarrollando y validando un prototipo de sistema de traducción semántica integrado como herramienta en el sistema operativo ROS, compatible así con la plataforma drag&bot.

Se ha estudiado cómo representar ROS y sus componentes de software como ontología, así como los diferentes mecanismos que posibilitan configurar el sistema de un modo sencillo para el usuario realizando las transformaciones semánticas pertinentes.

Analizando las tecnologías existentes que pudieran ser de utilidad para el proyecto, la arquitectura de ROS y drag&bot, así como los casos de uso más relevantes, se ha diseñado e implementado el prototipo de traducción semántica integrado como nodo en ROS. De esta forma se proveen servicios que pueden ser utilizados por otros nodos de ROS para obtener información de elementos incompatibles mediante una traducción automática de los mismos. Dicho prototipo ha sido validado en un entorno ROS controlado, comprobando de este modo su viabilidad para la integración en la herramienta drag&bot.

1.2 Contexto

El proyecto se ha realizado para el grupo de Automatización de Montaje del departamento de Robótica y Sistemas de Asistencia del instituto Fraunhofer IPA para la Automatización de la Producción en Stuttgart. Es el grupo donde se ha desarrollado la herramienta drag&bot. A su vez, Fraunhofer IPA gestiona la parte europea del consorcio ROS-Industrial, en un esfuerzo internacional por integrar el sistema ROS en aplicaciones industriales.

Se ha trabajado a distancia, con visitas a las instalaciones del centro en Stuttgart para una mejor planificación y validación de la herramienta a desarrollar.

1.3 Trabajo Previo

ROS es utilizado tanto en robótica de servicio como en industrial, mientras que en el primer caso se han aprovechado tecnologías semánticas para el aprendizaje de los robots [5][6], con resultados notables; no se han encontrado herramientas de traducción semántica para ROS que puedan ser aplicadas en robótica industrial de forma útil y práctica, para facilitar su programación a través de drag&bot. Aunque la integración semántica de hardware ha sido estudiada en varios proyectos europeos como ReAPP [7] tras la finalización de dichos proyectos, los frameworks resultantes no han tenido una amplia repercusión por su excesiva complejidad. Este proyecto surge como continuación de una prueba de concepto basada en servicios web en el marco del proyecto SMERobotics [1]. Integrar la traducción como nodo ROS en vez de servicio web, permite su utilización en entornos sin acceso directo a internet, lo que puede representar una ventaja en cuanto a estabilidad del sistema y velocidad de uso en determinadas situaciones. Además, no imposibilita una integración posterior con tecnologías web si fuera necesario.

1.4 Metodología y tecnologías utilizadas

Se ha seguido una metodología de desarrollo incremental [8], haciendo primero un análisis del problema, para pasar a diseñar e implementar un sistema sencillo en el que se unieran *topics* de ROS equivalentes, y pasando después a ampliar el prototipo para que realizara las transformaciones necesarias de los datos en tiempo de ejecución. En ambos casos se han realizado pruebas para comprobar la viabilidad de la solución adoptada.

Se ha trabajado utilizando el lenguaje de programación Python, con la Api de ROS para el mismo sobre Linux.

1.5 Estructura de la memoria

En el capítulo 2 se hace un análisis de la tecnología sobre la que debe funcionar el prototipo, así como una obtención de requisitos y diagramas de análisis de la herramienta a desarrollar. El capítulo 3 está dedicado a detalles de diseño e implementación, en qué afecta el desarrollo incremental, la elección de lenguaje y las herramientas para el mismo, cómo se realizan las búsquedas, qué estructura tiene la aplicación, su interfaz y los detalles de concurrencia. En el cuarto capítulo se habla de los tipos de pruebas realizadas para validar el sistema y los resultados obtenidos. Y por último, el apartado 5 contiene las conclusiones.

2. Análisis

En este capítulo se explica el análisis realizado del sistema operativo con el que se debía trabajar en este proyecto, las herramientas disponibles para realizar las transformaciones requeridas, los casos de uso más relevantes, los diagramas diseñados con ellos y el análisis de tecnologías semánticas.

2.1 Sistema Operativo para Robótica ROS y ROS-Industrial

ROS-Industrial está soportado por un consorcio internacional de empresas e instituciones. Es un *framework* flexible para escribir software para robots que se establecía como requisito en la realización de este proyecto. Se compone de una colección de herramientas, librerías y convenciones que llaman a simplificar la tarea de crear comportamientos para robots complejos y robustos que puedan funcionar sobre una amplia variedad de plataformas.

ROS-Industrial es un proyecto de código abierto que extiende las capacidades de ROS para ser utilizado en robótica industrial. La figura 1 muestra su arquitectura.

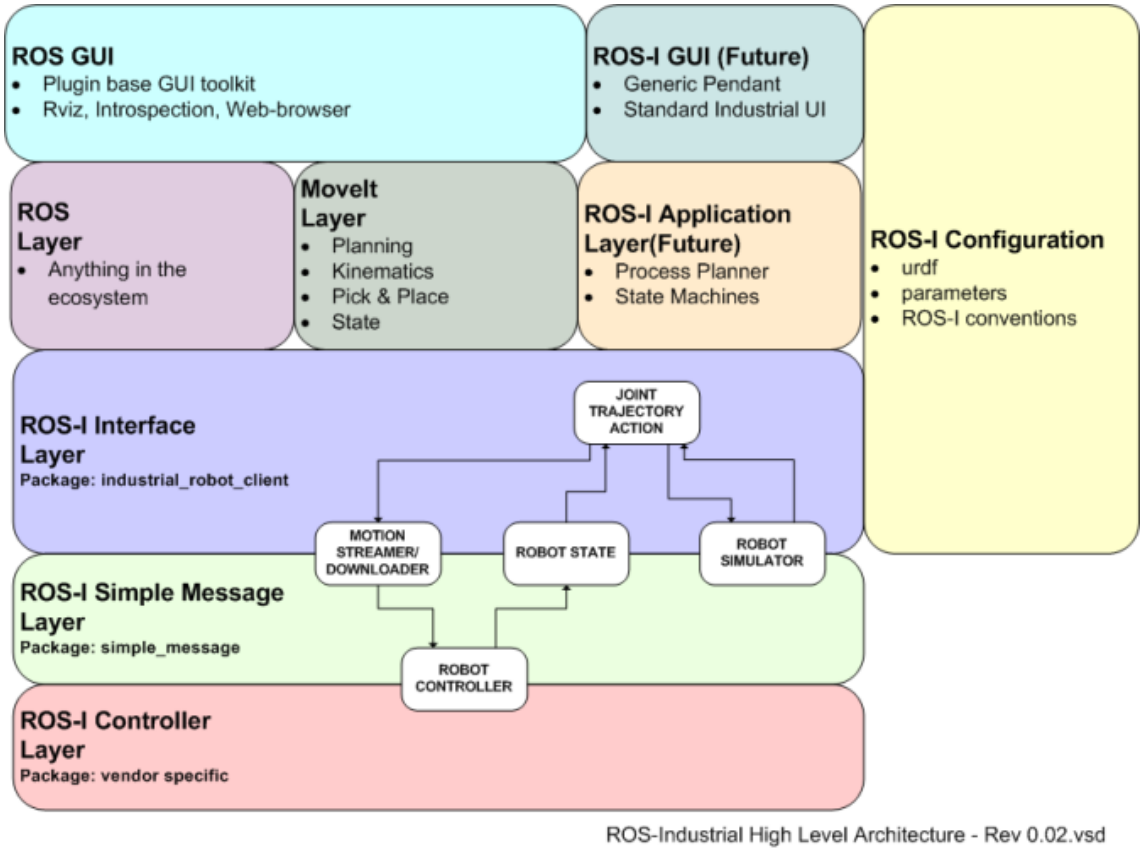


Figura 1: Niveles de Arquitectura de ROS-Industrial

Se estudió ROS-Industrial para obtener información acerca de cómo debía ser la herramienta de traducción semántica a realizar. A continuación, se presenta una breve explicación de los conceptos más relevantes para la misma.

2.1.1 Conceptos relevantes de la arquitectura de ROS

Para el desarrollador, el sistema se compone de programas, nodos, dependientes de uno en concreto llamado ROS-master. Estos se ejecutan en paralelo, e intercambian mensajes con otros nodos mediante los llamados *topics*. Un topic es un canal de comunicación en el que uno o varios nodos pueden publicar mensajes (*publishers* o escritores) en un formato concreto de mensaje, determinado en la creación del *topic*; y en el que uno o varios nodos pueden estar leyendo esos mensajes (*subscribers* o lectores). Los mensajes se leen de forma ordenada, y el hecho de que un nodo recoja un mensaje, no hace que este deje de estar disponible en el *topic* para otros nodos. Por ejemplo, un nodo controla el giro de un motor, otro nodo ejecuta acciones de localización, otro deposita en *topics* continuamente información recogida de un sensor láser, otro monitoriza el estado del sistema, otro muestra la información que viene de una cámara, etc.

Un nodo, puede proveer servicios y acciones. Los servicios, como se ve en la figura 2, son tareas que otros nodos pueden pedir que se realicen, dando unos datos de entrada y recibiendo datos de salida. La comunicación en los servicios se realiza de forma análoga a los mecanismos internos en los que se basan los *topics*, pero de forma simplificada. Los datos de entrada y salida tienen formatos concretos que se eligen en el momento de crearlos. Las acciones se plantean para tareas que tengan una larga duración en el tiempo, y también tienen entrada y salida, así como una monitorización periódica del estado de la acción; además, se pueden cancelar en mitad de la ejecución. La comunicación en las acciones se realiza mediante *topics* [9].

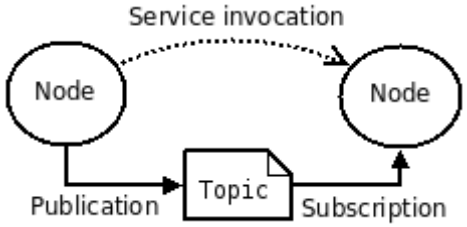


Figura 2: Conceptos básicos de ROS

Todos estos pasos de mensajes, se realizan con formatos de datos concretos definidos en los paquetes estándar de ROS o en determinados componentes de *software*, y a menudo el *software* disponible para un nodo, necesita información de *topics* que ya existen en el sistema, pero los formatos son incompatibles, aunque semánticamente representen lo mismo. Un ejemplo recurrente es el de las posiciones de un robot, que pueden ser representadas en cuaterniones o usando ángulos de Euler. La información es equivalente, una posición de un robot en un momento determinado, pero su representación es incompatible sin una conversión.

A su vez, múltiples *topics* pueden estar publicando datos del mismo formato, (por ejemplo, mensajes con un Float64, un *string*, y una lista), y aunque las herramientas del sistema permiten obtener el tipo de mensaje de cada *topic* en ejecución, no es posible saber a qué categoría semántica pertenecen, pueden estar refiriéndose a datos de tipos semánticos diferentes, un número entero puede representar temperatura, distancia, ángulo, etc.

2.1.2 Entorno en ROS como ontología de dominio

Considerando ROS como un árbol de categorías semánticas, por extensión, se puede asociar un *topic* con una de ellas. Esta taxonomía permite construir una jerarquía de mensajes asociados a elementos existentes en el mundo real. Por lo general, en el ámbito industrial en el que ROS es utilizado, muchos de ellos corresponden a variables físicas como posición, orientación, aceleración o fuerza, pero no se ven necesariamente limitados en la categorización. La taxonomía permite a su vez la composición de categorías de forma que categorías padre, puedan quedar identificadas a

través de la suma de categorías hijas. Las funciones de transformación modelan la información sobre composición de categorías.

Por ejemplo, considerando que la posición del robot, definida como *Pose*, queda determinada por la posición cartesiana Position y la orientación de la pinza en el espacio *Orientation*, se puede construir un árbol de la siguiente forma:

/Pose

/Pose/Position

/Pose/Orientation

Donde */Pose* será la unión de */Position* y */Orientation* siempre que exista un transformador que realice la composición. En sentido inverso, se puede descomponen la categoría de nivel superior, en categorías de nivel inferior con el transformador que realice la extracción. La prioridad de este proyecto ha sido obtener un sistema flexible que permita ampliaciones de forma sencilla.

2.1.3 Requisitos obtenidos

Del análisis de ROS se dedujo que la herramienta debe proveer una forma de buscar en el sistema qué *topics* contienen la información semántica pedida, y hacer una o varias transformaciones necesarias, dando el resultado equivalente en el formato requerido. Puesto que ROS no almacena información semántica de cada *topic*, la herramienta debe proporcionar algún método para añadirla. Además, una vez establecida la transformación o transformaciones a realizar, estas deben permanecer activas en el tiempo y procesarse continuamente, para cada dato publicado en el *topic* de origen, hasta que ya no sea necesaria más información. Como puede ser necesario traducir información de diversos *topics*, en los que se publica en paralelo y a diferentes velocidades, la herramienta de traducción semántica debía proveer también una forma de realizar varios trabajos de traducción de forma simultánea.

Los nodos no son estáticos en un sistema, sino que el número puede variar durante el tiempo de ejecución, se pueden añadir nuevos componentes a un robot, nuevos robots que trabajen juntos o simplemente nodos que realizan nuevas tareas y crean nuevos *topics*. Por lo que se añadía como requisito la necesidad de incorporar en tiempo de ejecución nuevas transformaciones disponibles de datos, y eliminar otras no relevantes o que deben ejecutarse de forma diferente.

2.2 drag&bot

El proyecto tenía como requisito técnico inicial que el prototipo sea integrable en la herramienta drag&bot, por lo que se analizó la misma.

Drag&bot ofrece una solución gráfica de programación, por bloques o a través de asistentes, para robots mediante interfaz web para ser usado en ordenadores o *tablets* desde Internet, o desde una intranet. El objetivo es que trabajadores sin experiencia en desarrollo de software, puedan ser capaces de asignar nuevas tareas a los robots de una fábrica para cambiar el método de producción, por ejemplo, al trabajar con nuevo modelo de producto fabricado, con piezas que deben ser montadas por el robot en diferente lugar u orden.

En la figura 3 se observa la interfaz gráfica de la herramienta, tanto del constructor de bloques como del asistente para trayectorias. Cada bloque que ve el usuario, es un módulo que ejecuta un programa Python sobre ROS, abstrayendo al usuario de los detalles de programación del mismo. Cuando necesite buscar un origen de datos para un bloque, se le mostrarán los semánticamente compatibles, para que sea más fácil la programación. La herramienta desarrollada en este PFC, puede ser integrada como asistente, o de forma más transparente para el usuario.

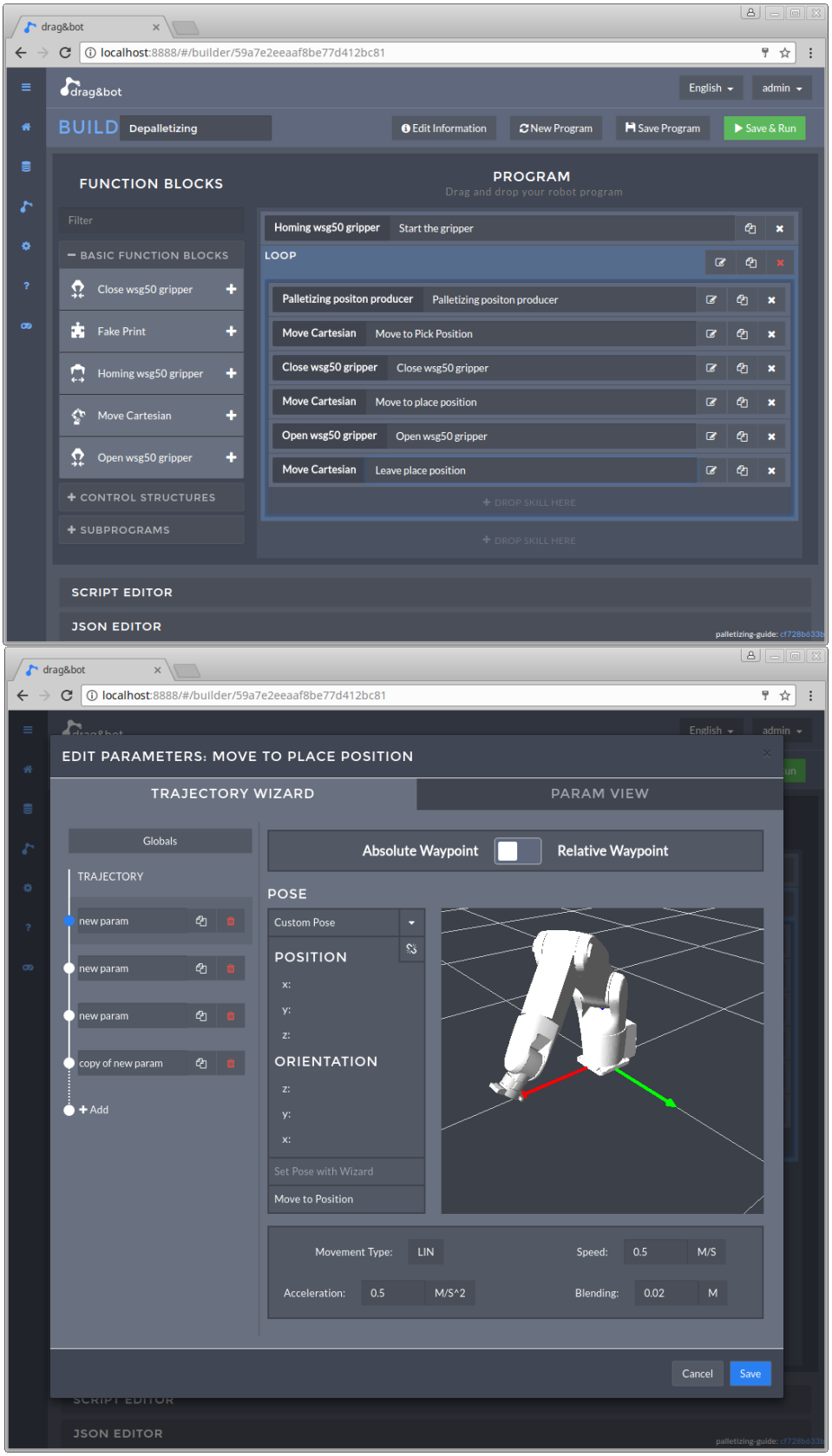


Figura 3: drag&bot constructor de bloques y asistente para trayectorias

El sistema drag&bot se divide en partes representadas en la Figura 4: *frontend*, *backend*, y uno o varios *robot systems* (basados en ROS).

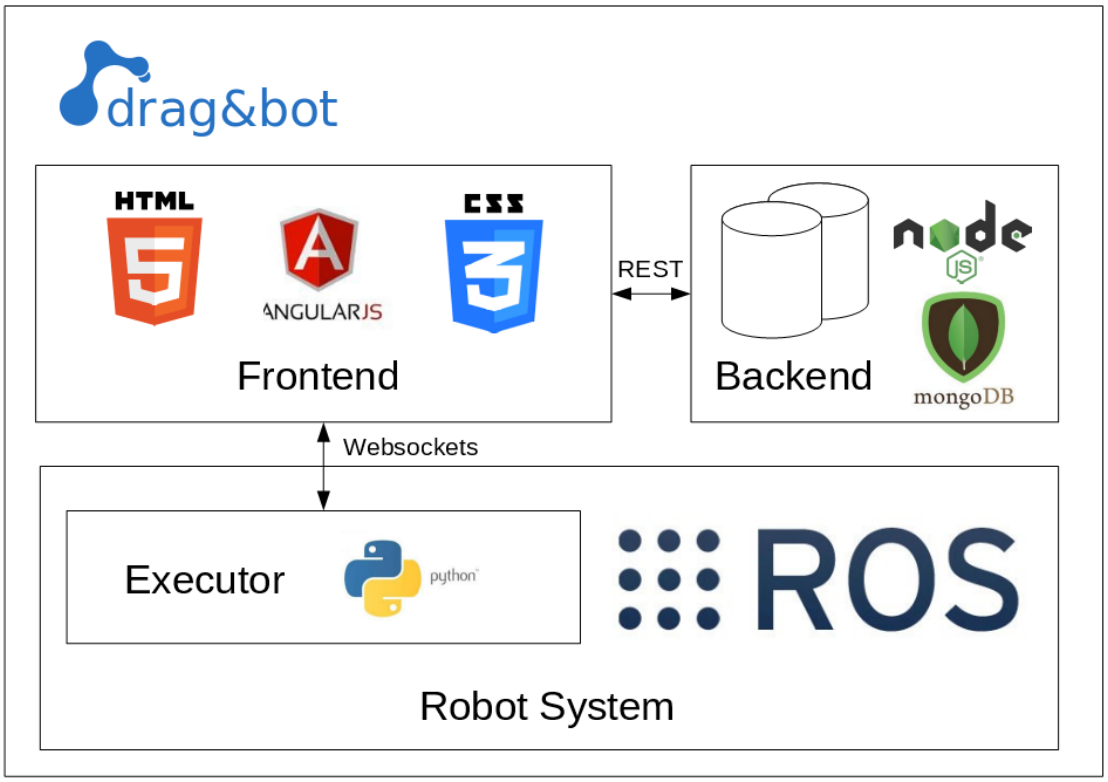


Figura 4: Diagrama de clases de drag&bot

Frontend es la página web implementada en HTML5, AngularJS y CSS, y puede funcionar en diferentes servidores web como Apache, nginx o Node.js. Se comunica con el servidor de *backend* a través de la interfaz REST proporcionada por el mismo. A través de la librería *roslibsjs* [10] y en extensión Robot Web Tools [11] la web puede acceder a los datos de diferentes *robot systems*, siendo cada uno de ellos un entorno ROS independiente.

Backend es el servidor. Almacena en una base de datos MongoDB los bloques, configuraciones o información sobre los usuarios de drag&bot. El entorno de ejecución Node.js ejecuta la aplicación JavaScript que proporciona la interfaz de acceso REST.

Robot system se refiere a un entorno ROS independiente conectado a diferentes componentes *hardware*. Cada *robot system* tiene su correspondencia física con una celda con un robot y diferentes sensores (cámara 2D, cámaras 3D, barreras de luz, etc.) y actuadores (autómatas, robots, pinzas, etc). En cada *robot system*, un nodo ROS llamado Rosbridge [12] proporciona acceso externo al entorno ROS a través de un WebSocket vía TCP/IP[13]. Este nodo permite acceder a *topics*, servicios y acciones. En cada *robot system* se encuentra también un nodo ROS llamado *Executor*, que es el encargado de ejecutar la máquina de estados creada por el usuario a través del *frontend* de drag&bot. Cada estado corresponde a un bloque y cada bloque es un programa Python con acceso a la API de ROS. Estos bloques pueden, por tanto, obtener información de *topics* y llamar a servicios y acciones de otros nodos ROS, que pueden ser componentes de software o drivers para sensores, robots o actuadores. Cada *driver*, es también un nodo ROS que realiza la conexión final con el *hardware*, y como a día de hoy no existen estándares, por lo general se realiza de forma diferente para cada elemento de hardware.

Con esta arquitectura, la herramienta desarrollada en este proyecto puede integrarse de dos formas no excluyentes: bien como herramienta externa, con una interfaz web cómoda para el usuario, de forma similar a como se interactúa con Executor; bien siendo llamada desde el propio nodo Executor.

2.3 Transformaciones

Los datos a reenviar de unos elementos a otros del sistema, pueden ser enviados entre servicios, *topics* y acciones, primando para el prototipo la implementación de reenvío y transformación entre *topics*.

El caso más simple es una transformación con un dato de entrada y uno de salida, pudiéndose encadenar un conjunto de transformaciones. Un ejemplo sencillo sería pasar de milímetros a kilómetros encadenando una transformación de milímetros a metros con una de metros a kilómetros, tal y como se ve en la figura 5, donde los triángulos son categorías semánticas, y las flechas transformaciones de un dato de una categoría en otra. En un entorno ROS en funcionamiento, se pueden añadir nodos, en cualquier momento, y a su vez *topics*, acciones y servicios de categorías semánticas desconocidas a priori, por lo que es un requisito de la herramienta añadir nuevas transformaciones. Siguiendo el ejemplo de la figura 5, se añade la categoría decámetros, y transformaciones para transformar los datos con dos de las categorías ya existentes. Esto añade un camino más, el usuario puede elegir cuál escoger.

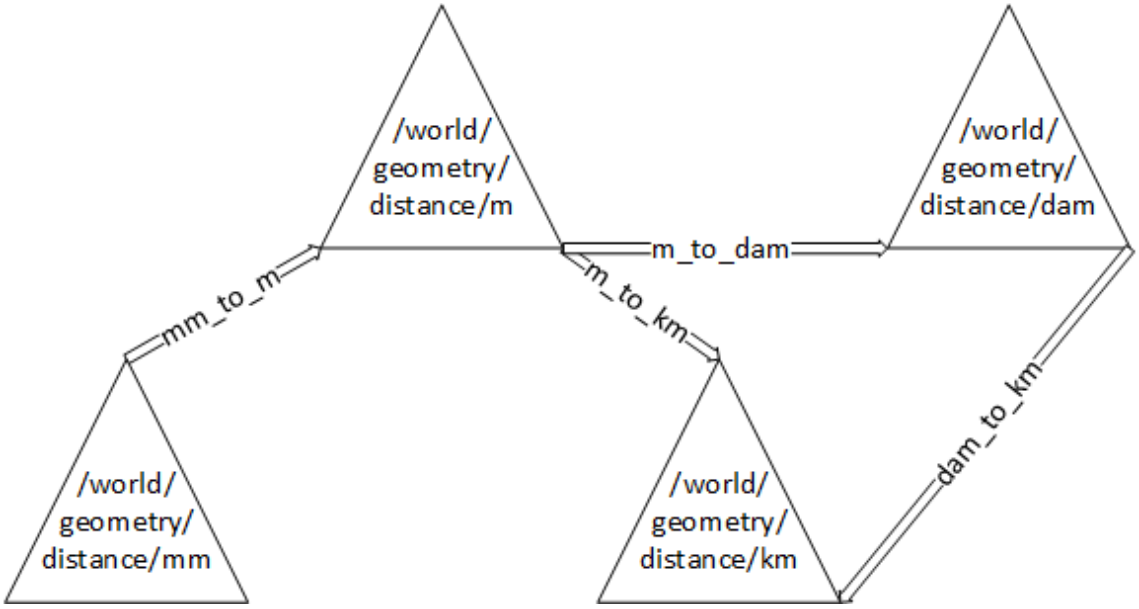


Figura 5: Ejemplo de relaciones entre transformaciones y clases semánticas

A veces de un dato, se pueden extraer porciones de información más pequeñas, especialmente en aquellos compuestos, con una transformación que extraiga la información necesaria, es un caso análogo al anterior.

El caso contrario sucede cuando se necesitan varios datos para una salida, la composición.

2.4 Requisitos

Una vez establecidas las necesidades de la aplicación, se redactaron las siguientes listas de requisitos, estableciendo junto al grupo de desarrollo de drag&bot, niveles de prioridad en

requisitos funcionales a diseñar e implementar en el prototipo, para seguir el modelo de desarrollo incremental, comenzando por los niveles de máxima prioridad.

2.4.1 Requisitos no funcionales

De los dos primeros requisitos iniciales se dedujo un tercero respecto a la implementación, puesto que las librerías para la API de ROS, están programadas para ser usadas en Python o C++:

- Funcionar en ROS
- Integrable en drag&bot: interfaz de la aplicación compatible con nodo ROS *Executor* y/o compatible con *frontend*.
- Programada en Python o C++

2.4.2 Requisitos funcionales

La siguiente tabla especifica los requisitos funcionales, así como su nivel de prioridad: máxima en los requisitos de nivel 1, y según aumentan de nivel, bajan en prioridad.

Nombre del requisito	***Nivel de prioridad***
Almacenar transformaciones disponibles	2
Añadir en ejecución nuevas transformaciones disponibles	2
Eliminar en tiempo de ejecución, transformaciones disponibles	2
Añadir información semántica de transformaciones disponibles	3
Buscar *topics* en el sistema ROS y la información de sus mensajes	1
Añadir información semántica de elementos del sistema	1
Modificar información semántica de elementos en el sistema	1
Eliminar información semántica de elementos en el sistema	1
Reenviar información entre elementos semánticamente compatibles:	-
Sin transformación	1
Con 1 transformación	2
Con una cadena de transformaciones	3
Extrayendo sólo parte de la información del mensaje	2
Con varios *topics* de entrada	4
Desactivar proceso de reenvío de elementos existente	2
Almacenar información de conexiones:	-
Entre elementos sin transformaciones	1
Con transformaciones	2

El nivel de prioridad se estableció de tal modo, que al cumplir los requisitos de un nivel de prioridad, el sistema pudiera ser utilizado en la práctica, y permitiendo saber en cada paso, que es una herramienta viable.

2.5 Casos de uso

Identificados los requisitos se pasó a identificar los casos de uso más comunes para la herramienta de traducción STT (*Semantic Translation Tool*). Se identificaron dos actores, por un lado, el usuario del sistema, y por otro quien añada o elimine las transformaciones. Ambos pueden

ser la misma persona, y en el caso de integrar el sistema en drag&bot, será la herramienta la que interactúe como intermediario pudiendo añadir más transformaciones en nuevas versiones.

A partir de ahora se distinguirá entre "transformaciones disponibles" y "transformaciones en ejecución", siendo las primeras, las que el sistema almacena para ser ejecutadas, y refiriéndose el segundo término a las conexiones entre *topics* con transformaciones, que también deben ser almacenadas por el sistema para volver a ejecutarlas entre apagados y encendidos del mismo.

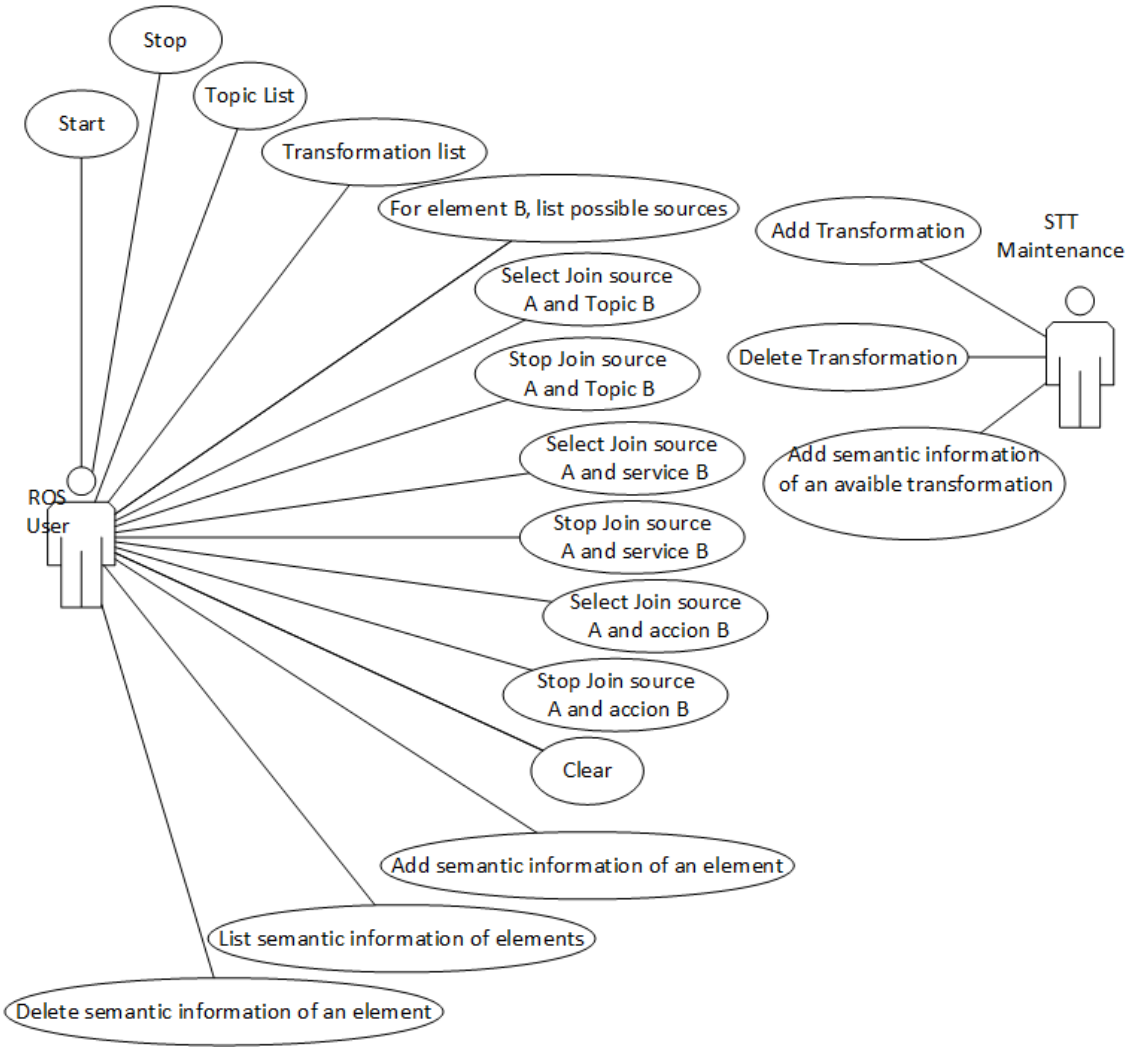


Figura 6: Diagrama de casos de uso

La figura 6 muestra el diagrama de casos de uso, a continuación se explica cada uno.

Start: inicia STT, cargando información disponible de sesiones anteriores, incluyendo los procesos de reenvío entre *topics*.

Stop: para la ejecución de STT, con ello, se paran también las transformaciones en ejecución, pero quedan guardadas para cuando el sistema se vuelva a iniciar. De este modo, no será necesario volver a ejecutar todas las búsquedas y configurar las transformaciones cada vez que el robot se apague y se vuelva a encender.

Topic list: ofrece un listado de los *topics* del sistema y la información semántica almacenada de cada uno, si la hay.

Transformation list: devuelve un listado de las transformaciones disponibles almacenadas en el sistema para ser ejecutadas al unir un *topic* con otro, y su información semántica almacenada de cada una si la hay.

For element B, list possible sources: busca un listado de *topics*, acciones o servicios semánticamente compatibles con el elemento B, que puede ser un *topic*, acción o servicio (mediante transformaciones o si los hay compatibles directamente).

Select Join source A and Topic B: publicar en el *topic* B la información del elemento A, que puede ser topic acción o servicio, si es semánticamente compatible.

Stop join source A and Topic B: desactiva el proceso de reenvío de datos entre el elemento A, que puede ser *topic*, acción o servicio, y el *topic* B, deteniendo también las transformaciones en ejecución para esa conexión, si las hay.

Select Join source A and service B: publicar en la entrada del servicio B la información del elemento A, que puede ser *topic* acción o servicio, si es semánticamente compatible.

Stop join source A and action B: desactiva el proceso de reenvío de datos entre el elemento A, que puede ser *topic*, acción o servicio, y el servicio B, deteniendo también las transformaciones en ejecución para esa conexión, si las hay.

Select Join source A and action B: ejecutar la acción B usando como entrada la información del elemento A, que puede ser *topic* acción o servicio, si es semánticamente compatible.

Stop join source A and action B: desactiva el proceso de reenvío de datos entre el elemento A, que puede ser *topic*, acción o servicio, y la acción B, deteniendo también las transformaciones en ejecución para esa conexión, si las hay.

Clear: desactiva todas las conexiones de elementos existentes en el sistema.

Add semantic information of an element: añade al STT la información semántica relativa a un elemento del sistema, puede ser *topic*, servicio o acción. Esta información no se puede obtener de ROS.

List semantic information of elements: busca el listado de la información semántica almacenada de los elementos del sistema.

Delete semantic information of an element: elimina del STT la información semántica relativa a un elemento del sistema, puede ser *topic*, servicio o acción. No elimina el elemento, pero sin información semántica, STT no lo podrá usar como fuente para conexiones.

Add Transformation: añade al STT una nueva transformación de datos disponible para ser utilizada en conexiones de *topics*.

Delete Transformation: elimina del STT una transformación de datos disponible. Si está siendo usada para conectar *topics*, se desactiva también esa conexión.

Add semantic information of an avaible transformation: añade al STT información semántica de los datos de entrada y de salida de una transformación disponible.

Buscar las fuentes de origen de los datos, podría incluirse en los casos de uso que implican conectar dos elementos del sistema. Al separarlos, se evita la repetición de este paso para el usuario cuando vuelve a unir elementos que ya había unido y separado anteriormente, y que por ello, conoce ya la cadena de transformaciones de los mismos. Por ejemplo, cuando vuelve utilizar la posición de una pinza que había dejado de necesitar cuando el robot ejecutaba otras tareas o tenía acoplada otra pieza como un destornillador.

2.6 Diagramas de secuencia y de flujo de datos

Para facilitar el modelado del sistema, con los requisitos y casos de uso obtenidos, se diseñó una serie de diagramas de secuencia y flujo de datos de los casos de uso más representativos. A continuación se muestran unos ejemplos representativos, el conjunto entero de diagramas de secuencia y flujo de datos están en los anexos A.3 y A.4.

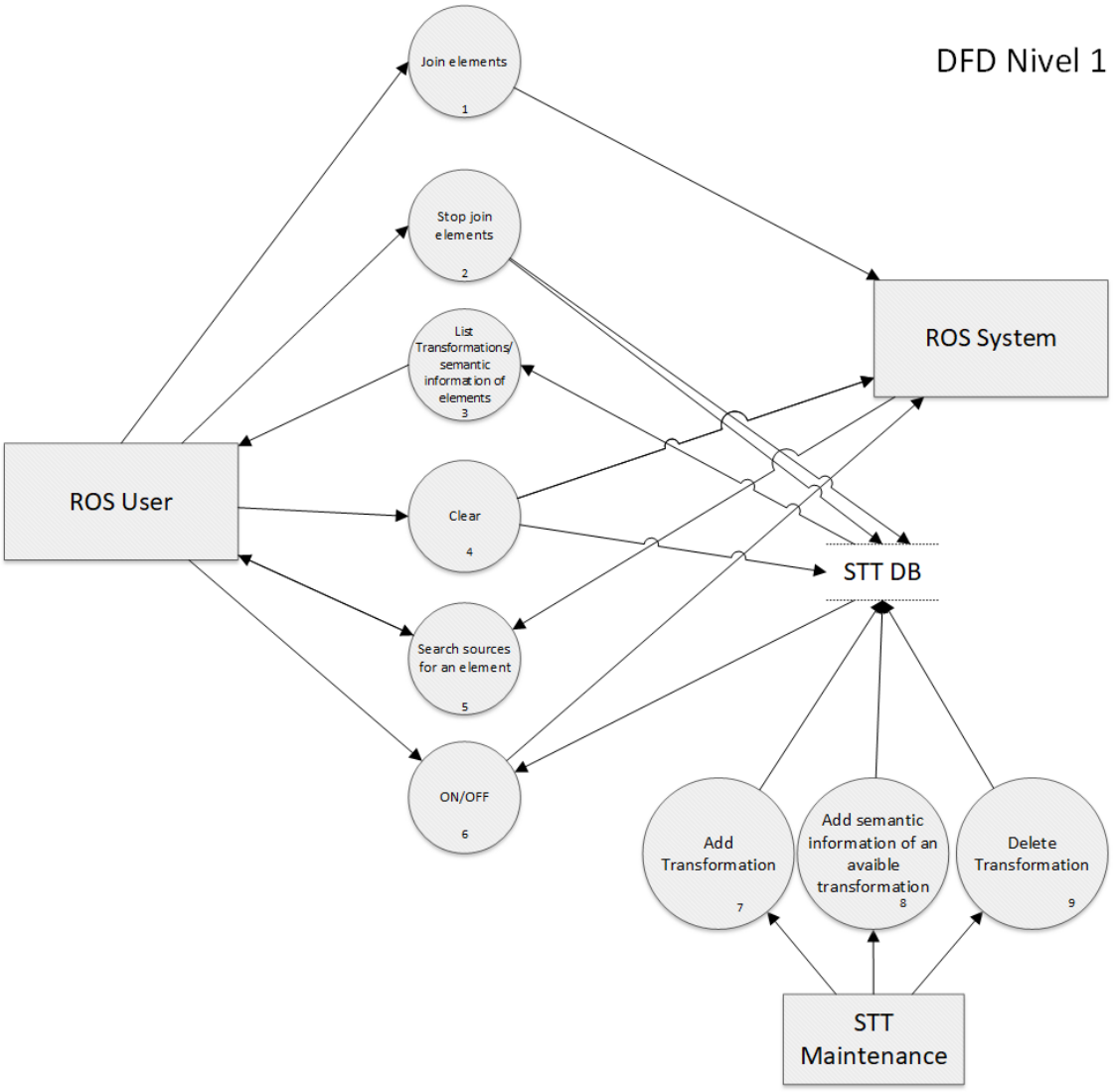


Figura 7: Diagrama de flujo de datos, nivel 1: diagrama de nivel superior de STT

En la figura 7 se observa el flujo de datos con la base de datos y el sistema ROS de cada caso.

En el nivel dos, del caso 1 (figura 8), unir elementos del sistema (válido para topics, servicios y acciones), se comprobó que podían ser casos independientes para la interfaz; por un lado, buscar transformaciones, y por otro, realizar la conexión. De ese modo, si el usuario ya conoce las transformaciones necesarias, ahorra el tiempo de buscarlas.

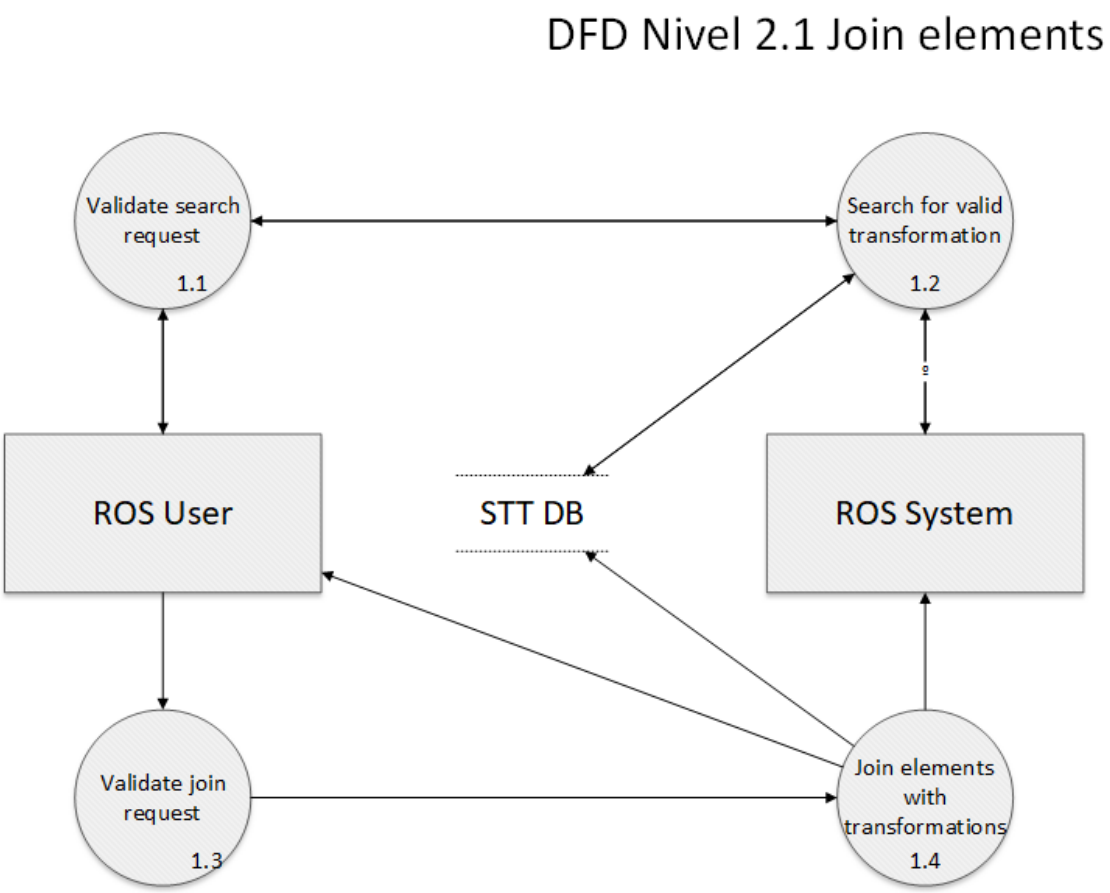


Figura 8: Diagrama de flujo de datos de nivel 2: conectar elementos del sistema

La secuencia de acciones a realizar para este caso, uniendo dos *topics* cuya transformación es posible, se detalla en la figura 9.

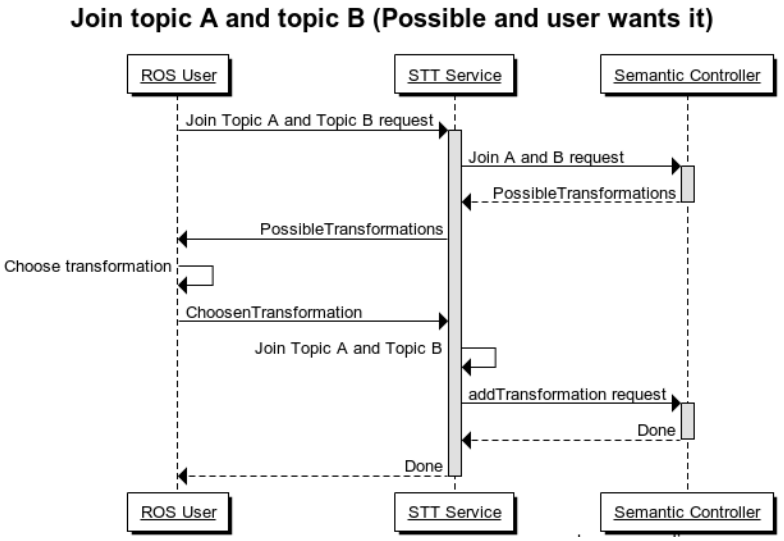


Figura 9: Diagrama de secuencia: buscar conexión y conectar topics válidos

2.7 Tecnologías analizadas para trabajar con semántica

Se buscó información sobre tecnologías para modelar e implementar el grafo de información semántica como YAML[14], OWL[15], RDF[16] o GraphML[16]; de bases de datos para almacenarlas Neo4j; y de razonadores semánticos Hermit[17] o Pellet[18].

Las búsquedas de transformaciones son un caso a usar menos en relación a la ejecución de las transformaciones, y además, es un caso de uso cuya respuesta se da a un usuario humano, no a un sistema robótico, por lo que el tiempo de ejecución es menos crítico para drag&bot que unir *topics*. Por este motivo, a utilizar un grafo en OWL para usarlo posteriormente con un analizador semántico como Hermit (más rápido que Pellet [19]) se le asignó prioridad baja en el diseño incremental de la aplicación. Primero se debía implementar una forma de trabajar con semántica que no dependiese de librerías externas, facilitando a su vez integración en entornos diversos, y se pasaría a implementar la representación con herramientas semánticas en posteriores iteraciones de diseño e implementación del desarrollo incremental.

3. Diseño e implementación

A lo largo de este capítulo se explica cómo se diseñó la aplicación y el porqué de las tecnologías utilizadas, así como los detalles más representativos de la implementación.

3.1 Desarrollo incremental

Siguiendo la metodología de desarrollo incremental, las fases de diseño e implementación se fueron sucediendo de forma iterativa, comenzando por los casos de uso que cumplieran los requisitos de máxima prioridad vistos en el apartado 2.1.5 del capítulo de Análisis, hasta llegar a tener implementados aquellos de prioridad 3.

3.2 Elección de lenguaje y librerías

Las librerías de ROS-Industrial están implementadas para ser usadas en lenguaje Python y C++. Era imprescindible añadir nuestras transformaciones a la aplicación en tiempo de ejecución, que tienen como entradas y salidas tipos de datos desconocidos en tiempo de programación, y que pueden ser estructuras de datos complejas. C++ es un lenguaje fuertemente tipado [20], lo que añade complejidad a la integración de transformaciones y el reenvío de información entre *topics* cuyo tipo de datos de mensaje, tampoco es conocido en tiempo de programación. El tipado dinámico de Python [21], en cambio, permite ejecutar las mismas funciones para diferentes tipos. De este modo, se pueden ejecutar funciones de los ficheros de transformaciones sin saber el tipo de datos, y sin necesidad de realizar llamadas que ejecuten órdenes de un *shell*. Lo que baja el tiempo de ejecución. Además, se comprobó empíricamente al comienzo del diseño, que el reenvío de datos de un *topic* a otro ejecutado mediante código en Python, no añade un tiempo reseñable en la ejecución de un sistema robótico que trabaja a 100Hz como máximo. Se decidió, por tanto, usar Python para la implementación del prototipo.

En cuanto a las versiones del lenguaje, se ha usado Python 2.7, la versión para la que están implementadas actualmente las librerías de ROS y ROS-Industrial.

3.3 Representación y búsqueda de información semántica

La elección de lenguaje de programación supuso el descarte en implementación de OWL como herramienta para representación de la semántica, debido a que las principales herramientas encontradas, como OWLready, están creadas para ser usadas con la versión 3 de Python, y presentan incompatibilidades con la 2.7 [22]. Esto, unido al hecho de crear una herramienta para desarrolladores, que pueda funcionar en entornos ROS variados, los cuales a veces se ejecutan en ordenadores industriales muy limitados en cuanto a capacidades, motivó implementar la búsqueda de elementos compatibles directamente en el prototipo, y dejar para futuras versiones las pruebas con analizadores semánticos más potentes.

3.4 Estructura de la aplicación

STT sigue un esquema modular clásico de modelo-vista-controlador [23]. Con una interfaz basada en servicios ROS, pudiendo ser así integrada en drag&bot de las dos formas descritas ya en análisis.

3.4.1 Diagrama de clases

En la figura 10 se muestran las principales clases del sistema.

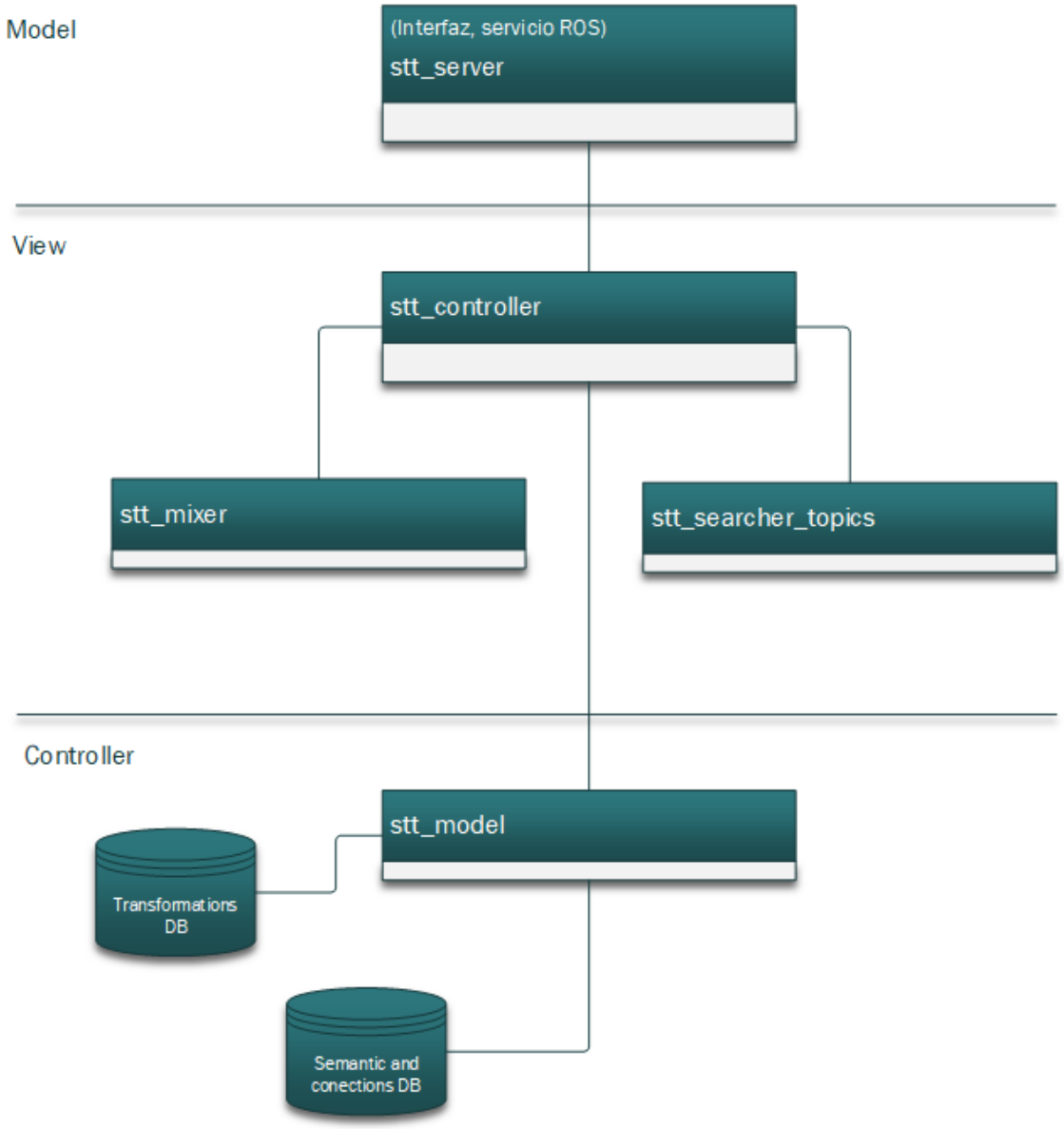


Figura 10: Diagrama de clases

Stt_server se encarga de la interfaz, evalúa las órdenes que llegan al sistema y llama a la función correspondiente de *stt_controller*.

Stt_model almacena y elimina los datos en el sistema, tanto de transformaciones disponibles y su semántica, como de las conexiones que se realizan. Si el sistema se cae, se queda almacenada información de esas conexiones y se vuelven a conectar al iniciarse.

Stt_controller recoge las peticiones de *stt_server* y ejecuta las acciones necesarias, incluyendo las búsquedas semánticas. Para ello llama a ***stt_mixer*** que une *topics* entre sí, y a ***stt_searcher_topics*** que los busca en el sistema ROS.

3.4.2 Estructura de archivos

El árbol de directorios sigue el esquema organizativo de un paquete de ROS. En la figura 11 se muestran los directorios, que deben estar dentro de un *workspace* de ROS.

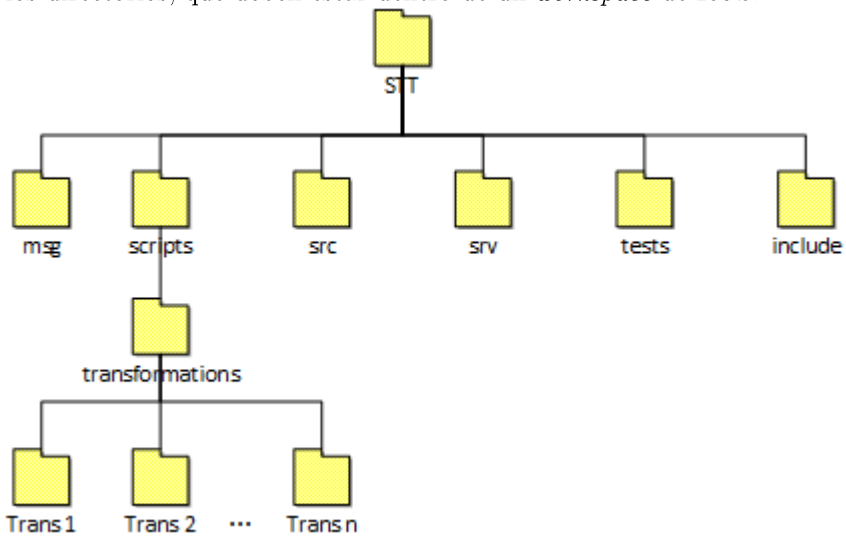


Figura 11: Árbol de directorios

Las carpetas */STT/msg* y */STT/srv*, almacenan formatos de datos de mensajes y de servicios respectivamente, que la herramienta necesita para funcionar al crear los servicios ROS necesarios.

La carpeta */STT/scripts* contiene los paquetes vistos en el diagrama de clases.

/STT/tests se utiliza para almacenar información de las pruebas realizadas.

/STT/src y */STT/include* son creadas automáticamente al crear el paquete ROS, se dejan en el árbol con el objetivos de posibilitar la integración de futuros nodos realizados en C++.

En */STT/transformations* se añaden las transformaciones que el sistema podrá ejecutar. Cada una será una carpeta cuyo nombre coincide con el de la transformación que ejecuta. Dentro se encuentra un archivo llamado *__init__.py* con una función llamada *transformate(data)*, que será a la que llame el sistema cuando necesite realizar la transformación referida en esa carpeta, por lo que deberá tener como entrada las variables necesarias, y devolver el resultado en la salida. Puede contener más archivos si son necesarios para ejecutar el código de la transformación. De este modo, se importan en tiempo de ejecución como módulos, las transformaciones de datos necesarias, y se pueden añadir nuevas transformaciones sin apagar el sistema ni detener las que ya se están usando.

3.4.3 Bases de datos de conexiones y semántica de los elementos del sistema

Para dar robustez a la herramienta desarrollada, se guardan en disco en todo momento las conexiones realizadas entre elementos del sistema, así como los datos semánticos conocidos.

drag&bot cuenta con una base de datos MongoDB, pero se descartó utilizarla en este caso, porque está en el *backend*, y los datos del entorno ROS le llegan mediante conexión por Internet. Esto supone que para usarla debe haber una conexión estable a Internet, y que el *frontend* debe estar conectado en todo momento, mientras que drag&bot está pensado para conectarla sólo en momentos de desarrollo cambios y monitorización puntual.

La variedad de entornos en los que trabaja ROS, y las posibles limitaciones o incompatibilidades de algunos de ellos para almacenar grandes bases de datos, produjo la decisión

de almacenar directamente en ficheros la información de conexiones y semántica. Se consideró además que la segunda versión, ROS 2.0, se está desarrollando como software de tiempo real [24].

Se ha comprobado que guardar de este modo los datos, mantiene una velocidad aceptable para un sistema robótico en una celda, puesto que utilizan los datos almacenados al añadir por parte del usuario nueva información, o al buscar y nuevas conexiones, pero no es necesario leer de la base de datos cada vez que se ejecuta una transformación de datos de los mensajes de un *topic* para republicarlos en otro. Además se reducen las dependencias al mínimo, para facilitar la integración de la herramienta.

3.5 Interfaz y modo de uso

STT está implementado como nodo ROS, con el que se interactúa mediante servicios ROS. Así se puede acceder al sistema desde consola o desde otros programas que utilicen las librerías de ROS. Para activarlo, primero hay que activar un entorno ROS, y una vez se ejecuta este, se llama al STT mediante la siguiente orden del entorno ROS, donde *stt* es el nombre del paquete, y *stt_server.py* el del ejecutable.

```
$ rosrun stt stt_server.py
```

Con STT activo, se almacenan en ROS los servicios que presta el programa. Varios nodos diferentes pueden llamar a los servicios de la herramienta sin que esta se bloquee.

Cada servicio ejecuta una orden del sistema de traducción, enviando para ello una petición donde puede haber datos, en el formato concreto especificado para cada servicio, y recibirá la respuesta correspondiente. Una vez que se ha iniciado el programa, es posible consultar a ROS qué servicios hay en el sistema, y qué formatos de entrada y salida tiene cada uno, incluyendo los del traductor semántico. Esto se puede realizar desde un programa, o desde consola del siguiente modo:

`$ rosservice list` lista los servicios activos.
`$ rosservice type` muestra el tipo de un servicio.

Para llamar al servicio deseado, basta con ejecutar la orden:

```
$ rosservice call "service_name"
```

Como ejemplo, para borrar la información de una categoría semántica, se ejecuta desde consola:

```
$ rosservice call remove_semantic_category  /world/distance/m
```

que es equivalente a realizar la siguiente llamada desde un programa Python que utilice las librerías de ROS:

```
rospy.wait_for_service('remove_semantic_category')
try:
    result = rospy.ServiceProxy('remove_semantic_category', '/world/distance/m')
except rospy.ServiceException, e:
     print "Service call failed"
```

donde *'result'* almacena un dato de tipo *boolean* que indica si se ha borrado la categoría indicada correctamente.

Esta interfaz es totalmente compatible con drag&bot, y permite integrar la herramienta tanto en el *frontend*, como para ser llamada desde el *executor*, cumpliendo así todos los requisitos no funcionales del apartado 2.1.4 de esta memoria.

3.6 Concurrencia

Cuando se unen dos elementos del sistema ROS, por ejemplo, dos *topics*, STT lee cada dato que se publica en el *topic* de origen, realiza las transformaciones necesarias para que sea un mensaje compatible con el *topic* de destino, y publica el dato convertido en el *topic* de destino. Esto se realiza para cada conexión de elementos, y los nodos publican información en los *topics* a diferentes velocidades, llegando incluso algunos a parar su ejecución sin previo aviso. Además, las transformaciones pueden ser muy variadas, algunas muy rápidas y sencillas como conversiones de medidas, y otras más lentas al trabajar con datos más pesados como imágenes, incluso algunas de ellas pueden fallar. Para evitar bloqueos en la herramienta, se utilizan *threads*.

Para cada conexión que se realiza, se crea un *thread* encargado de la misma. El *thread* importa la transformación o transformaciones correspondientes, y a partir de entonces, y hasta que se le manda parar la conexión, espera cada nuevo dato en el *topic* de origen, realiza la transformación, y republica en destino el resultado. Se puede observar este proceso en la figura 12.

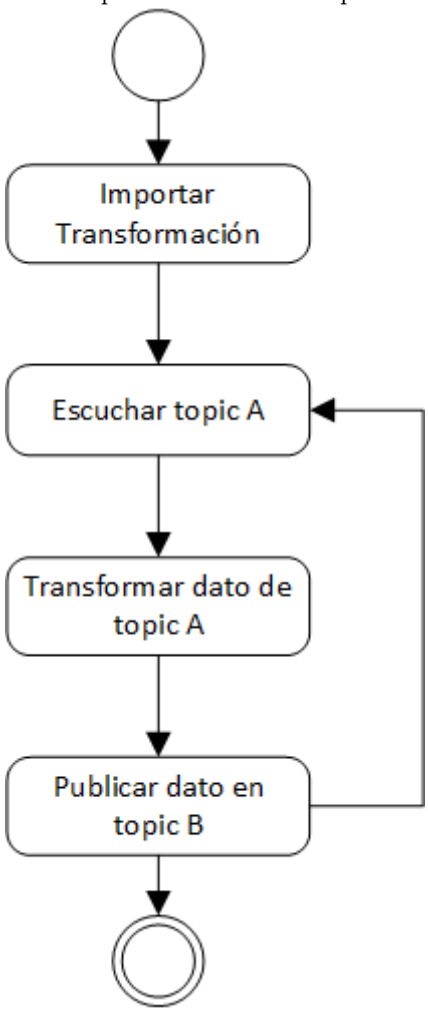


Figura 12: Diagrama de actividad de un thread de conexión

Los *threads* no se comunican entre sí, ni dependen unos de otros, y si un *topic* deja de tener datos nuevos, sólo quedará parado el que lo tenga como origen, el resto seguirán ejecutando sus tareas. Además, las bibliotecas *rospy*, *rostopic* y otras que permiten acceder a funcionalidades de bajo nivel de ROS, son seguras para *threads*.

3.7 Búsqueda de transformaciones

Un caso de uso de la herramienta es: para un elemento dado como un *topic*, devolver qué elementos del sistema semánticamente compatibles se pueden usar como origen de datos para ese *topic*, y con qué cadena de transformaciones. Relacionado con este, otro es dar al sistema dos elementos, y que STT busque cómo transformar los datos del primero en el segundo y republicarlos transformados.

El conjunto de todas las transformaciones disponibles forma un grafo dirigido inconexo con ciclos, que va cambiando de forma en ejecución según se añaden o eliminan transformaciones. Con el siguiente algoritmo, se buscan caminos en el sistema tomando como caminos válidos todos los que tienen como origen de la cadena de transformaciones un elemento con categoría semántica compatible con el destino. El algoritmo 1 resuelve el primer caso:

Algoritmo 1

```
search_compatible_origins(in: semantic_type_produced, used, current_path;
                          in out connections_found):
    for t in all_transformations do
        if (t is not in used) and (t.semantic_target = semantic_type_produced) then
            add t to used
            add t to current_path
            for element in all_elements_semantic_information do
                if semantic_compatible(t.semantic_origin, element) then
                    add(element, current_path) to connections_founds
                end if
            end for
            search_compatible_origins(t.semantic_origin, used, current_path, connections_found)
            delete_last_element(connection_path)
        end if
    end for
end search_compatible_origins
```

Para cada transformación cuya salida es semánticamente compatible con la de destino, y que no esté usada en el camino actual (para evitar ciclos en los caminos de transformaciones), se comprueba si hay elementos en el sistema compatibles con su entrada, de ser así, el camino está completo y se añade al conjunto de caminos encontrados (*connections_found*). Además, se busca recursivamente si hay caminos más largos con origen en otros elementos, puesto que el más corto no tiene por qué coincidir con el que más interese al usuario. Por ejemplo, puede necesitar la temperatura del robot en un punto, como en una pieza para soldaduras, y que el camino de transformaciones más corto tenga como origen un *topic* donde se publica la temperatura de otra parte del sistema, como la ambiental.

Para el segundo caso, el proceso de búsqueda es similar, pero en vez de buscar elementos compatibles, busca los caminos que tengan como origen el elemento pedido. Si se encuentra un

camino, se descarta buscar cadenas de transformaciones que lo tengan como subcadena, tal y como se muestra en el algoritmo 2. Después, se elige el más corto.

Algoritmo 2

```
search_transformation ( in: semantic_type_produced, used, current_path, origin_element;
                        in out connections_found):
    for t in all_transformations do
        if (t is not in used) and (t.semantic_target = semantic_type_produced) then
            add t to used
            add t to current_path
            if semantic_compatible(t.semantic_origin, origin_element) then
                    add(origin_element, current_path) to connections_founds
            else
                search_transformation(t.semantic_origin, used, current_path, origin_element
connections_found)
            end if
        end if
        delete_last_element(connection_path)
    end for
end search_transformation
```

Los algoritmos 1 y 2, se han descrito así por claridad. En la herramienta, se ha implementado con una indexación en los bucles. Así sólo se ejecuta el código interior del bucle para los datos que cumplen la condición, y se evita recorrer secuencialmente los conjuntos de transformaciones, informaciones semánticas y elementos usados. Para ello se ha aprovechado que los datos se almacenan como tablas *hash*, lo cual baja sensiblemente el coste en el caso promedio, puesto que se espera que el conjunto de transformaciones almacenado sea muy heterogéneo, lo que generará un grafo compuesto por muchos grafos no conectados entre sí.

4. Validación

En este apartado se explican las pruebas realizadas al prototipo, tanto para comprobar su correcto funcionamiento, como su integrabilidad en drag&bot. Las pruebas unitarias y de integración se han realizado durante el proceso de implementación, para comprobar en cada paso que lo añadido funcionaba, las pruebas de rendimiento, estabilidad y despliegue se han realizado con la herramienta final.

4.1 Pruebas unitarias

Para cada función de cada módulo, se han realizado pruebas unitarias, asegurando que funcionan antes de integrarlas con otros módulos. Estas pruebas se han realizado de dos formas. Para las funciones más sencillas, se ha utilizado el intérprete de Python o la consola para ejecutar órdenes de ROS. Para el resto, se han diseñado porciones de código como el que sigue, comprobando los diferentes casos posibles. Primero con la captura de interrupciones desactivada, para ver qué fallos había, y después añadiendo esa captura para evitar que el sistema caiga cuando falla una parte (por ejemplo, la carga de un fichero de una nueva transformación) y sigan activas las conexiones que se están ejecutando.

```
# ------------------------------------------------------------------
# Tests without error cases
# ------------------------------------------------------------------

def unit_test_all_transformations():

  semantic_categories_add_or_update("world/geometry/distance/m",
"std_msgs/Float64")
  semantic_categories_add_or_update("world/geometry/distance/km","std_
msgs/Float64")

semantic_categories_add_or_update("world/geometry/distance/mm","std_ms
gs/Float64")
  ti_1 = TransformatorInfo("m_to_km", "world/geometry/distance/m",
"world/geometry/distance/km", "m_to_km.py")
  ti_2 = TransformatorInfo("mm_to_m", "world/geometry/distance/mm",
"world/geometry/distance/m", "m_to_km.py")

  print ti_1

  available_transformations_add_or_update(ti_1)
  available_transformations_add_or_update(ti_2)
  print [[x, y] for x, y in all_transformations.iteritems()]
  available_transformations_load()
  print [[x, y] for x, y in all_transformations.iteritems()]
  available_transformations_remove(ti_1.id)
  available_transformations_remove(ti_2.id)
  print [[x, y] for x, y in all_transformations.iteritems()]
```

4.2 Pruebas de integración

Se han implementado una serie de *scripts* en *bash*, con llamadas a los servicios del STT, para comprobar que los componentes del sistema una vez integrados, funcionan correctamente para todos los casos de uso.

Cuando se ejecuta cada *script*, se comprueba en pantalla, o mediante las herramientas de ROS que la ejecución se realiza correctamente. En la figura 13, se ve parte la salida del *script integration_1_semantic_categories.sh*, encargado de comprobar que al añadir o eliminar categorías semánticas, el sistema funciona con las salidas esperadas en caso. En la figura, la salida mostrada es en mitad de la ejecución, cuando debe mostrar en pantalla un conjunto dado de categorías y el formato ROS de los datos pertenecientes a dicha categoría semántica.

Figura 13: Salida de consola para prueba de integración

La lista completa de tests de integración es:

- *integration_1_semantic_categories.sh*
- *integration_2_topic_semantic_information.sh*
- *integration_3_transformations.sh*
- *integration_4_search.sh*
- *integration_5_join.sh*
- *populate_semantic_categories.sh*

Los *scripts* de integración de 1 a 3, comprueban las operaciones *CRUD* (crear, leer, actualizar y borrar) sobre los datos almacenados en el sistema.

El cuarto genera un grafo y realiza operaciones de búsqueda, y el 5 realiza transformaciones. El script *populate_semantic_categories* sirve para generar un árbol semántico con el que realizar otras pruebas.

4.3 Pruebas de rendimiento

Por el uso esperado de STT, el punto crítico de rendimiento son las conexiones con transformaciones, por lo que estas pruebas se han centrado en comprobar el retraso producido por la herramienta al reenviar información de un *topic* a otro, con transformaciones, y en diferentes condiciones. Para el desarrollo de estas pruebas, se ha utilizado siempre el mismo ordenador, con CPU Intel Core i7-7500U 3.5GHz sobre el que se ejecuta una máquina virtual de 9GB de memoria en la que está instalado *XUbuntu* 64bits, con el sistema ROS instalado en él. También se ha comprobado el uso de memoria mientras se realizaban, gracias al visualizador de procesos *htop*. En todas ellas el consumo de memoria se ha mantenido por debajo del 1% y no se han detectado

fugas de memoria. Con ello, se ha comprobado que el consumo de memoria no es un aspecto crítico para el uso de la aplicación.

4.3.1 Retraso introducido en cadenas de transformaciones

Mide el retraso que se produce al leer los datos de un *topic*, realizar una cadena de transformaciones y publicarlos en otro. Se ha implementado el *script test_delay_monitor.py*, que publica en un *topic* un dato, y lee ese dato de otro *topic*, comprobando el retraso y haciendo la media cada 100 datos (retraso medio en 1 segundo). El *script test_call_join.py*, realiza la conexión de esos *topics*, mediante el número de transformaciones indicado en los datos de entrada multiplicados por 2 (realiza una transformación simple y su inversa, de metros a milímetros y de milímetros a metros, el número de veces que indiquen los datos de entrada, reenviando el mismo dato que había al comienzo). La tabla siguiente muestra la media de los tiempos de retraso en segundos para diferente número de transformaciones. Se ha realizado la media entre 200 mediciones de los tiempos de retraso medios obtenidos por segundo:

Número de transformaciones/2	1	5	10	20	50	100
Número de transformaciones	2	10	20	40	100	200
Tiempo medio (s)	0.00073637	0.00066445	0.00080224	0.00077827	0.00094993	0.00110064

El retraso introducido por el sistema es de una milésima de segundo al republicar los *topics* con 200 transformaciones intermedias, un retraso insignificante en un caso extremo.

4.3.2 Comportamiento al crecer el número de publicadores y subscriptores

ROS tiene un mecanismo de descarte de paquetes en las colas de entrada de los subscriptores para cuando llegan más paquetes de los que el sistema es capaz de procesar. Con esta prueba se comprobó la pérdida de mensajes que se producía según aumentaba la carga en el sistema.

Para realizarla se utilizan los scripts *test_delay_monitor_multiple.py*, que publica y se subscribe a un número de *topics* dados, publicando a 100Hz; y *test_call_join_multiple.py*, que llama al STT para conectar los *topics* creados con el primer *script*.

Los resultados calculados son:

- Para 10 conexiones de 10*2 transformaciones se satisfacen el 100% de los paquetes.
- Para 25 conexiones de 10*2 transformaciones se satisfacen el 100% de los paquetes.
- Para 50 conexiones de 10*2 transformaciones se satisfacen el 100% de los paquetes.
- Para 75 conexiones de 10*2 transformaciones se satisfacen el 90% de los paquetes.
- Para 100 conexiones de 10*2 transformaciones se satisfacen el 60% de los paquetes.

Se comprobó que en estas condiciones, el sistema ejecuta bien hasta unas 10000 transformaciones por segundo. En un sistema típico, como los que se utilizan con drag&bot, suele haber unas 100.

4.4 Prueba de estabilidad

Se mantuvo la aplicación 24 horas encendida con transformaciones en ejecución con 50 conexiones. No hubo problemas de fuga de memoria, ni dejaron de funcionar las transformaciones ni hubo ningún problema por seguir usando la herramienta después para añadir nuevas transformaciones.

4.5 Prueba de despliegue

Con este test se comprueba la utilidad y el funcionamiento del STT con un robot. El *software* utilizado es URSim que simula idénticamente al controlador real, un robot UR10, mostrado en la figura 14.

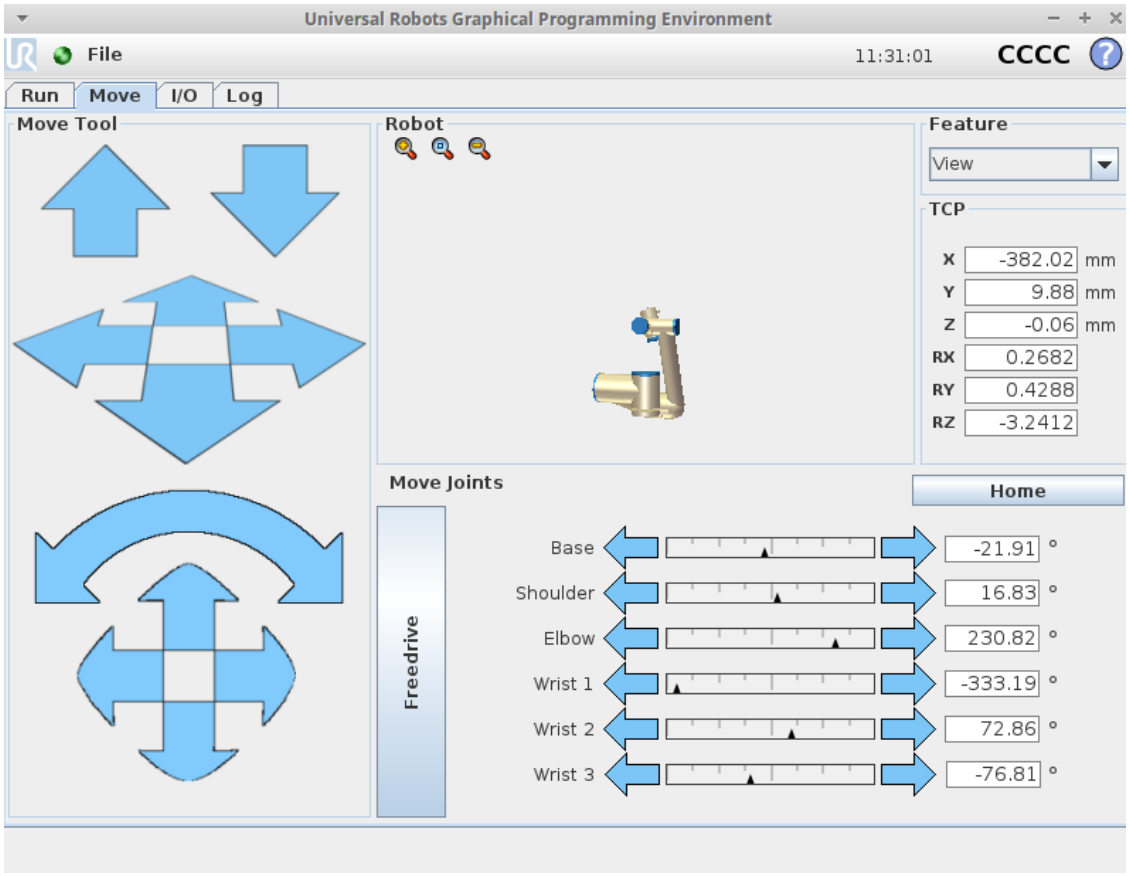


Figura 14: Controlador gráfico de robot UR10

Para monitorizar el estado de la prueba se utilizó la herramienta gráfica de ROS *rqt*, tal y como se ve en la figura 15. Después puso en funcionamiento el STT, y se ejecutó el driver del robot. Por último, se realizaron llamadas al servicio de la herramienta para realizar la transformación de Euler a cuaterniones, publicar los datos obtenidos desde el *topic /tool_frame* en el *topic topic /quaternion*.

No se observó que se introdujera ninguna demora apreciable hasta el límite de resolución del programa *rqt*. Este es un caso de uso real necesario que había que resolver para el *framework* drag&bot.

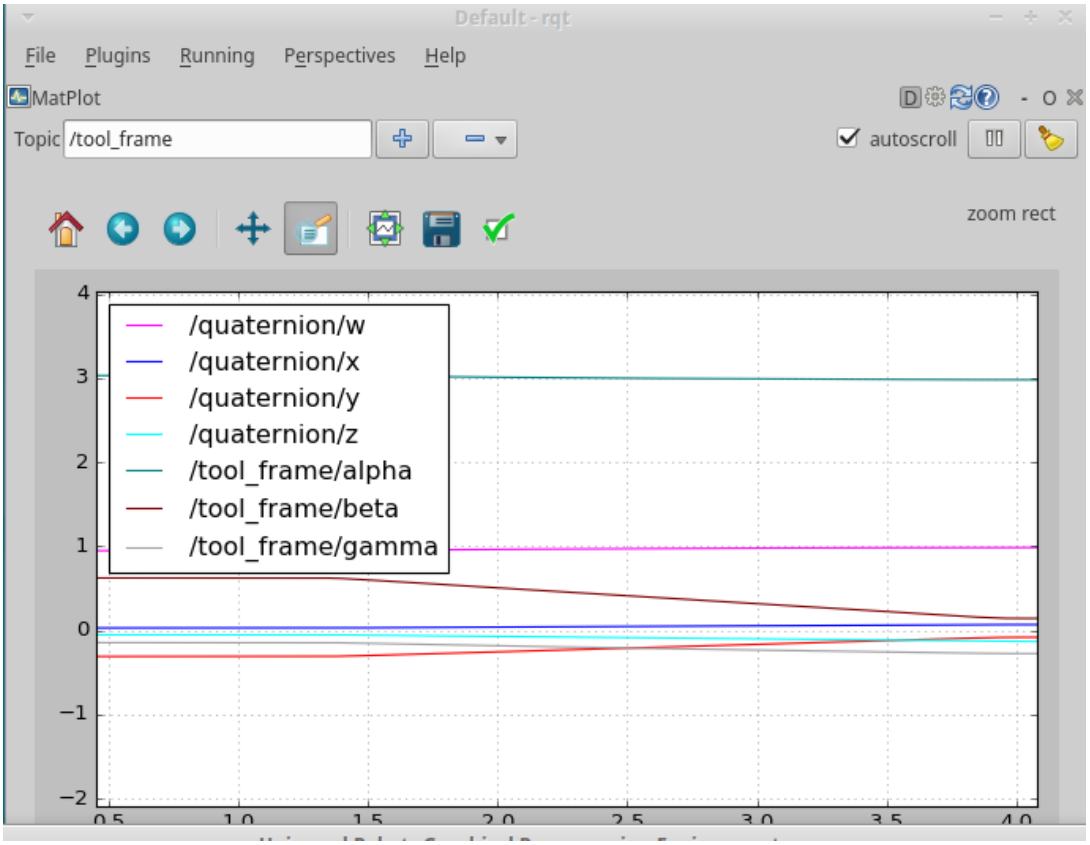


Figura 15: Salida de la herramienta rqt en prueba de despliegue

5. Conclusiones

En este capítulo se exponen las conclusiones del proyecto, el trabajo que se ha realizado, qué se ha conseguido, cómo se puede ampliar en futuras versiones, cómo se ha distribuido el tiempo mientras se llevaba a cabo, y una valoración personal sobre el mismo.

5.1 Resumen del trabajo realizado

En el tiempo de duración del proyecto, mediante desarrollo incremental, se ha conseguido desarrollar y validar un prototipo de sistema de traducción semántica, STT, que cumple tanto los requisitos no funcionales, como los funcionales detallados en apartado 2.1.5 de esta memoria, hasta el nivel de prioridad 3.

Se ha comprobado que es posible integrar transformaciones semánticas en ROS para ser utilizadas en robótica industrial, de forma práctica y robusta, creando una herramienta útil para desarrolladores de diferentes niveles, que además no introduce retrasos significativos en la ejecución de tareas en ROS.

El proyecto implementado ha probado a su vez ser extensible, porque ofrece la posibilidad de añadir nuevas transformaciones de forma sencilla, y porque siguiendo la filosofía modular de ROS, puede ser usado desde otros nodos.

Aunque el código fuente escrito forma parte del software de drag&bot, se valorará por parte del departamento una futura publicación en la comunidad ROS bajo licencia Apache 2.

5.1.1 Integración en drag&bot

La integración de STT con drag&bot no sólo es posible, si no que, al término de la redacción de esta memoria, el equipo de desarrollo de drag&bot ha comenzado ya a integrar el prototipo implementado en este proyecto, creando para él una interfaz web, a modo de asistente para el usuario. Por tanto, se ha comprobado que es factible llamar a los servicios de la aplicación desde el *frontend*.

5.2 Trabajo futuro

El sistema desarrollado ofrece múltiples posibilidades de ampliación según las necesidades de los desarrolladores que lo utilicen, a corto plazo las más interesantes pueden ser:

- Desarrollar una herramienta gráfica, al estilo de *rqt* de ROS, para interactuar con STT.
- Es posible añadir las transformaciones con varios datos de entrada, cambiando el algoritmo de búsqueda en el grafo de transformaciones.
- Crear un paquete de las transformaciones más usadas en robótica, como los cambios entre diferentes formatos de representaciones de las posiciones de un robot.
- Probar el rendimiento de un analizador semántico y estudiar estadísticamente en qué entornos es más conveniente tener la herramienta implementada con uno u otro. Y lo mismo para diferentes tipos de bases de datos.

5.3 Distribución del tiempo

Como se puede apreciar en el diagrama de Gantt de la figura 16, el proyecto requería una primera fase de formación, que se ha mantenido durante la ejecución de gran parte del mismo. Se debe principalmente al aprendizaje de tecnologías como ROS, ROS-Industrial, el lenguaje de programación Python, o el aprendizaje sobre analizadores semánticos, aunque estos fueran descartados para la versión entregada del prototipo en una de las primeras fases de diseño e implementación.

Figura 16: Diagrama de Gantt con distribución de tareas del proyecto

La figura 17 muestra la distribución de horas según las diferentes fases. Es difícil separar la formación de algunas fases del ciclo de diseño e implementación, porque se solapaba aprendizaje con implementación y surgían nuevas opciones a estudiar o nuevos problemas que había que solventar.

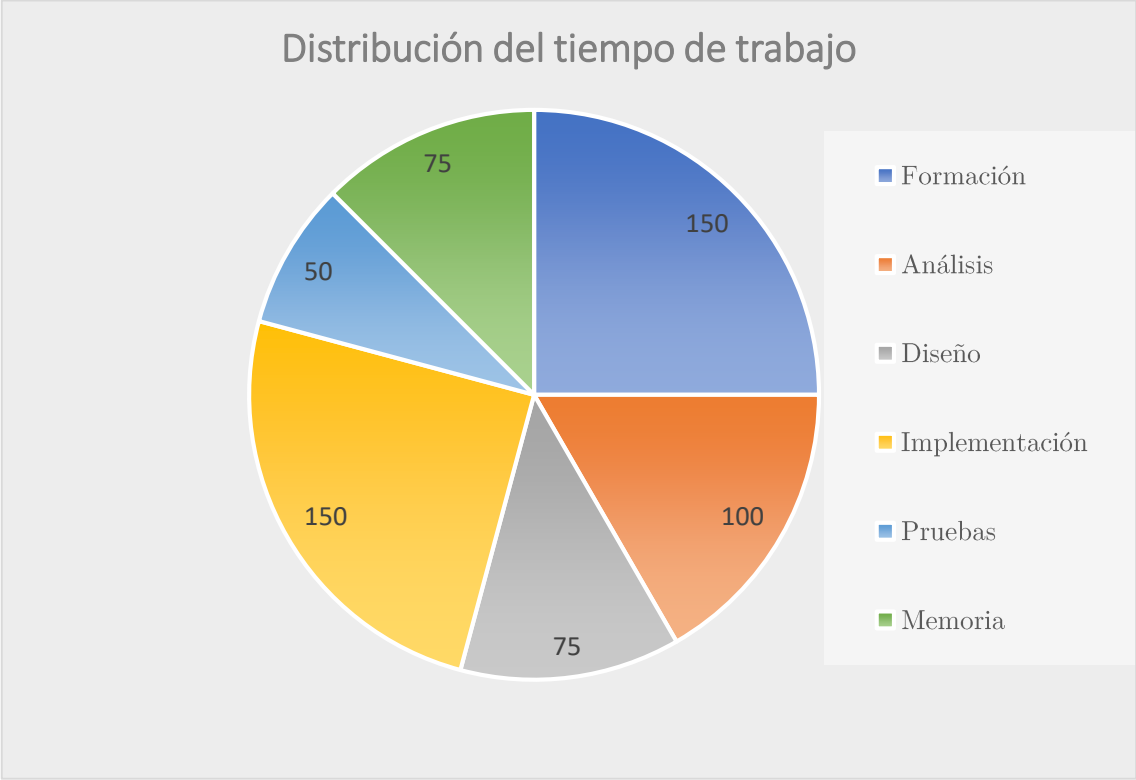


Figura 17: Gráfica circular de distribución de tareas del proyecto

5.4 Valoración personal

Tras un tiempo apartada de la informática, la carrera a la que escogí dedicarme, y a la que por motivos personales no había podido hasta ahora dedicar todos mis esfuerzos, este proyecto ha supuesto un ejercicio de aprendizaje interesante y desafiante, con el añadido de haber conseguido desarrollar una herramienta práctica, que no quedará en el olvido de algunos desarrollos que no pasan del plano teórico. También he podido trabajar en un equipo internacional. Y me ha permitido aprender mucho sobre diversas tecnologías, la situación actual de la robótica industrial, un nuevo lenguaje de programación o los métodos de trabajo que siguen en el Instituto Fraunhofer IPA.

Las principales dificultades vinieron por un lado por el desconocimiento previo del sistema operativo de robótica ROS, y por la dificultad de integración de algunas librerías por incompatibilidades, lo que llegó a suponer cambiar el diseño de una fase de desarrollo para programar directamente las búsquedas de semántica.

La relación con el tutor y la ponente ha sido excelente, resolviendo en todo momento mis dudas sobre cómo debía desarrollar del proyecto, y guiándome en los pasos a realizar para poder sacarlo adelante.

Para finalizar, considero que he podido aplicar los conocimientos de ingeniería informática aprendidos durante la carrera, y aprender nuevas cosas para llevar a cabo un proyecto de ingeniería que ha llegado a término.

Bibliografía

[1] Quilez, P. (2016, June). Semantic Translation Tool for Robotics Applications. In ISR 2016: 47st International Symposium on Robotics; Proceedings of (pp. 1-7). VDE.

[2] Brettel, M., Friederichsen, N., Keller, M., & Rosenberg, M. (2014). How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective. International Journal of Mechanical, Industrial Science and Engineering, 8(1), 37-44.

[3] Schmidt, D. C. (1999). Why software reuse has failed and how to make it work for you. C++ Report, 11(1), 1999.

[4] Naumann, M., Quilez, P., Tobias, S. & Seebauer, D. (2016) Produkt. Recuperado de https://www.dragandbot.com/en (accedido por última vez: 07/09/2017)

[5] Beetz, M., Tenorth, M., & Winkler, J. (2015, May). Open-EASE. In Robotics and Automation (ICRA), 2015 IEEE International Conference on (pp. 1983-1990). IEEE.

[6] Sünderhauf, N., Dayoub, F., McMahon, S., Talbot, B., Schulz, R., Corke, P., ... & Milford, M. (2016, May). Place categorization and semantic mapping on a mobile robot. In Robotics and Automation (ICRA), 2016 IEEE International Conference on (pp. 5729-5736). IEEE.

[7] Zander, S., Heppner, G., Neugschwandtner, G., Awad, A., Essinger & M., Ahmed, N. (2016). A Model-Driven Engineering Approach for ROS using Ontological Semantics.

[8] Larman, C., & Basili, V. R. (2003). Iterative and incremental developments. a brief history. Computer, 36(6), 47-56.

[9] Fernández, E., Sánchez Crespo, L., Mahtani, A, & Martínez, A. Learning ROS for Robotics Programming (second Edition). Birmigham, UK: Packt Publishing Ltd.

[10] Toris, R., (2015) The Standard ROS JavaScript Library. Recuperado de http://wiki.ros.org/roslibjs (accedido por última vez: 07/09/2017)

[11] Toris, R., Kammerl, J., Lu, D., Lee, J., Jenkins, O.C., Osentoski, S., Wills, M. & Chernova, S. Robot Web Tools: Efficient Messaging for Cloud Robotics. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2015.

[12] Livingston, S., (2015) rosbridge_suite. Recuperado de http://wiki.ros.org/rosbridge_suite

[13] Baal, A., rosbridge v2.0 Protocol Specification (2013). Recuperado de https://github.com/RobotWebTools/rosbridge_suite/blob/groovy-devel/ROSBRIDGE_PROTOCOL.md (accedido por última vez: 07/09/2017)

[14] Ben-Kiki, O., Evans, C., & Ingerson, B. (2005). YAML Ain't Markup Language (YAML™) Version 1.1. yaml. org, Tech. Rep.

[15] Bechhofer, S. (2009). OWL: Web ontology language. In Encyclopedia of Database Systems (pp. 2008-2009). Springer US.

[16] Álvarez García, S. (2014). Compact and efficient representations of graphs.

[17] Shearer, R., Motik, B., & Horrocks, I. (2008, October). HermiT: A Highly-Efficient OWL Reasoner. In OWLED (Vol. 432, p. 91).

[18] Parsia, B., & Sirin, E. (2004, November). Pellet: An owl dl reasoner. In Third international semantic web conference-poster (Vol. 18, p. 13).

[19] Bobed, C., Yus, R., Bobillo, F., & Mena, E. (2015). Semantic reasoning on mobile devices: Do Androids dream of efficient reasoners?. Web Semantics: Science, Services and Agents on the World Wide Web, 35, 167-183.

[20] Mitchell, R. J. (1993). Introduction. In C++ Object-Oriented Programming (pp. 1-15). Macmillan Education UK.

[21] Sanner, M. F. (1999). Python: a programming language for software integration and development. J Mol Graph Model, 17(1), 57-61.

[22] Lamy, J.B., (2017), Owlready 0.3.1. Recuperado de https://pypi.python.org/pypi/Owlready (accedido por última vez: 07/09/2017)

[23] Deacon, J. (2009). Model-view-controller (mvc) architecture. Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf. (accedido por última vez: 07/09/2017)

[24] Kay, J. & Tsouroukdissian, A. R. (2015) Real-time control in ROS and ROS 2.0 Recuperado de https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf (accedido por última vez: 07/09/2017)

A.1. Categorías semánticas

En la siguientes tablas se muestran ejemplos de representación de las categorías semánticas más relevantes en el ámbito de ROS-Industrial y sus posibles formatos en ROS.

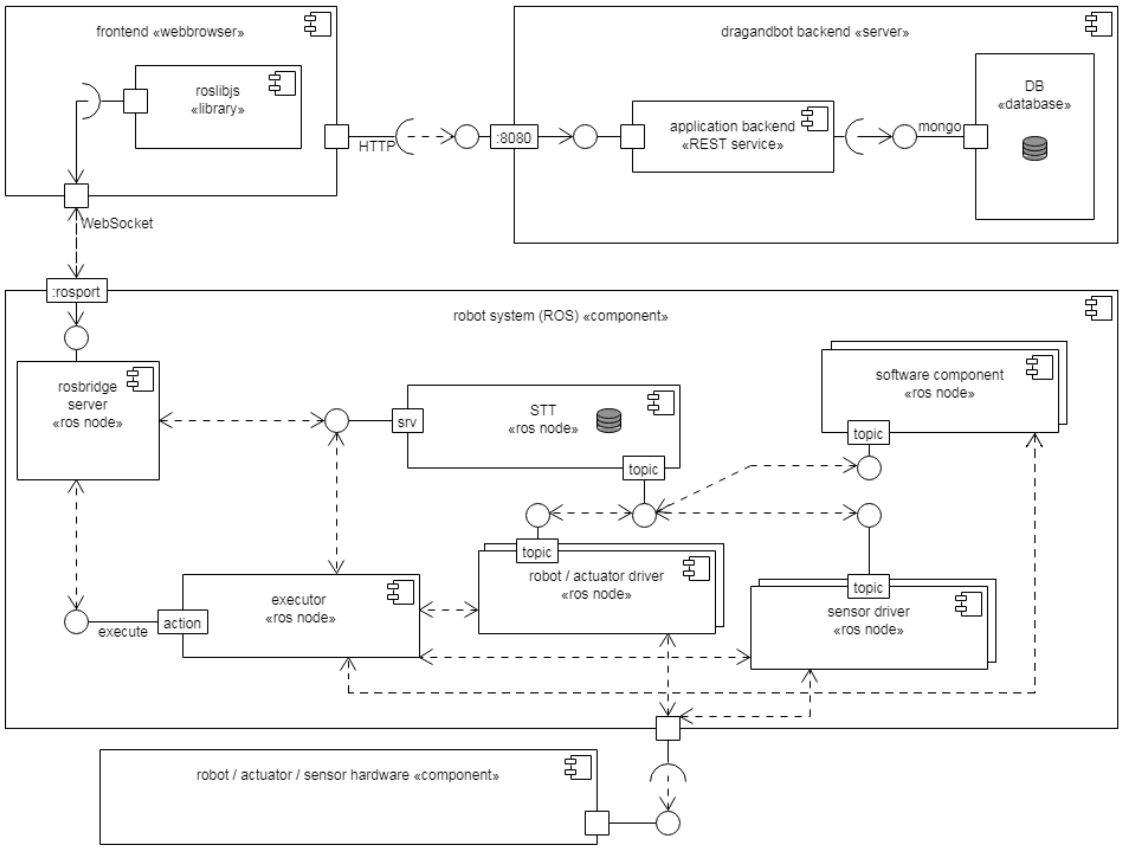
Categoría semántica	**Formato ROS**
Distancias:	
/world/geometry/distance/mm	std_msgs/Float64
/world/geometry/distance/cm	std_msgs/Float64
/world/geometry/distance/dm	std_msgs/Float64
/world/geometry/distance/m	std_msgs/Float64
/world/geometry/distance/dam	std_msgs/Float64
/world/geometry/distance/hm	std_msgs/Float64
/world/geometry/distance/km	std_msgs/Float64
/world/geometry/distance/mm	std_msgs/Float32
/world/geometry/distance/cm	std_msgs/Float33
/world/geometry/distance/dm	std_msgs/Float34
/world/geometry/distance/m	std_msgs/Float35
/world/geometry/distance/dam	std_msgs/Float36
/world/geometry/distance/hm	std_msgs/Float37
/world/geometry/distance/km	std_msgs/Float38
/world/geometry/distance/mm	std_msgs/Int64
/world/geometry/distance/cm	std_msgs/Int65
/world/geometry/distance/dm	std_msgs/Int66
/world/geometry/distance/m	std_msgs/Int67
/world/geometry/distance/dam	std_msgs/Int68
/world/geometry/distance/hm	std_msgs/Int69
/world/geometry/distance/km	std_msgs/Int70
/world/geometry/distance/mm	std_msgs/Int32
/world/geometry/distance/cm	std_msgs/Int33
/world/geometry/distance/dm	std_msgs/Int34
/world/geometry/distance/m	std_msgs/Int35
/world/geometry/distance/dam	std_msgs/Int36
/world/geometry/distance/hm	std_msgs/Int37
/world/geometry/distance/km	std_msgs/Int38

Categoría semántica	Formato ROS
Posiciones:	
/world/geometry/pose	geometry_msgs/Pose
/world/geometry/pose/position/m	geometry_msgs/Point
/world/geometry/pose/position/cm	geometry_msgs/Point
/world/geometry/pose/position/mm	geometry_msgs/Point
/world/geometry/pose2d	geometry_msgs/Pose2D
/world/geometry/pose2d/position/m	dragandbot_msgs/point2
/world/geometry/pose2d/position/cm	dragandbot_msgs/point2
/world/geometry/pose2d/position/mm	dragandbot_msgs/point2
/world/geometry/pose/orientation/quaternion	geometry_msgs/Quaternion
/world/geometry/pose/euler	robot_movement_interface/E
/world/geometry/pose/orientation/euler/intrinsic/zyx/rad	geometry_msgs/Vector3
/world/geometry/pose/orientation/euler/intrinsic/xyz/rad	geometry_msgs/Vector3
/world/geometry/pose/orientation/euler/extrinsic/zyx/rad	geometry_msgs/Vector3
/world/geometry/pose/orientation/euler/extrinsic/xyz/rad	geometry_msgs/Vector3
/world/geometry/pose/orientation/euler/intrinsic/zyx/degrees	geometry_msgs/Vector3
/world/geometry/pose/orientation/euler/intrinsic/xyz/degrees	geometry_msgs/Vector3
/world/geometry/pose/orientation/euler/extrinsic/zyx/degrees	geometry_msgs/Vector3
/world/geometry/pose/orientation/euler/extrinsic/xyz/degrees	geometry_msgs/Vector3
Posición basada en joints (ángulos de giro de los motores del robot):	
/world/geometry/pose/joints/rad	sensor_msgs/JointState
/world/geometry/pose/joints/degrees	sensor_msgs/JointState
Variables físicas:	
/world/physics/speed/lineal/vector/m_s	geometry_msgs/Vector3
/world/physics/speed/lineal/m_s	std_msgs/Float64
/world/physics/speed/lineal/m_s	std_msgs/Float34
/world/physics/speed/lineal/m_s	std_msgs/Int66
/world/physics/speed/lineal/m_s	std_msgs/Int34
/world/physics/speed/angular/vector/m_s	geometry_msgs/Vector3
/world/physics/speed/angular/rad_s	std_msgs/Float64
/world/physics/speed/angular/rad_s	std_msgs/Float34
/world/physics/speed/angular/rad_s	std_msgs/Int66
/world/physics/speed/angular/rad_s	std_msgs/Int34
/world/physics/acceleration/lineal/vector/m_s2	geometry_msgs/Vector3
/world/physics/acceleration/lineal/m_s2	std_msgs/Float64
/world/physics/acceleration/lineal/m_s2	std_msgs/Float34
/world/physics/acceleration/lineal/m_s2	std_msgs/Int66
/world/physics/acceleration/lineal/m_s2	std_msgs/Int34
/world/physics/acceleration/angular/vector/rad_s2	geometry_msgs/Vector3
/world/physics/acceleration/angular/rad_s2	std_msgs/Float64
/world/physics/acceleration/angular/rad_s2	std_msgs/Float34
/world/physics/acceleration/angular/rad_s2	std_msgs/Int66
/world/physics/acceleration/angular/rad_s2	std_msgs/Int34

Categoría semántica	**Formato ROS**
/world/physics/temperature/celcius	std_msgs/Float64
/world/physics/temperature/celcius	std_msgs/Float34
/world/physics/temperature/celcius	std_msgs/Int66
/world/physics/temperature/celcius	std_msgs/Int34
/world/physics/temperature/fahrenheit	std_msgs/Float64
/world/physics/temperature/fahrenheit	std_msgs/Float34
/world/physics/temperature/fahrenheit	std_msgs/Int66
/world/physics/temperature/fahrenheit	std_msgs/Int34
/world/physics/temperature/kelvin	std_msgs/Float64
/world/physics/temperature/kelvin	std_msgs/Float34
/world/physics/temperature/kelvin	std_msgs/Int66
/world/physics/temperature/kelvin	std_msgs/Int34
/world/physics/inertia	geometry_msgs/Inertia
Tiempos:	
/world/time/duration/seconds	std_msgs/Duration
/world/time/duration/seconds	std_msgs/Float64
/world/time/duration/milliseconds	std_msgs/Duration
/world/time/duration/milliseconds	std_msgs/Float65
/world/time/duration/nanoseconds	std_msgs/Duration
/world/time/duration/nanoseconds	std_msgs/Float66
/world/time/date/ros	std_msgs/Time
Imágenes:	
/world/vision/image/2d/compressed_ros_image	sensor_msgs/CompressedImag
/world/vision/image/2d/ros_image	sensor_msgs/Image
/world/vision/image/2d/jpg	dragandbot_msgs/jpg
/world/vision/image/2d/png	dragandbot_msgs/png
/world/vision/image/2d/bmp	dragandbot_msgs/bmp
/world/vision/image/3d/pointcloud	sensor_msgs/PointCloud
/world/vision/image/3d/pointcloud2	sensor_msgs/PointCloud2
Estado del sistema:	
/world/status/robot	industrial_msgs/RobotStatus
/world/status/gripper	common_msgs/GripperStatus
/world/status/robot/rmi/result	robot_movement_interface/R
/world/status/sensor/light	std_msgs/Bool
Control del robot:	
/world/command/rmi/list	robot_movement_interface/C
/world/command/rmi/command	robot_movement_interface/C
/world/command/joint_trajectory	trajectory_msgs/JointTrajecto
/world/command/joint_trajectory_point	trajectory_msgs/JointTrajecto

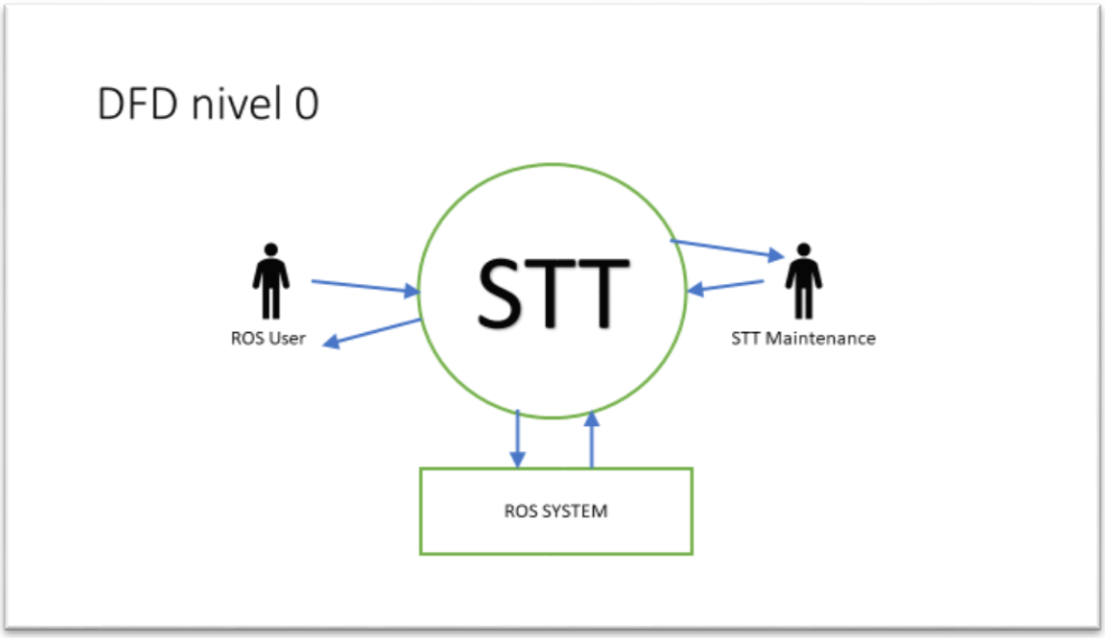
A.2. Diagrama de componentes de drag&bot

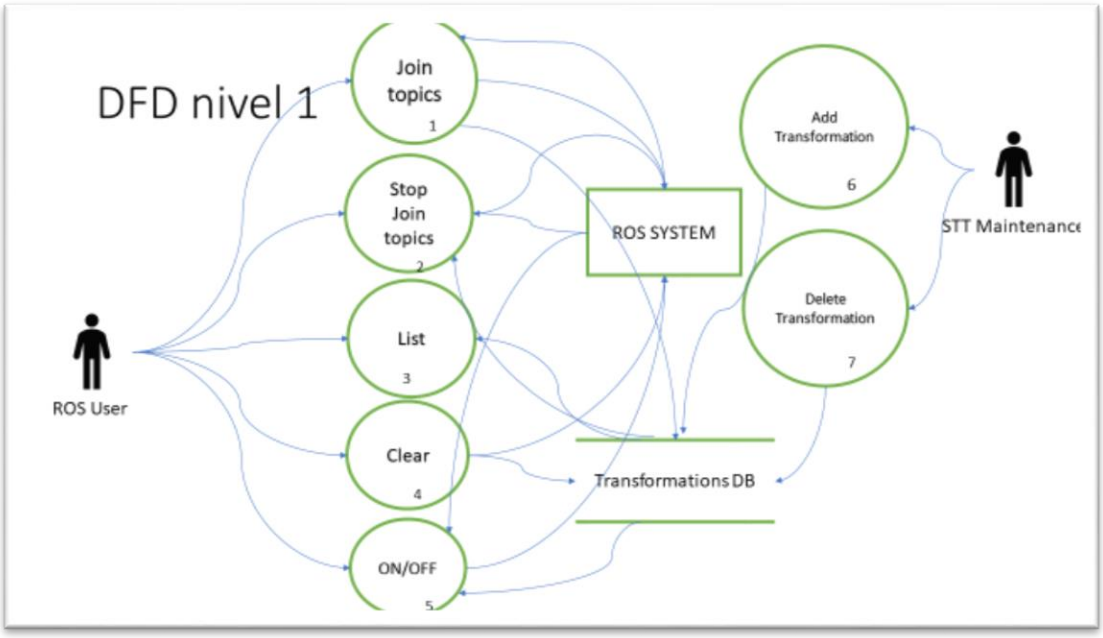
En este anexo se presenta un diagrama de componentes de drag&bot más completo al presentado en la sección 3.2 de la memoria. Se ha añadido además la herramienta STT tal y como se integra en drag&bot.

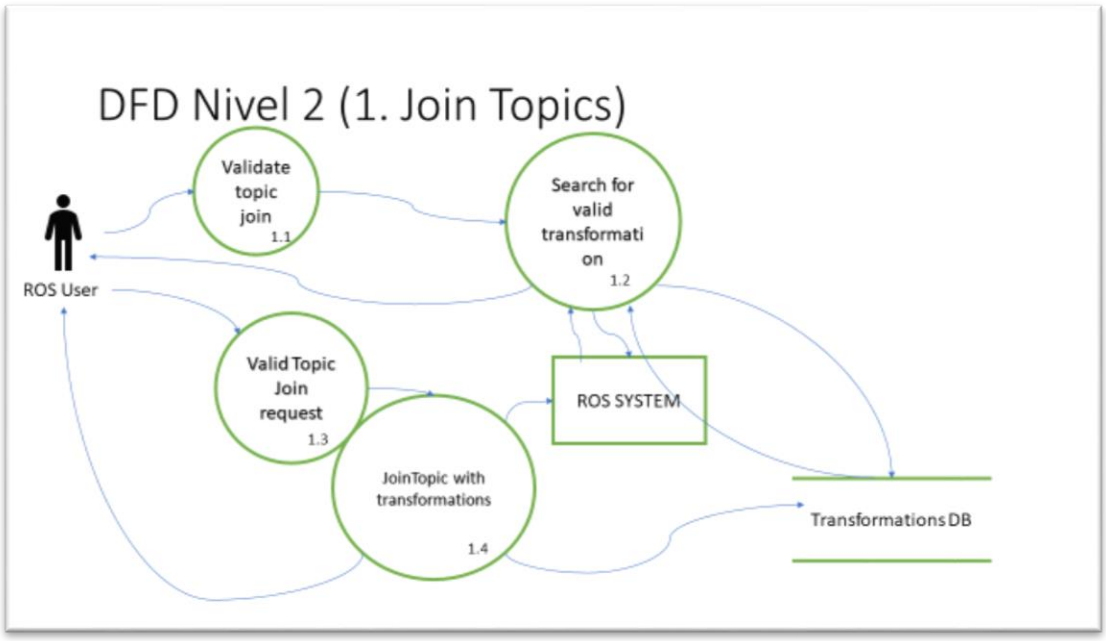


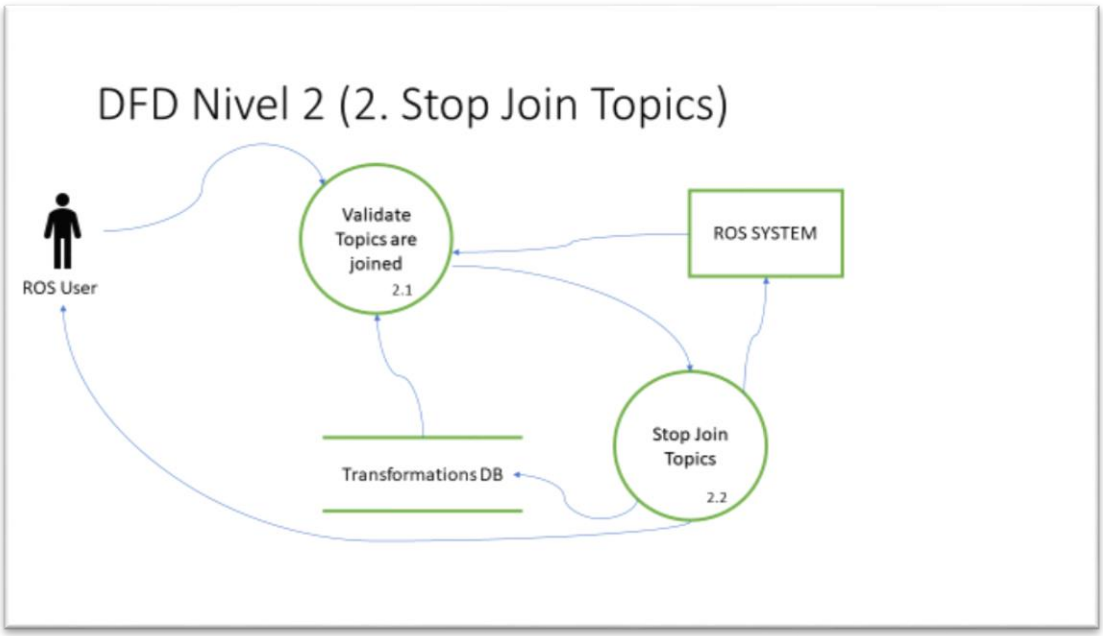
A.3. Diagramas de flujo de datos

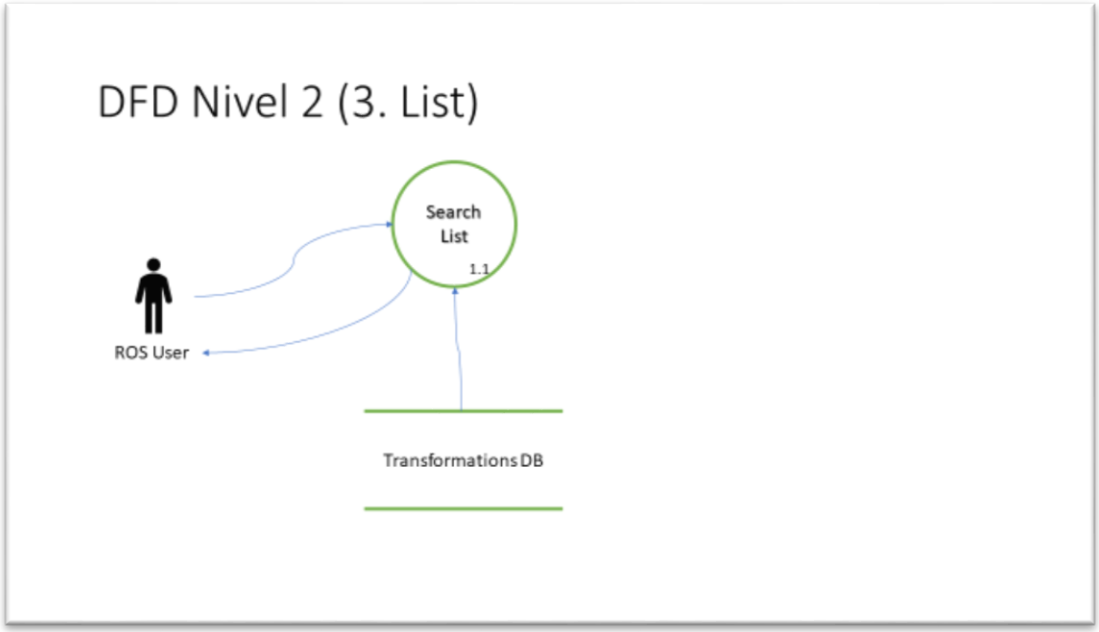
En fase de análisis se diseñaron los siguientes Diagramas de flujo de datos de los casos de uso más representativos, usando como referencia para las conexiones los *topics* 1 a 1.

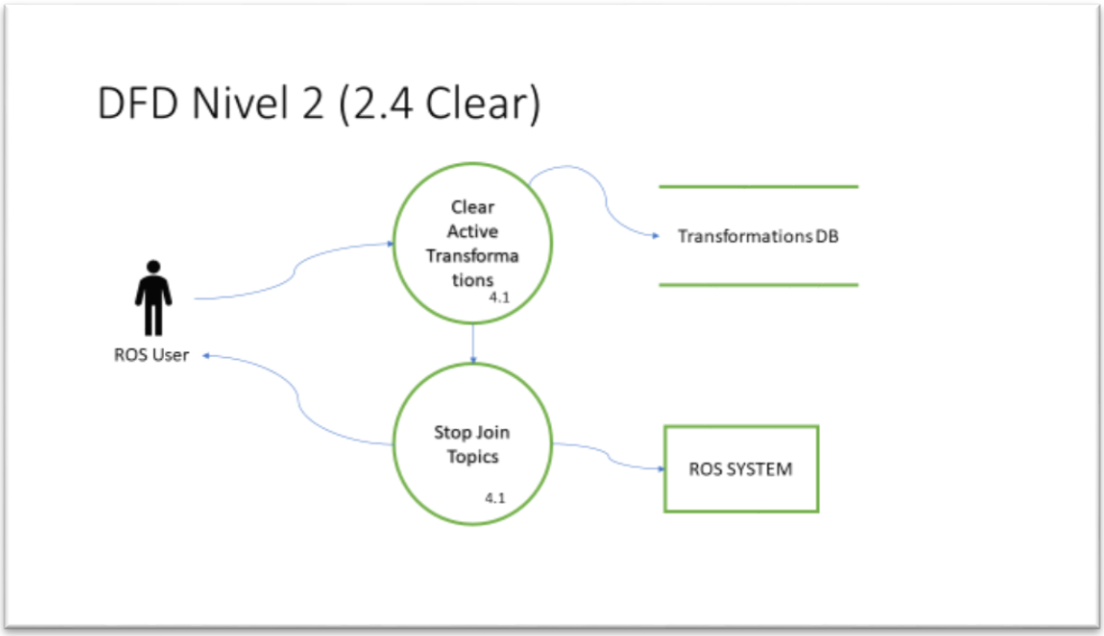


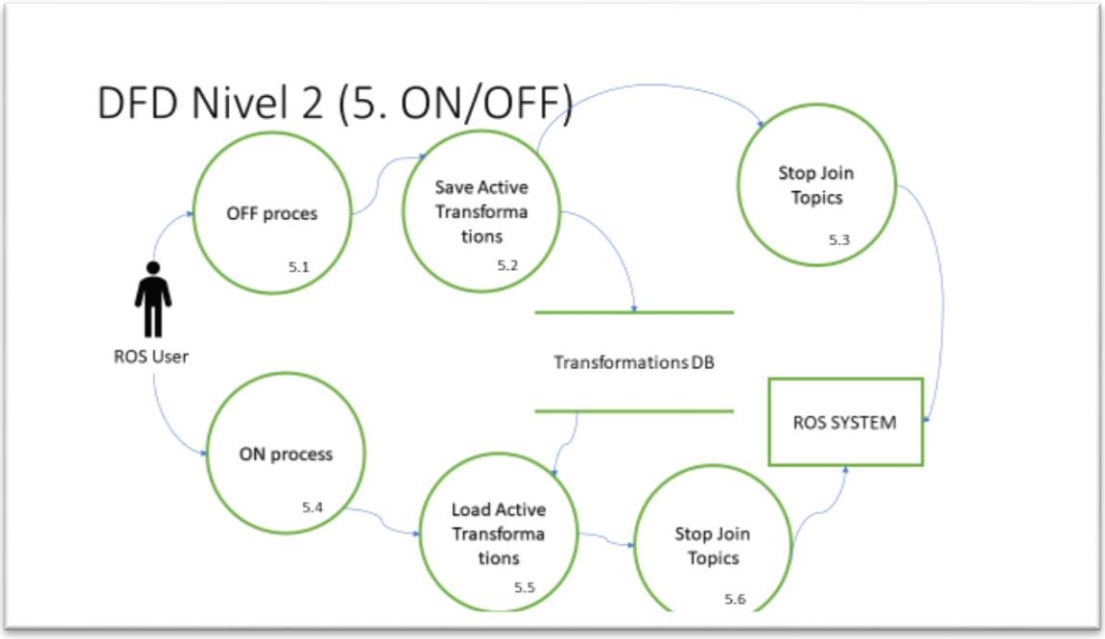


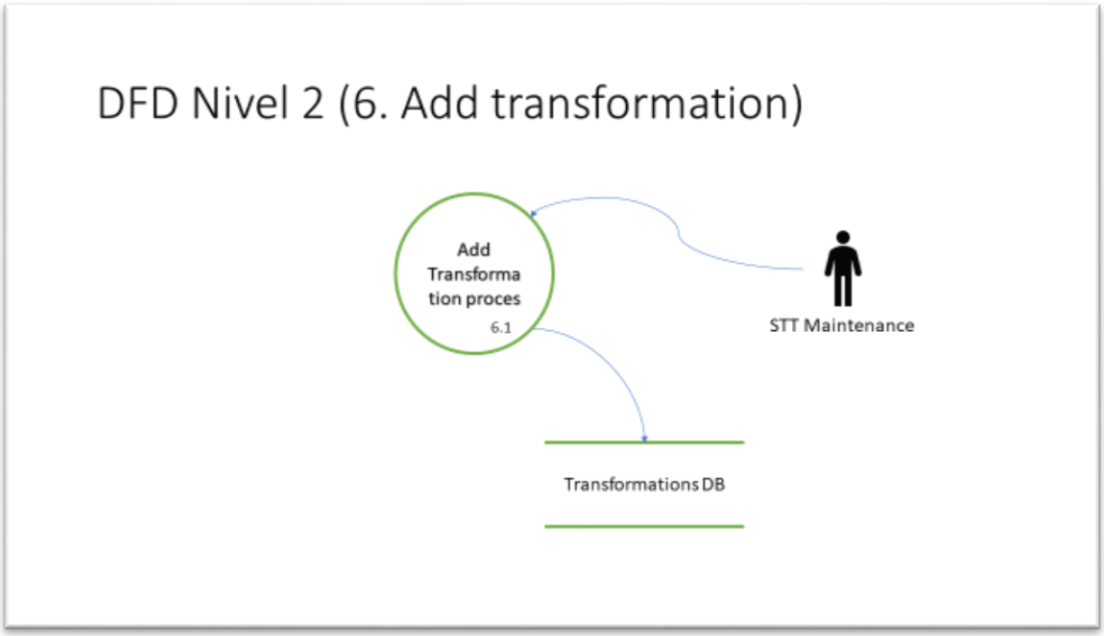

DFD Nivel 2 (1. Join Topics)
Validate topic join
1.1
Search for valid transformati on
1.2
ROS User
Valid Topic Join request
1.3
ROS SYSTEM
JoinTopic with transformations
1.4
Transformations DB

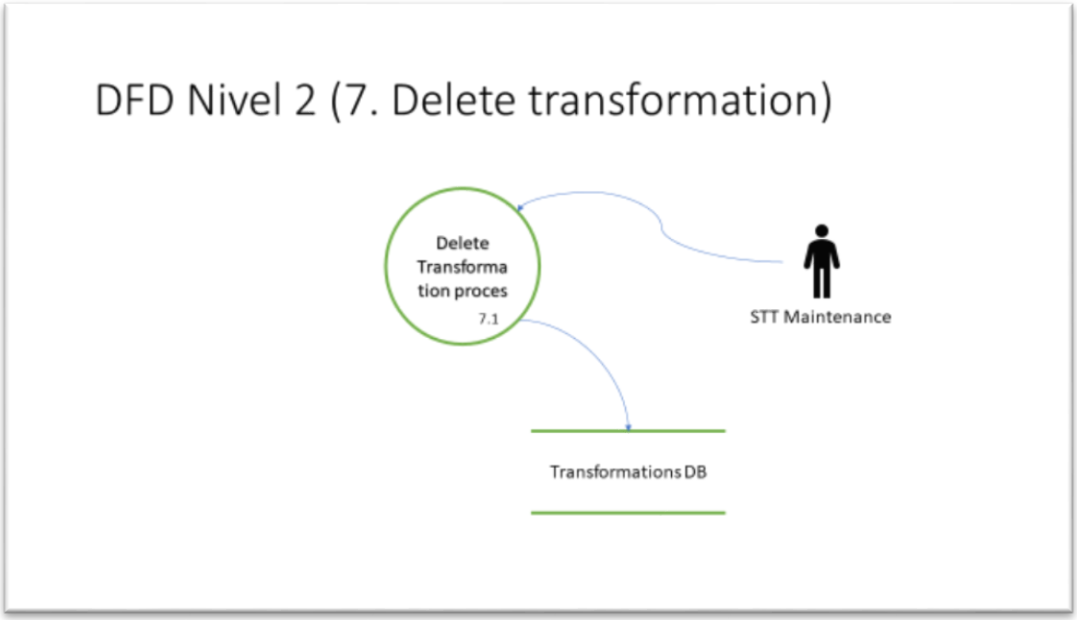

DFD Nivel 2 (2. Stop Join Topics)
ROS User
Validate Topics are joined
2.1
ROS SYSTEM
Transformations DB
Stop Join Topics
2.2


DFD Nivel 2 (3. List)
Search List
1.1
ROS User
Transformations DB


DFD Nivel 2 (2.4 Clear)
Clear Active Transforma tions
4.1
Transformations DB
ROS User
Stop Join Topics
4.1
ROS SYSTEM

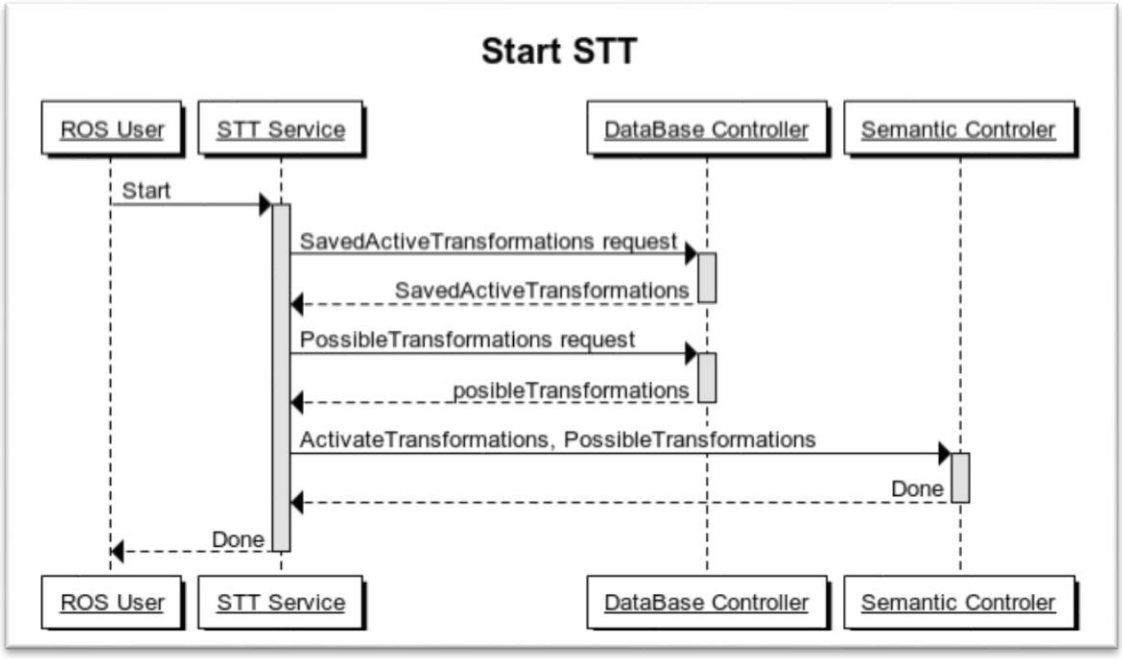

DFD Nivel 2 (5. ON/OFF)
OFF proces
5.1
Save Active Transforma tions
5.2
Stop Join Topics
5.3
ROS User
Transformations DB
ROS SYSTEM
ON process
5.4
Load Active Transforma tions
5.5
Stop Join Topics
5.6

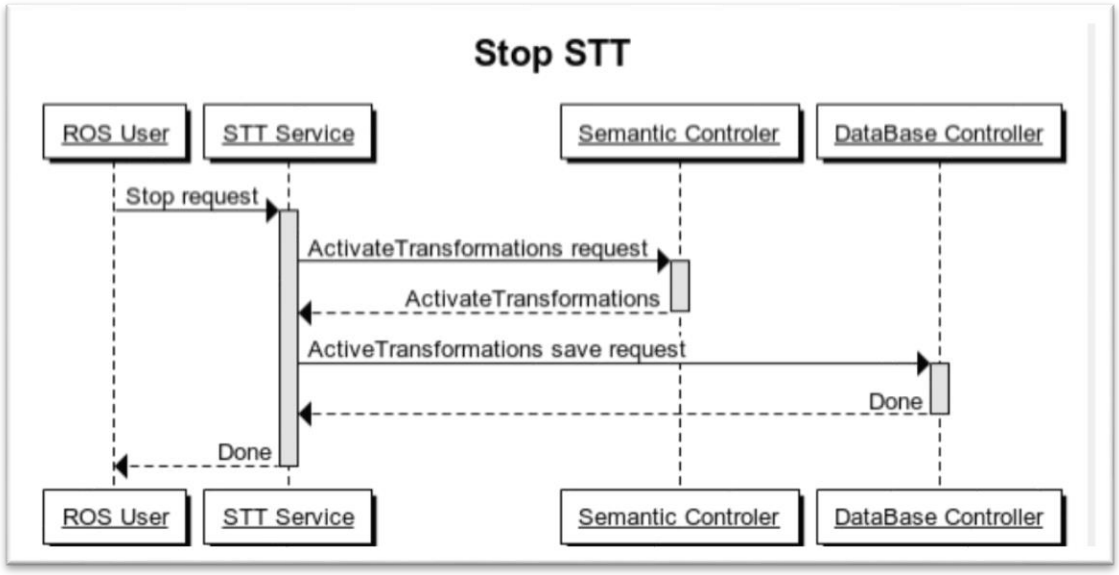

DFD Nivel 2 (6. Add transformation)
Add Transforma tion proces
6.1
STT Maintenance
Transformations DB

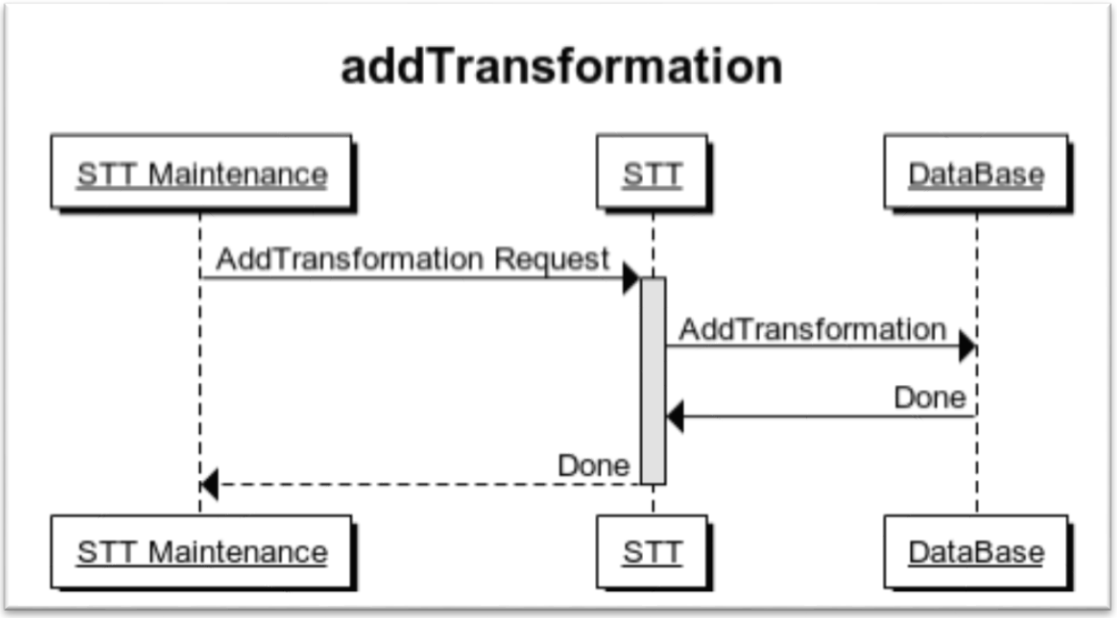

DFD Nivel 2 (7. Delete transformation)
Delete Transforma tion proces
7.1
STT Maintenance
Transformations DB

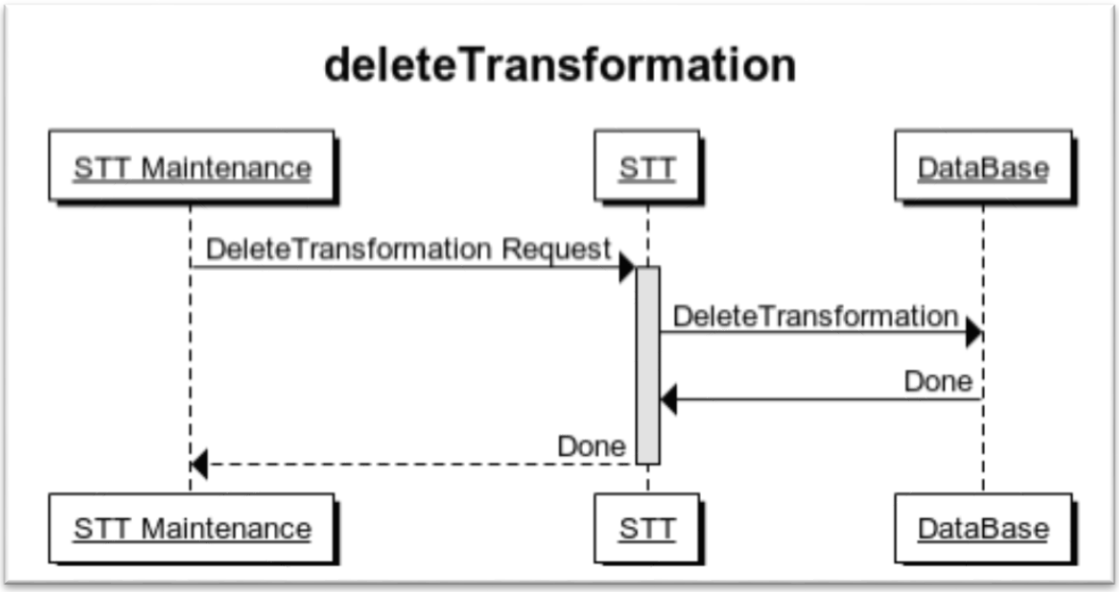
A.4. Diagramas de secuencia

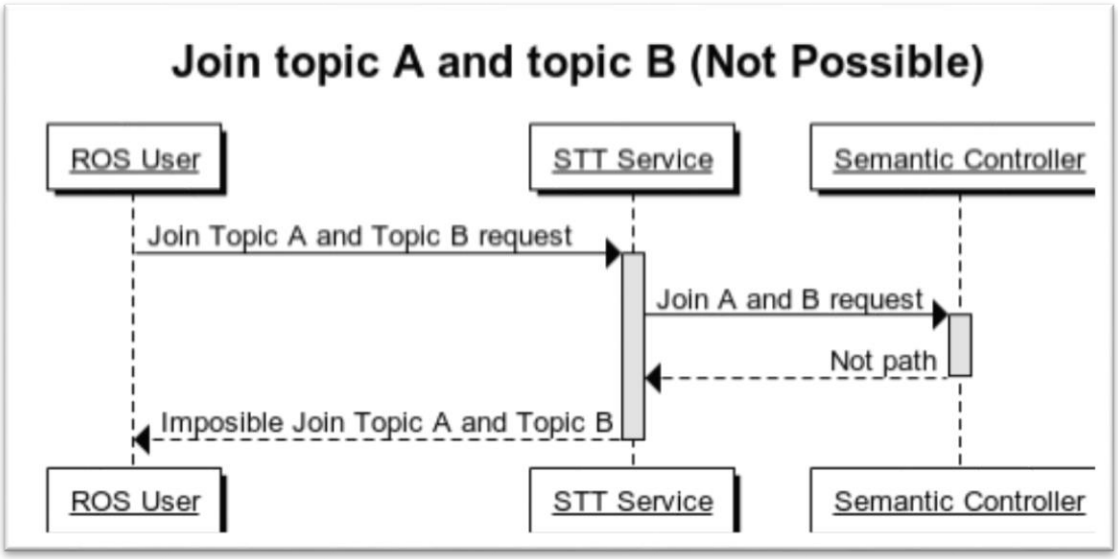
En fase de análisis se diseñaron los siguientes Diagramas de secuencia de los casos de uso más representativos, usando como referencia para las conexiones los *topics* 1 a 1.

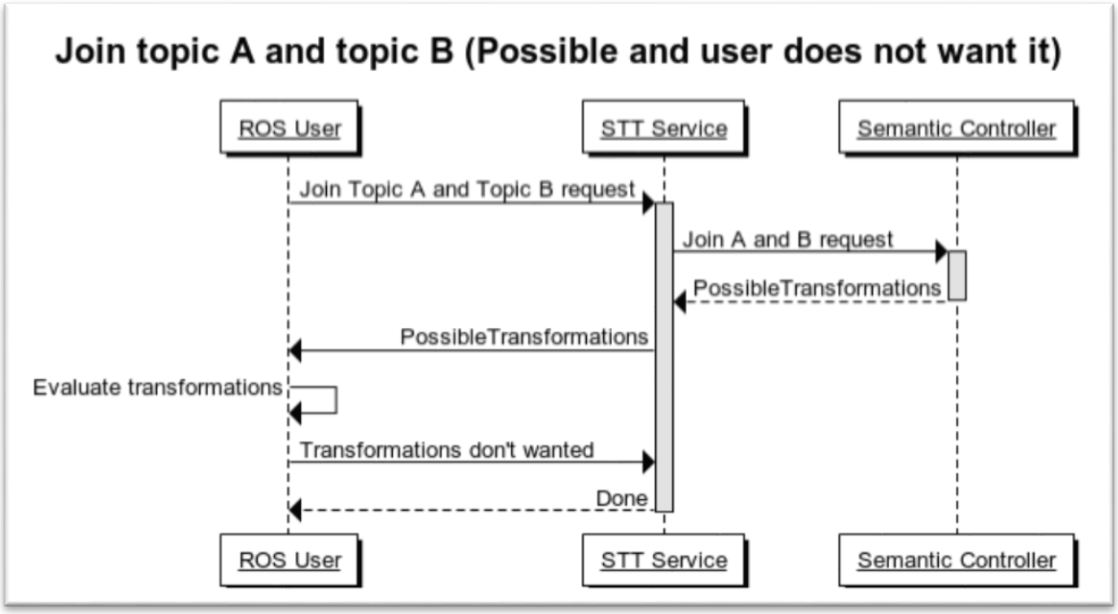


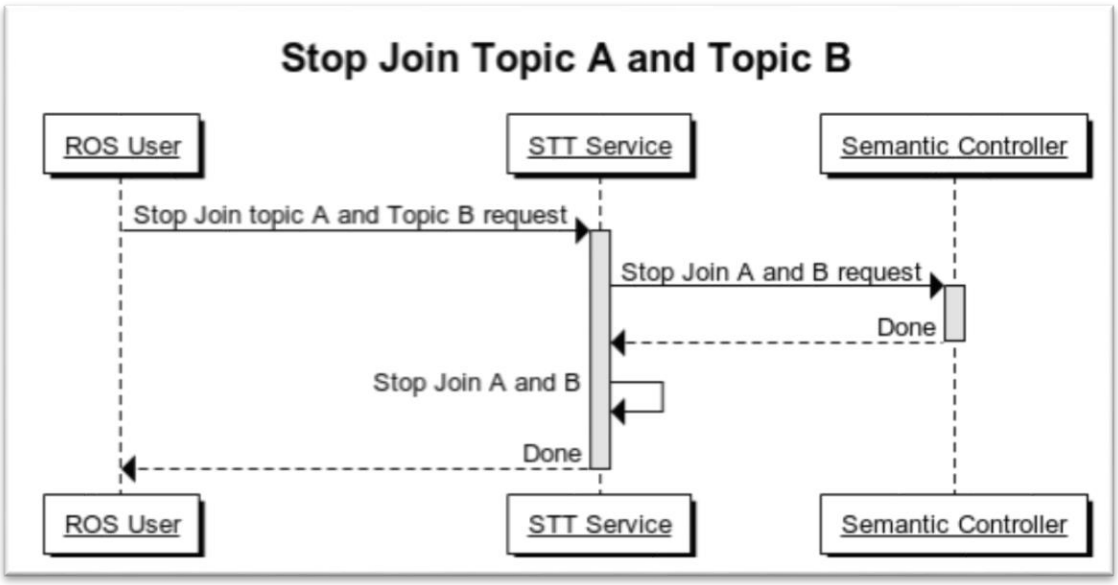


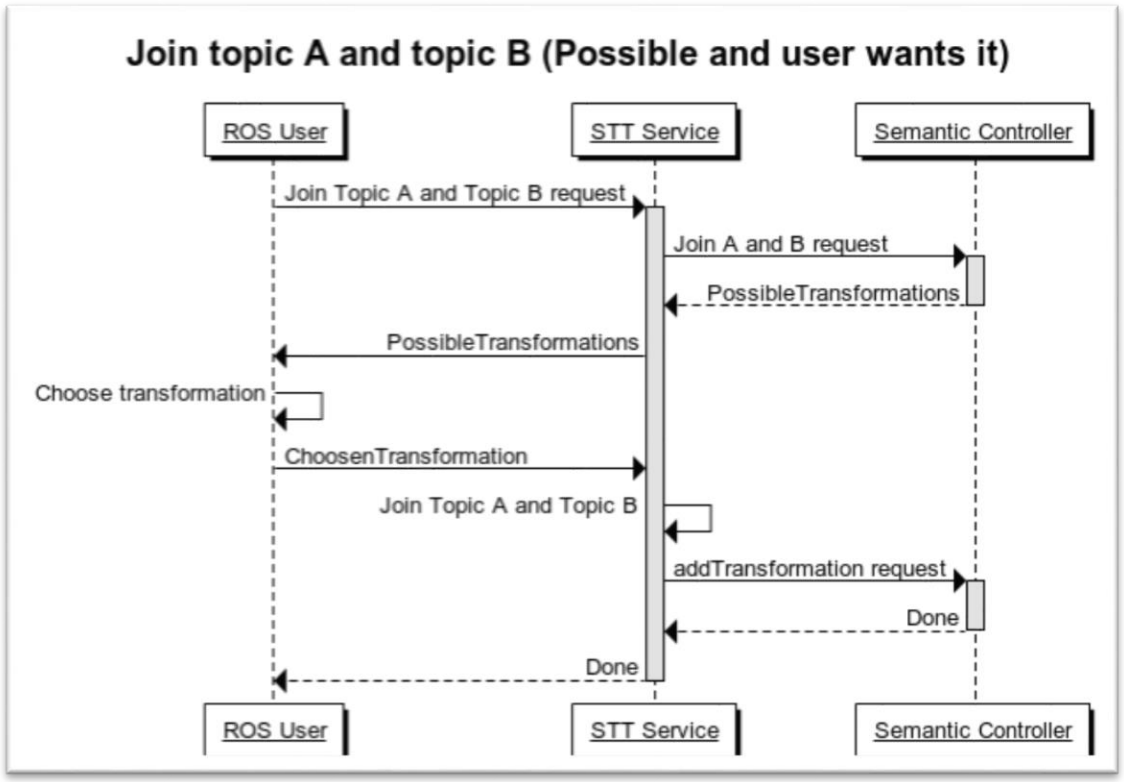

addTransformation
STT Maintenance
STT
DataBase
AddTransformation Request
AddTransformation
Done
Done
STT Maintenance
STT
DataBase

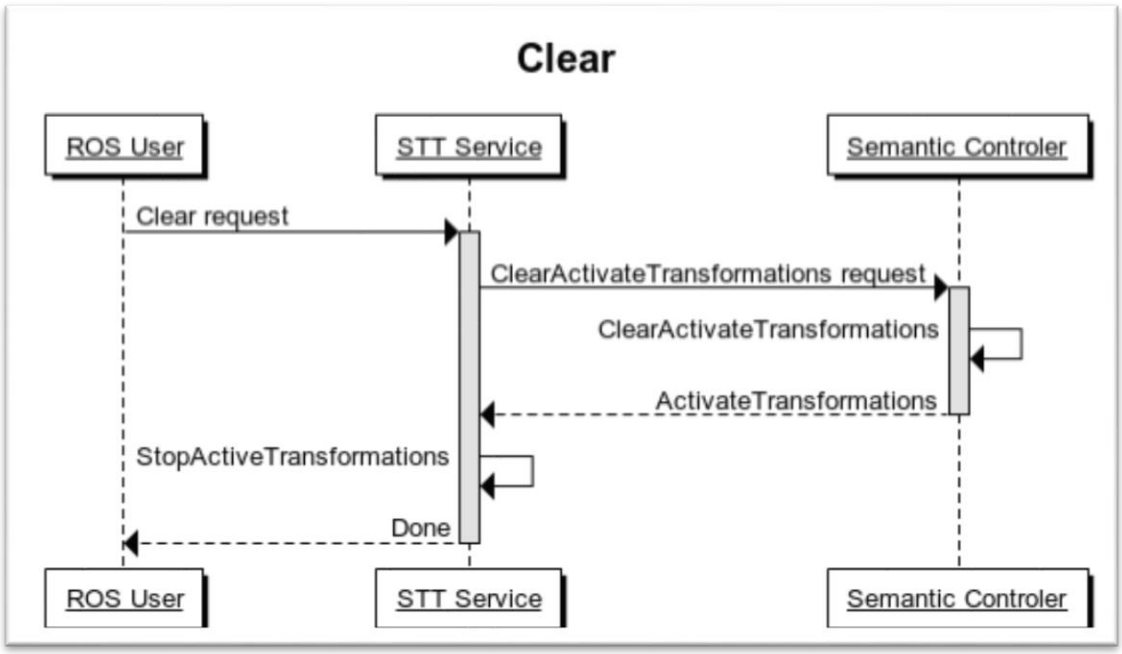

deleteTransformation
STT Maintenance
STT
DataBase
DeleteTransformation Request
DeleteTransformation
Done
Done
STT Maintenance
STT
DataBase

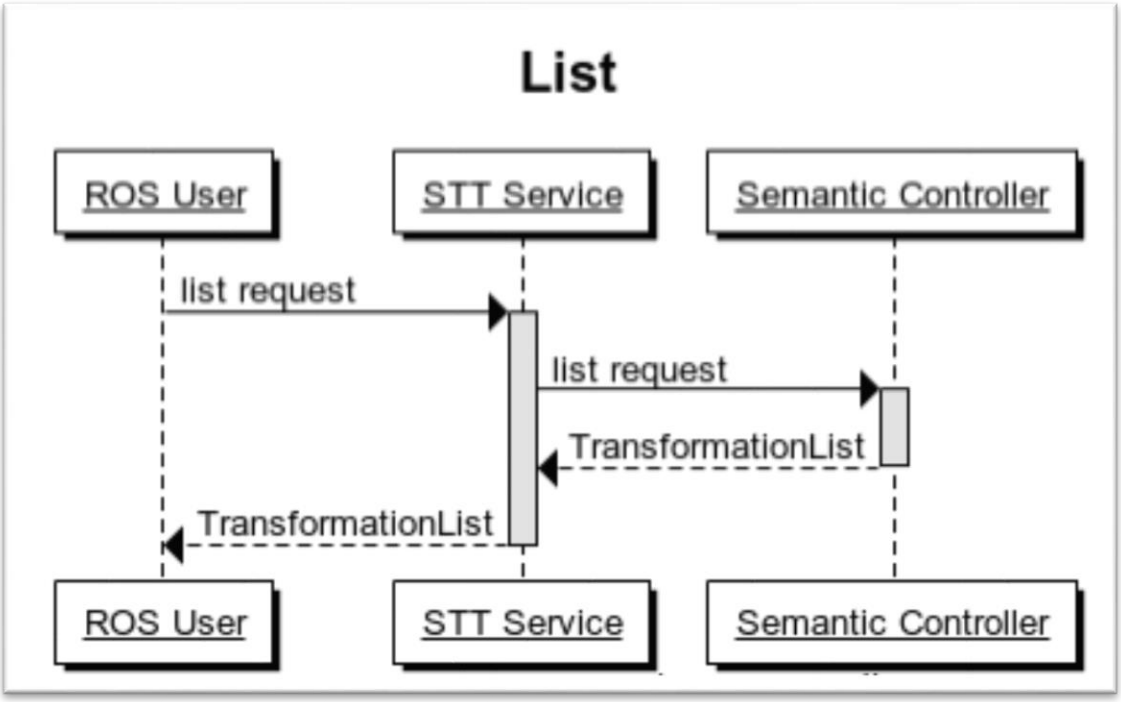

Join topic A and topic B (Not Possible)
ROS User
STT Service
Semantic Controller
Join Topic A and Topic B request
Join A and B request
Not path
Imposible Join Topic A and Topic B
ROS User
STT Service
Semantic Controller


Join topic A and topic B (Possible and user does not want it)
ROS User
STT Service
Semantic Controller
Join Topic A and Topic B request
Join A and B request
PossibleTransformations
PossibleTransformations
Evaluate transformations
Transformations don't wanted
Done
ROS User
STT Service
Semantic Controller


Stop Join Topic A and Topic B
ROS User
STT Service
Semantic Controller
Stop Join topic A and Topic B request
Stop Join A and B request
Done
Stop Join A and B
Done
ROS User
STT Service
Semantic Controller


Join topic A and topic B (Possible and user wants it)
ROS User
STT Service
Semantic Controller
Join Topic A and Topic B request
Join A and B request
PossibleTransformations
PossibleTransformations
Choose transformation
ChoosenTransformation
Join Topic A and Topic B
addTransformation request
Done
Done
ROS User
STT Service
Semantic Controller


Clear
ROS User
STT Service
Semantic Controler
Clear request
ClearActivateTransformations request
ClearActivateTransformations
ActivateTransformations
StopActiveTransformations
Done
ROS User
STT Service
Semantic Controler


List
ROS User
STT Service
Semantic Controller
list request
list request
TransformationList
TransformationList
ROS User
STT Service
Semantic Controller

A.5. Listado completo de archivos y directorios del STT

En este anexo se muestra el árbol de archivos y directorios del STT en su última versión, incluyendo los archivos necesarios para las pruebas.

```
STT
│   CMakeLists.txt
│   package.xml
│
├──db
│       available_transformations.db
│       semantic_categories.db
│       topic_semantic_information.db
│
├──include
│   └──stt
├──msg
│       Category.msg
│       Connection.msg
│       TopicInformation.msg
│       Transformator.msg
│
├──scripts
│   │   stt_controller.py
│   │   stt_mixer.py
│   │   stt_model.py
│   │   stt_searcher_topics.py
│   │   stt_server.py
│   │   test_beep.py
│   │   test_call_join.py
│   │   test_call_join_multiple.py
│   │   test_delay_monitor.py
│   │   test_delay_monitor_multiple.py
│   │   test_many_topics.py
│   │   test_multiple_topic_publisher.py
│   │   test_simple_publisher.py
│   │   test_simple_suscriber.py
│   │
│   └──transformations
│       │   ___init___.py
│       │
│       ├──dam_to_km
│       │       ___init___.py
│       │
│       ├──mm_to_m
│       │       ___init___.py
│       │
│       ├──m_to_dam
│       │       ___init___.py
```

```
│       │
│       ├──m_to_km
│       │       __init__.py
│       │
│       └──m_to_mm
│               __init__.py
│
├──src
├──srv
│       AddAvaibleTransformation.srv
│       AddSemanticCategory.srv
│       AddTopicSemanticInformation.srv
│       JoinTopics.srv
│       JoinTopicsTransformations.srv
│       ListAvaibleTransformations.srv
│       ListPossibleTransformationSources.srv
│       ListSemanticCategories.srv
│       ListTopicSemanticInformations.srv
│       RemoveAvaibleTransformation.srv
│       RemoveSemanticCategory.srv
│       RemoveTopicSemanticInformation.srv
│       StopJoin.srv
│       STTServiceType.srv
│
└──tests
        commands.txt
        integration_1_semantic_categories.sh
        integration_2_topic_semantic_information.sh
        integration_3_transformations.sh
        integration_4_search.sh
        integration_5_join.sh
        populate_semantic_categories.sh
        test_performance_1.txt
```