UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Un enfoque dirigido por modelos para la creación de sistemas robóticos con misión predeterminada.
Autores: Mattone, Nicolas y Montanari, Franco.
Director: Dra. Giandini, Roxana.
Codirector: Dra. Pons, Claudia.
Asesor profesional: Lic. Pérez, Gabriela.
Carrera: Licenciatura en Informática.

Resumen

El objetivo de esta tesina es aplicar los conceptos del Desarrollo de Software Dirigido por Modelos (del inglés, Model Driven Development, MDD) a la creación de sistemas robóticos. Puntualmente, se define un lenguaje específico de dominio o DSL (del inglés, Domain Specific Language) para el siguiente problema: Un robot recorre un espacio conocido con anterioridad. Ese recorrido consiste en llegar a un conjunto ordenado de ubicaciones de ese espacio. Cuando el robot llega a una ubicación, puede ejecutar (o no) una serie de acciones. Una vez finalizada la ejecución de la/s accione/s en una ubicación, el robot continúa con el recorrido, trasladándose hacia la siguiente ubicación del conjunto. Luego, se presentan las transformaciones para generar el código fuente a partir del DSL. Esta generación, que transforma un modelo abstracto a código fuente específico de la plataforma, es automática (es decir, sin intervención humana). Como plataforma específica se utiliza el framework ROS (Robot Operating System). Por último, se construye una herramienta gráfica que permite especificar una instancia del DSL y generar el código a partir de ésta.

Palabras Claves

Model Driven Development, Domain Specific Language, Robótica, Eclipse IDE, Ecore, Metamodelos, Modelos, Transformación modelo a texto, Eclipse Modeling Framework, Acceleo, Robot Operating System.

Conclusiones

Gracias a los avances en el campo y a las herramientas disponibles en el mercado, se logró definir un DSL para robots con misión predeterminada, desarrollar con una herramienta gráfica y ejecutar con éxito programas generados automáticamente en un robot real. MDD ha demostrado ser un paradigma que puede ser aplicado y utilizado de forma práctica, con resultados reales y concretos. El trabajo de esta tesina sirve para demostrar esta afirmación.

Trabajos Realizados

Temas de estudio: MDD. Conceptos y problemas comunes del dominio de la robótica. Casos similares.
Análisis de herramientas: Eclipse, EMF, Acceleo, ROS.
Definición de un lenguaje específico de domino: DSL para robots con misión predeterminada.
Implementación de generadores de código, a partir de modelos definidos con el mencionado DSL, con sus respectivas demostraciones en un robot simulado y en un robot real.
Desarrollo de una herramienta gráfica.

Trabajos Futuros

Mejorar la herramienta gráfica: se propone migrar la aplicación desarrollada en Java a un sistema web
Extender el metamodelo del DSL para robots con misión predeterminada: Se pretende identificar más acciones comunes a cualquier sistema robótico.
Implementar una generación de código para otras plataformas específicas diferentes a ROS.
Restringir el metamodelo del DSL para robots con misión predeterminada a un dominio similar al planteado en esta tesina, pero más específico.

Fecha de la presentación: Julio 2018

Índice general

Capítulo 1

Introducción

Motivación

Los sistemas robóticos son un tipo particular de sistemas: Están formados por diferentes componentes, diferentes sensores, deben estar preparados para hacer frente al entorno físico y son, en esencia, sistemas distribuidos de tiempo real. Se necesita que estos sistemas cuentan con una mejor calidad que un sistema de propósito general. La arquitectura compleja y variable, sumando a los problemas de un entorno incierto y cambiante, llevan a que un robot sólo pueda ser configurado y programado por expertos en el área de la robótica. Los enfoques tradicionales utilizados en el proceso de desarrollo de este tipo de sistemas están basados principalmente en codificar las aplicaciones sin ningún tipo de técnica de modelado. Entonces, al utilizar lenguajes de programación específicos se pierde la posibilidad de generalizar conceptos que pueden ser extraídos, reutilizados y aplicados en otros sistemas, lo que permitiría evitar la codificación (o recodificación) desde cero. Las metodologías y las herramientas utilizadas actualmente están tendiendo a solucionar este problema, pero aún no alcanzan a cubrir las necesidades de estos sistemas tan complejos ni tampoco ayudan a los usuarios menos experimentados a resolver problemas más simples. En esta tesina se presenta una posible solución a un dominio particular de la robótica. Tomando los conceptos y las alternativas que ofrece el mercado actual, se define un modelo listo para utilizar y que puede tomarse de base para extender y resolver problemas más específicos.

Objetivo

El objetivo de esta tesina es aplicar los conceptos del Desarrollo de Software Dirigido por Modelos (del inglés, Model Driven Development) a la creación de sistemas robóticos. Puntualmente, se define un lenguaje específico de dominio o DSL (del inglés, Domain Specific Language) para el siguiente problema: Un robot recorre un espacio conocido con anterioridad. Ese recorrido consiste en llegar a un conjunto ordenado de ubicaciones de ese espacio. Cuando el robot llega a una ubicación, puede ejecutar (o no) una serie de acciones. Una vez finalizada la ejecución de la/s accione/s en una ubicación, el robot continúa con el recorrido, trasladándose hacia la siguiente ubicación del conjunto. Luego, se presentan las transformaciones para generar el código fuente a partir del DSL. Esta generación, que transforma un modelo abstracto a código fuente específico de la plataforma, es automática (es decir, sin intervención humana). Como plataforma específica se utiliza el framework ROS (Robot Operating System). Por último, se construye una herramienta gráfica que permite especificar una instancia del DSL y generar el código a partir de ésta.

Estructura de la tesina

Este documento se encuentra organizado en ocho capítulos.

En el presente capítulo introductorio se menciona la motivación del trabajo de tesis, el objetivo y la estructura del documento.

En el capítulo dos se explican los conceptos básicos del Desarrollo de Software Dirigido por Modelos: modelos, transformaciones y definición formal de lenguajes.

En el capítulo tres se discuten las alternativas que ofrece el mercado para la resolución del objetivo planteado. Se habla de las propuestas, artículos de investigación y trabajos relacionados que fueron estudiados (y que conceptos fueron tomados).

En el capítulo cuatro se describen las tecnologías a utilizar en el desarrollo de esta tesina junto con una justificación de su elección. Específicamente, se habla del framework ROS (una visión general de los conceptos que maneja el proyecto) y de dos herramientas de la plataforma Eclipse: EMF para la definición del lenguaje y Acceleo para la generación del código.

En el capítulo cinco se presenta el DSL, junto con su respectiva documentación y explicación.

En los siguientes dos capítulos se presentan dos ejemplos prácticos de generación de código a partir de nuestro DSL. En el capítulo cinco se trata el caso de un TurtleBot 2 simulado y en el capítulo seis se trata el caso de un robot real (Create 1). Las dos soluciones trabajan bajo el framework ROS.

En el capítulo conclusivo, se detallan las conclusiones obtenidas, los alcances y las limitaciones de la tesina y los posibles trabajos a futuro.

Por último, se adjuntan los siguientes anexos:

- Una justificación de la representación de ciertas entidades del modelo y la explicación de ciertos conceptos de ROS, incluido el Stack de Navegación.
- Un manual para la herramienta gráfica desarrollada, que sirve también como ejemplo para la sintaxis concreta del DSL.

Capítulo 2

Conceptos básicos

En este capítulo se explican los conceptos del Desarrollo de Software Dirigido por Modelos (MDD, Model Driven Software Development), necesarios para la comprensión del desarrollo de esta tesina. Estos fueron extraídos del libro *Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica* [1].

2.1 Desarrollo de Software Dirigido por Modelos

MDD es un paradigma de desarrollo software que permite mejorar el proceso de construcción de software, basándose en un proceso guiado por modelos y soportado por distintas herramientas. Al ser "dirigido por modelos", MDD asigna un rol central y activo a los modelos en este paradigma, de modo que sean al menos tan importantes como el código fuente. Los modelos se generan partiendo de los más abstractos hasta llegar, mediante transformaciones y/o refinamientos, a los más concretos y finalmente, al código. Algunos de sus puntos claves son los descritos a continuación:

- **Mayor nivel de abstracción** en la especificación tanto del problema a resolver como de la solución en comparación con los métodos tradicionales de desarrollo de software. Se definen lenguajes de modelado específicos de dominio cuyos conceptos reflejan los conceptos del dominio del problema, al mismo tiempo que se ocultan o minimizan los aspectos relacionados con las tecnologías de implementación.
- **Aumento de confianza en la automatización** asistida por computadora para soportar el análisis, el diseño y la ejecución. La automatización es el mejor método para elevar la productividad y la calidad, mediante el uso de las computadoras para automatizar tareas repetitivas que usualmente los seres humanos no realizan con errores.
- **Uso de estándares industriales** como medio para facilitar las comunicaciones, la interacción entre diferentes aplicaciones y productos, y la especialización tecnológica.

La figura 2.1 muestra la parte del proceso de desarrollo de software en donde la intervención humana es reemplazada por herramientas automáticas. Los modelos pasan de ser entidades contemplativas (es decir, artefactos que son interpretadas por los diseñadores y programadores) para convertirse en entidades productivas a partir de las cuales se deriva la implementación en forma automática.

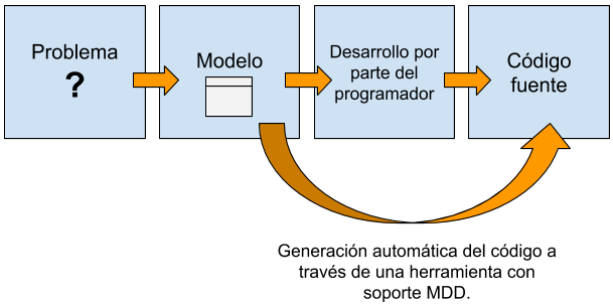

Figura 2.1: Diagrama de proceso de desarrollo de software. A partir de un problema, se define un modelo que representa la solución, que un desarrollador interpreta e implementa para obtener el código fuente. MDD reemplaza la interacción humana a través de herramientas automatizadas.

El ciclo de vida de desarrollo de software usando MDD es similar al ciclo de vida tradicional, sin embargo, una de las mayores diferencias se encuentra en el tipo de los artefactos generados durante el proceso de desarrollo, los cuales son modelos formales, es decir, modelos que pueden ser comprendidos por una computadora. En el desarrollo de software dirigido por modelos se pueden identificar distintos tipos de modelos:

- **CIMs** (Computation Independent Model): son modelos de alto nivel de abstracción independientes de cualquier metodología computacional.
- **PIMs** (Platform Independent Model): son modelos independientes de cualquier tecnología de implementación.
- **PSMs** (Platform Specific Model): modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica.
- **IMs** (Implementation Model): modelos que representan el código fuente en sí mismo.

En el desarrollo de software tradicional, las transformaciones de modelo a modelo, o de modelo a código son realizadas la mayoría de las veces con la necesidad de la intervención humana. Muchas herramientas tienen la posibilidad de generar código a partir de modelos, aunque usualmente generan solo un esqueleto de código, que luego debe ser completado manualmente. Diferenciándose, las transformaciones MDD son siempre ejecutadas por herramientas. Algo novedoso que propone MDD es que las transformaciones entre modelos de mayor nivel de abstracción (por ejemplo de un PIM a PSMs) sean automáticas. De este modo, se puede afirmar que el motor de MDD está constituido por las transformaciones entre modelos.

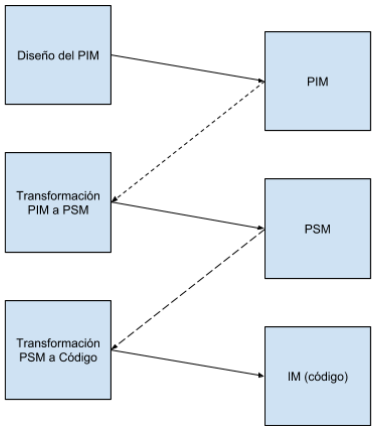

Figura 2.2: Modelos y Transformaciones en MDD.

2.2 Propuestas Concretas para MDD

Dos de las propuestas concretas del el ámbito de Model-Driven Development son:

2.2.1 Arquitectura Dirigida por Modelos

MDA (acrónimo inglés de Model Driven Architecture) es un concepto promovido por el OMG [10] (acrónimo inglés de Object Management Group) que tiene como objetivo afrontar los desafíos de integración de las aplicaciones y los continuos cambios tecnológicos. MDA es una arquitectura que proporciona un conjunto de guías que permiten la estructuración de especificaciones expresadas como modelos, mediante el proceso MDD. La funcionalidad del sistema es definida en primer lugar como un Platform-Independent Model a través de un lenguaje específico para el dominio del que se trate. Aquí aparece un tipo de modelo existente en MDA, y no mencionado por MDD:, que es el modelo de definición de la plataforma (Platform Definition Model o PDM). Entonces, dado un PDM correspondiente a CORBA, .NET, Web, etc., el modelo PIM puede traducirse a uno o más Platform-Specific Models para la implementación correspondiente, emplenado diferentes lenguajes específicos del dominio, o lenguajes de propósito general como Java, C#, Python, etc. MDA está relacionada con múltiples estándares, tales como el Unified Modeling Language (UML), el Meta-Object Facility (MOF), XML Metadata Interchange (XMI), Enterprise Distributed Object Computing (EDOC), el Software Process Engineering Metamodel (SPEM) y el Common Warehouse Metamodel (CWM). Otro de los beneficios de MDA es que asegura que el PIM sobreviva a las modificaciones que se produzcan en las tecnologías de fabricación y en las arquitecturas de software.

2.2.2 Modelado Específico del Dominio (DSM y DSLs)

La iniciativa DSM (Domain-Specific Modeling) se basa en la creación de modelos para un dominio empleando un lenguaje focalizado y especializado para dicho dominio. Estos lenguajes se denominan DSLs (Domain Specific Language) y permiten la especificación de la solución empleando conceptos del dominio del problema. Los productos finales son luego generados desde estas especificaciones de alto nivel. Esta automatización es posible si el lenguaje y los generadores se ajustan a los requerimientos de un único dominio (un dominio a un área de interés para un esfuerzo de desarrollo en particular). Cada solución DSM se enfoca en dominios acotados debido a que el foco reductor habilita mejores posibilidades para su automatización y estos dominios pequeños son más fáciles de definir. El desafío de los desarrolladores y empresas se focaliza en la definición de los lenguajes de modelado, la creación de los generadores de código y la implementación de frameworks específicos del dominio. En general, DSM usa los conceptos dominio, modelo, metamodelo, meta-metamodelo como MDD y propone la automatización en el ciclo de vida del software. Los DSLs no emplean ningún estándar del OMG para su infraestructura, y son utilizados para generar modelos, y usualmente suelen ser gráficos.

2.3 Modelos

Los modelos son el foco central del desarrollo de software dirigido por modelos. Son representaciones conceptuales o físicas a escala de proceso o sistema, que tiene como finalidad el análisis de su naturaleza, el desarrollo o comprobación de hipótesis o supuestos, y la brindar la posibilidad de una mejor comprensión del fenómeno real al cual él modelo representa permitiendo así él perfeccionamiento de los diseños previo al inicio de la construcción de obras u objetos reales. Se utilizan con frecuencia para el estudio de represas, aeronaves, puentes, puertos, etc. Muchas veces, se requiere de la construcción de más de un modelo. Por ejemplo, en la construcción

de una vivienda se podría crear un modelo del plano de la vivienda en general, y otro aparte, detallando el plano de las instalaciones eléctricas.

2.3.1 Tipos de modelos

Los sistemas informáticos requieren la presencia de varios modelos interdependientes, cada uno con diferentes niveles de abstracción, como por ejemplo, modelos de análisis, modelos de diseño y modelos de implementación. Cada uno de éstos, representa alguna parte del sistema (ej. interfaz con el usuario), o algún requisito (ej. seguridad), o alguna tarea (ej. modelos de pruebas). En muchos casos es posible generar un modelo a partir de otro. Se pueden distinguir distintos tipos de modelos:

- **Modelos a lo largo del proceso de desarrollo**: dentro de estos, se encuentran los modelos de análisis, los modelos de diseño, y los de implementación.
- **Modelos abstractos y modelos detallados**: la diferencia entre los modelos abstractos y los detallados, muchas veces pasa por la subjetividad con la que se lo mire al modelo.
- **Modelos de negocio y modelos de software**: un modelo de negocio describe a una parte o la totalidad de un negocio o una empresa. El lenguaje que se empleado para la construcción de un modelo de negocio incluye un vocabulario que brinda al modelador la posibilidad de especificar los procesos de negocio, los clientes, los departamentos, las dependencias entre procesos, etc. Como los modelos de negocio no describen necesariamente detalles acerca del sistema de software que se usa en la empresa, suele ser clasificado como CIM. Cuando una parte del negocio es soportada por un sistema, se construye un modelo de software diferente que describe dicha parte. Generalmente los requisitos del modelo de software derivan del modelo de negocio al cual el software brinda soporte.
- **Modelos estáticos (o estructurales) y modelos dinámicos (o de comportamiento)**: los modelos estáticos describen la base estructural sobre la que se apoya el sistema, definida por el conjunto de objetos que conforman al sistema, con sus propiedades y sus conexiones. Por otra parte, los modelos dinámicos describen la dinámica de comportamiento que los objetos del sistema despliegan. Ambos tipos de modelos están fuertemente interrelacionados y, en conjunto, constituyen el modelo global del sistema. También puede haber modelos híbridos, que posean sub-modelos estructurales y sub-modelos dinámicos.
- **Modelos independientes de la plataforma y modelos específicos de la plataforma**: los primeros describen al sistema de manera independiente con respecto a los conceptos técnicos que involucra su implementación sobre alguna plataforma de software, mientras que los segundos se basan en la descripción de dichos aspectos técnicos. Dentro de estos tipos de modelos, se identifican los tipos de modelos CIM, PIM, PSM y IM, que fueron descritos anteriormente. Cabe destacar que la transformación completamente automática entre CIM y PIM no es posible ya que la decisión acerca de cuáles partes del CIM serán soportadas por el sistema de software debe ser realizada por un humano.

2.3.2 Cualidades de un modelo

El modelo es la base fundamental de información sobre la que interactúan los expertos en el dominio del problema y los desarrolladores de software, siendo en consecuencia, de gran importancia que exprese la esencia del problema en forma clara y precisa. Es importante que los modelos definidos cumplan las siguientes cualidades:

- **Comprensibilidad**: el modelo debe ser expresado en un lenguaje que resulte accesible para todos sus usuarios.

- **Precisión**: el modelo debe ser una representación lo más certera posible del objeto o sistema modelado, para lo cual, el lenguaje de modelado debe tener una semántica precisa que permita la interpretación unívoca de los modelos.
- **Consistencia**: el modelo no puede presentar información contradictoria.
- **Completitud**: el modelo debe capturar todos los requerimientos necesarios.
- **Flexibilidad**: los modelos deben tener la posibilidad de ser fácilmente adaptados para reflejar los ocasionales cambios en el dominio del problema.
- **Re-usabilidad**: además de describir el problema, el modelo también tiene que proveer las bases para el re-uso de conceptos y construcciones que se presentan en forma recurrente en una amplia gama de problemas.
- **Corrección**: el modelo debe ser analizado para asegurar que cumple con las expectativas del usuario y puede ser utilizado como referencia para analizar la corrección de la implementación del sistema.

2.3.3 ¿Cómo se define un modelo?

El modelo del sistema se construye usando un lenguaje de modelado. Para los modelos informales, se utiliza el lenguaje natural, las figuras, las tablas y/o otras notaciones. En cambio, en los modelos formales la notación usada es un formalismo, y por ende posee una sintaxis y semántica bien definidos. También cabe destacar que existen estilos de modelado intermedios llamados semi-formales, ya que frecuentemente presentan una notación cuya sintaxis y semántica está sólo parcialmente formalizada.

2.3.4 El rol del UML en MDD

UML [11] (Unified Modeling Language) es un lenguaje de propósito general, ubicado entre los más elegidos a la hora de definir los modelos en MDD. Posee un estándar abierto, y es el estándar de facto para el modelado de software. Es un lenguaje que ha demostrado ser durable ya que su primera versión apareció en 1995, y además presenta la ventaja de contar con la disponibilidad de una gran cantidad de libros de buena calidad.

Un perfil en UML es un conjunto de extensiones que especializan a UML para su uso en un dominio particular, como por ejemplo, el perfil de testing o el perfil de servicios de software. Cada uno, define un conjunto de estereotipos que extienden a los conceptos de UML (por ejemplo, el perfil de UML de bases de datos define el atributo primaryKey), permitiendo la transmisión de información semántica adicional que resulta entendible para él modelador y las herramientas automáticas que procesan a los modelos.

2.4 Transformaciones

Se puede definir a una transformación como aquella que genera un nuevo modelo a partir de uno fuente de acuerdo a una definición de transformación, la cual consiste en una colección de reglas, que son especificaciones no ambiguas de las formas en que un modelo (o parte de él) puede ser usado para crear otro modelo (o parte de él).

Una transformación entre modelos puede verse como un programa de computadora que toma un modelo como entrada y produce un modelo como salida. Por lo tanto las transformaciones podrían describirse (es decir, implementarse) utilizando cualquier lenguaje de programación. Una herramienta que soporte MDD, toma un PIM como entrada y lo transforma en un PSM. La misma herramienta u otra, tomará el PSM y lo transformará a código. En la figura 2.3 se

muestra la herramienta de transformación como una caja negra, que toma un modelo de entrada y produce otro modelo como salida.

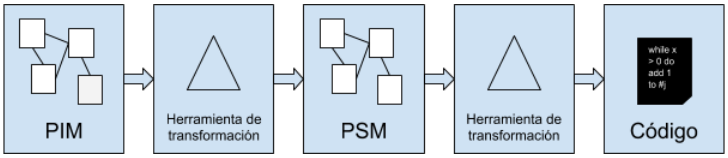

Figura 2.3: Proceso de transformación de modelos en otros modelos mediante herramientas de transformación.

La puesta en práctica de MDD requiere de la existencia de herramientas que le den soporte a la creación de modelos y transformaciones, dentro de las cuales deben existir:

- Editores gráficos para la creación de modelos.
- Repositorios para la persistencia de los modelos, el manejo de sus modificaciones, y su versionado.
- Herramientas para validación de los modelos (completitud, correctitud, etc).
- Editores de transformaciones de modelos que den soporte a los distintos lenguajes de transformación.
- Compiladores y debuggers de transformaciones.
- Herramientas para verificar y/o testear las transformaciones.

2.5 Definición formal de lenguajes de modelado

Generalmente, la sintaxis de los lenguajes se definía casi exclusivamente usando Backus Naur Form (BNF), que es una meta sintaxis usada para expresar gramáticas libres de contexto, es decir, una manera formal de describir cuáles son las palabras básicas del lenguaje y cuáles secuencias de palabras forman una expresión correcta dentro del mismo. Una especificación en BNF es un sistema de reglas de la derivación, escrito como:

```
<símbolo> ::= <expresión con símbolos>
```

Dónde *símbolo* es un no-terminal, y la *expresión* consiste en secuencias de símbolos separadas por la barra vertical, indicando una opción, cada una de las cuales es una posible sustitución para el símbolo a la izquierda. Los símbolos que nunca aparecen en un lado izquierdo son terminales. Este método resulta útil para lenguajes textuales, pero como usualmente los lenguajes de modelado no están basados en texto, sino que se basan en gráficos, hay que recurrir a un mecanismo distinto para su definición.

Algunas diferencias existentes entre los lenguajes basados en texto y los lenguajes basados en gráficos son:

- **Contenedor vs. referencia**: en los lenguajes textuales una expresión puede formar parte de otra expresión, la cual a su vez puede estar englobada en otra mayor. Esta relación de contenedor da origen a un árbol de expresiones. A diferencia de los anteriores, en los lenguajes gráficos, se genera un grafo de expresiones debido a que una sub-expresión puede ser referenciada desde dos o más expresiones distintas.
- **Sintaxis concreta vs. sintaxis abstracta**: en los lenguajes textuales ambas sintaxis coinciden casi exactamente, mientras que en los gráficos se presentan marcadas diferencias.

- **Ausencia de una jerarquía clara en la estructura del lenguaje**: a diferencia de los lenguajes textuales, los cuales en general son aprendidos leyéndolos de arriba hacia abajo y de izquierda a derecha, en los lenguajes gráficos el aprendizaje se realiza de manera diferente, según la sintaxis concreta. Este hecho influye en la jerarquía de la sintaxis abstracta, provocando que en muchas ocasiones no exista un orden entre las categorías sintácticas.

Para facilitar la definición de los lenguajes gráficos, se desarrolló la técnica llamada "metamodelado". El "metamodelo" es el modelo que describe qué elementos pueden ser empleados en la creación de un modelo y cómo los mismos pueden ser conectados. Por ejemplo, el lenguaje UML establece que dentro de un modelo se pueden utilizar los conceptos Clase, Atributo, Asociación, Paquete, etc. Entonces, se puede describir un lenguaje por medio de un metamodelo. A su vez, como un metamodelo es un modelo, debe estar escrito en un lenguaje bien definido, que se lo conoce como "metalenguaje". Desde este punto de vista, BNF es un metalenguaje. En la figura 2.4 se muestra gráficamente esta relación.

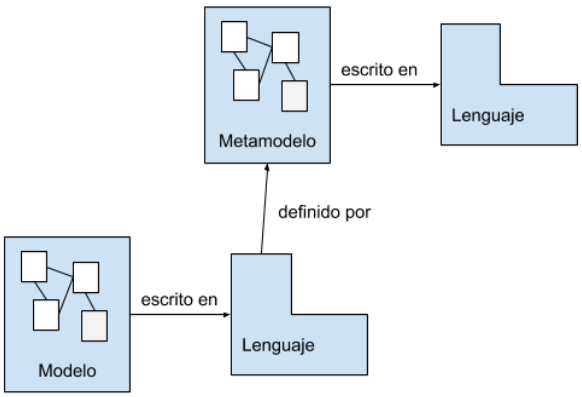

Figura 2.4: Relación entre modelos, metamodelos, lenguajes y metalenguajes.

El metamodelo describe la sintaxis abstracta del lenguaje, y es la base para el procesamiento automatizado de los modelos. Por otra parte, la sintaxis concreta es definida mediante otros mecanismos y no es relevante para las herramientas de transformación de modelos. La sintaxis concreta es la interfaz para el modelador e influye fuertemente en el grado de legibilidad de los modelos.

Como consecuencia del desacoplamiento entre el metamodelo y la sintaxis concreta de un lenguaje puede ocurrir que la misma sintaxis abstracta sea visualizada mediante distintas sintaxis concretas. Como ejemplo, se puede mencionar la posibilidad de que un mismo lenguaje tenga una sintaxis concreta gráfica y otra textual.

Pero **¿por qué el metamodelo es tan importante para MDD?** Básicamente, porque se necesita contar con un mecanismo no ambiguo que permita definir lenguajes de modelado, y que posibilite que una herramienta de transformación pueda leer, escribir y entender modelos. Además, las reglas de transformación que constituyen una definición de una transformación describen cómo un modelo en un lenguaje fuente puede ser transformado a otro en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para la definición de la transformación. Por último, se puede mencionar que la sintaxis de los lenguajes en los que se expresan las reglas de transformación también debe estar formalmente definida para posibilitar su automatización, por ende, también se utiliza el metamodelado.

2.6 La arquitectura de cuatro capas de modelado del OMG

El metamodelado es un mecanismo que permite definir formalmente lenguajes de modelado, como por ejemplo UML. A continuación se presenta la arquitectura de cuatro capas de modelado propuesta del OMG, en donde cada capa se define como instancia de la anterior. Esta arquitectura de modelos denominada MOF [12] (Meta Object Facility) representa el nivel más general (M3) y tiene el objetivo de permitir la incorporación de nuevos lenguajes de modelado (metamodelos) para propósitos específicos.

1. **Capa M0 (Instancias)**: En el nivel M0 se encuentran todos los objetos de la aplicación, que son entidades físicas.
2. **Capa M1 (Modelos)**: Por encima de la capa M0 se sitúa la capa M1, que es el modelo de un sistema de software. Los conceptos del nivel M1 representan categorías de las instancias de M0, es decir, que cada elemento de M0 es una instancia de un elemento de M1. Sus elementos son modelos de los datos.
3. **Capa M2 (Metamodelos)**: Tal como ocurre con las capas M0 y M1, los elementos del nivel M1 son a su vez instancias del nivel M2, el cual recibe el nombre de metamodelo.
4. **Capa M3 (Metametamodelo)**: De igual forma, podemos ver los elementos de M2 como instancias de la capa M3 o capa del meta-metamodelo. Un meta-metamodelo es un modelo que define el lenguaje para representar un metamodelo. M3 es el nivel más abstracto, y es el que permite definir metamodelos concretos (dentro del OMG, MOF es el lenguaje estándar de la capa M3, por lo que todos los metamodelos de M2 son instancias de MOF y a su vez, MOF se define a sí mismo).

La figura 2.5 ejemplifica cada capa de la arquitectura para los estándares de OMG e ilustra la relación entre cada capa.

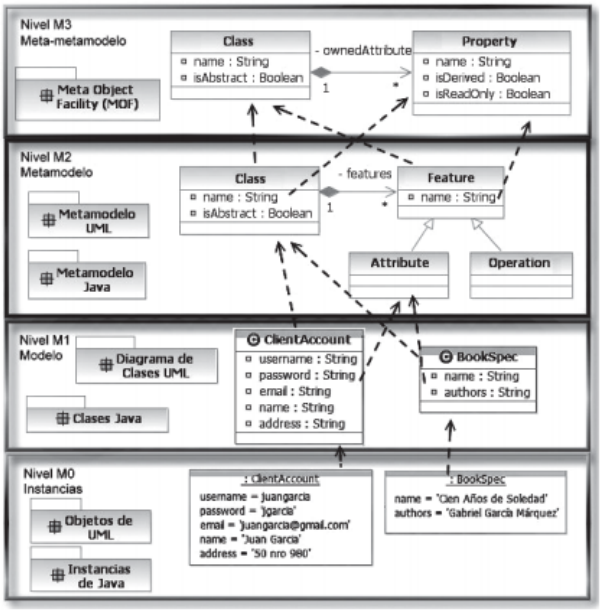

Figura 2.5: Vista general de las relaciones entre los cuatro niveles de la arquitectura de modelado del OMG.

2.7 El lenguaje de modelado más abstracto: Meta-Object Facility

El lenguaje MOF, nombrado en las secciones anteriores, es un estándar del OMG para la ingeniería conducida por modelos que provee un meta-metalenguaje que permite definir metamodelos en la capa M2. El ejemplo más popular es el metamodelo UML, que es una arquitectura de metamodelado cerrada (porque el metamodelo de MOF se define en términos de sí mismo) y estricta (porque cada elemento de un modelo en cualquiera de las capas tiene una correspondencia estricta con un elemento del modelo de la capa superior). MOF se basa en el paradigma de Orientación a Objetos, por lo que emplea los mismos conceptos y la misma sintaxis concreta que los diagramas de clases de UML.

Si se compara BNF con MOF, se puede decir que si bien la meta sintaxis BNF y el meta lenguaje MOF son formalismos creados con el objetivo de definir lenguajes textuales y lenguajes gráficos respectivamente, se ha demostrado que ambos tienen igual poder expresivo, siendo posible la transformación bi-direccional de sentencias cuya sintaxis fue expresada en BNF y sus correspondientes modelos cuya sintaxis fue expresada en MOF.

Capítulo 3

Estado del arte

En este capítulo se describen propuestas, trabajos y artículos actuales relacionados con la programación de robots y el paradigma Model Driven Development, que han servido como una guía para el desarrollo de esta tesina. Al finalizar el capítulo se comentan las conclusiones de los trabajos investigados.

3.1 Trabajos relacionados

3.1.1 Un enfoque dirigido por modelos para la creación de sistemas robóticos

En este proyecto [3] se investigan el uso actual de técnicas modernas de ingeniería de software (como MDE, SOA, CBD y DSM) y cómo las mismas permiten el desarrollo de sistemas robóticos y además un alto nivel de automatización. Se propone la definición de un framework de metodologías, teniendo en cuenta que las plataformas robóticas deben tener una alta capacidad dinámica adaptativa, de modo que se deben definir a las interfaces y los principales comportamientos de los sistemas robóticos con un alto grado de abstracción, permitiendo que puedan ser reutilizados y mantenidos.

Se propone como ejemplo a un robot de tres ruedas que combate incendios. Este robot está formado por distintos componentes y debe caminar por una plataforma que posee obstáculos random y debe encontrar incendios mediante la realización de fotos; una vez que encuentra un incendio, el robot debe dirigirse hacia el mismo y extinguir. Otra forma que tiene el robot de encontrar incendios, es mediante la interacción de componentes ajenos al robot, como detectores de incendio, de modo que si se produce el incendio, el robot debe preguntar a donde debe moverse, para lo cual es necesario que mande su posición al servicio “mapa” (mediante su GPS), el cual le enviará un camino a seguir para llegar al incendio que deberá apagar. De este modo, se identifican los componentes dentro del sistema desarrollado y a partir de estos se arma un PIM, que se puede visualizar en la figura 3.1.

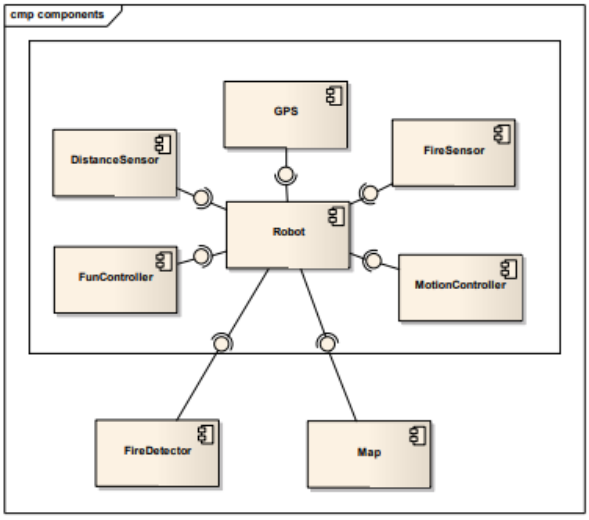

Figura 3.1: Componentes que forman al sistema del robot apaga incendios.

El artículo continúa con la definición de la interfaces de cada componente hasta llegar al comportamiento específico de cada una. La idea es que, al cambiar la plataforma específica, la generación de código siga funcionando sin alterar ni un componente, interfaz o comportamiento del modelo.

3.1.2 Component-Based Robotic Engineering (Partes I y II)

En esta serie formada por dos artículos [5] [6] se propone la aplicación de Ingeniería de Software basada en Componentes (CBSE, Component-based software engineering) en la robótica. Los autores definen a un componente como una pieza de software que implemente alguna funcionalidad específica del robot. El objetivo es diseñar principios y metodologías para permitir el desarrollo reusable y mantenible de estos componentes de software para ser empleados en robots.

En el primero de los artículos se explican los conceptos fundamentales de la propuesta, pero pone énfasis en: el diseño de un componente individual, en separar la especificación del componente de su implementación y en técnicas para lograr la mayor flexibilidad e interoperabilidad entre ellos.

Como ejemplo, propone una solución al problema de la planificación de ruta o path planning. En ella se definen cuatro componentes llamados *Cartesian space*, *Configuration space*, *Collision checker* y *Path Planner* (figura 3.2).

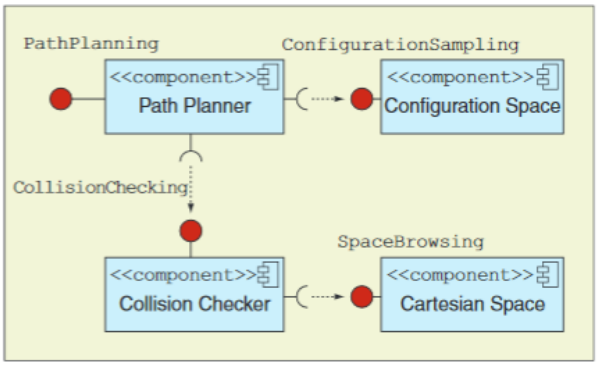

Figura 3.2: Ejemplo de solución al problema del Path Planning basada en componentes. En esta figura, los círculos rojos (lollipop) representan una interfaz que él componente provee, los semicírculos (socket) representan interfaces que él componente requiere; el nombre de la interfaz (en ambos casos) aparece cerca del símbolo; y la dependencia se aprecia con una flecha que sale desde un socket hasta un lollipop.

- *Cartesian space* encapsula la estructura que representa al entorno que rodea al robot. Provee información acerca de objetos y posiciones y además provee servicios como calcular la posición relativa entre dos objetos.
- *Configuration space* encapsula las estructuras de datos que definen una configuración espacial del robot y provee servicios para, dada dos configuraciones espaciales, poder comparar, medirlas, interponerlas y etc.
- *Collision checker* verifica que, dado una configuración espacial, esta esté libre de obstáculos. Para ello, requiere acceso a la información provista por Cartesian space (como se aprecia en la 3.2).
- Por último, *Path planner* calcula un camino de robot desde el principio y el final de la configuración del robot. Estos algoritmos utilizan la información actual del entorno y la propia del robot, por lo que el componente necesita de los servicios de Configuration space y Collision checker.

A partir del ejemplo, el artículo clasifica las interfaces de los componentes en ciertos grupos como por ejemplo, interfaces con servicios de datos e interfaces con servicios de funcionalidad o interfaces requeridas o interfaces provistas. Además, indica cómo deberían tipificarse los datos de cada componente (y de las interfaces). Y, a la hora de la implementación, propone encarar la programación de un componente como si de un framework se tratase.

En el segundo artículo, se discuten las técnicas para ensamblar los componentes individuales en un sistema completo y los métodos para derivar las aplicaciones finales desde una base común de componentes reusables.

3.1.3 Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines

Este paper [7] tiene un objetivo similar a esta tesina, es decir, aplicar Model Driven Development (MDD) para programar un determinado conjunto de robots. Se plantea un problema para un dominio particular, se define un DSL y se genera el código para una plataforma específica de manera automática. De esta forma, un usuario puede programar un robot abstrayéndose de toda la implementación. La figura 3.3 representa esta idea.

Como motivación, se analiza un caso común de uso de robots: los robots que levantan objetos y los depositan en otro lugar. A partir de esa definición del dominio, se observa que resulta

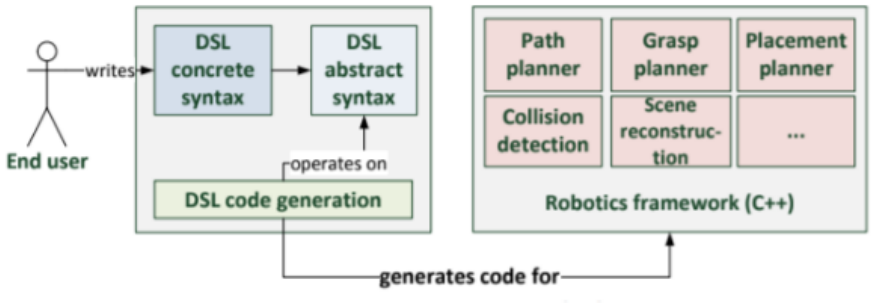

Figura 3.3: Diagrama del plan del paper "Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines". El usuario, a través de una herramienta gráfica, construye una instancia del DSL, que luego de varias transformaciones automáticas termina en código C++ listo para ejecutar.

complejo programar un robot personal para este tipo de tareas y que además surgen otras subtareas a implementar para el robot (y que deben ser abstraídas para el usuario).

Se utiliza EMF [14] para la definición de la sintaxis abstracta, a partir de la cual los usuarios finales podrán definir la sintaxis concreta usando operaciones de alto nivel, como por ejemplo agarrar, colocar o soltar. Luego, se genera el código para ROS [16] (Robot Operating System) mediante el uso de Acceleo [15].

3.1.4 Automatic generation of detailed flight plans from high-level mission descriptions

Este paper [4] se basa en la popularidad de los drones (ya que simplifican una gran cantidad de tareas cotidianas) y la especificidad y la posibilidad de errores en el desarrollo de programas para los mismos.

Como se viene discutiendo hasta ahora (y para cualquier sistema robótico), esto limita el desarrollo de software únicamente a personas expertas del tema. Además, otro de los problemas que describen es la necesidad de dos operadores por cada dron utilizado: donde el primero controle los movimientos del mismo y el segundo, la instrumentación.

A partir de esto, se indica la necesidad de la aplicación de técnicas de ingeniería de software que permita el soporte de la definición, el desarrollo y la realización de misiones que involucran conjuntos de drones autónomos.

El artículo describe la plataforma FLYAQ [19], que permite a los usuarios no expertos definir misiones con un alto nivel de abstracción, ocultando así la complejidad de bajo nivel y la información relacionada con la dinámica del vuelo de los drones. Además, FLYAQ provee un DSL extensible llamado Monitoring Mission Language (MML), que permite definir gráficamente las misiones del dron. Para calcular los puntos de referencia y trayectorias, se utiliza un lenguaje intermedio llamado QBL (Quadrotor Behaviour Languaje). MML está compuesto por tres capas:

- La que sirve para especificar la misión mediante los constructores del lenguaje.
- La que sirve para especificar el contexto en el que el conjunto de drones van a operar.
- Un mapa, representando la zona geográfica donde la misión va a ser ejecutada.

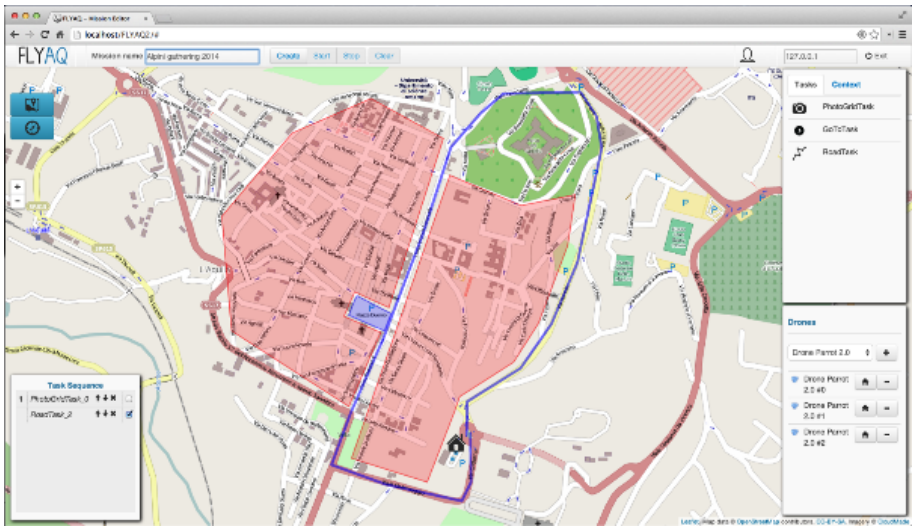

Figura 3.4: Captura de la herramienta FLYAQ.

Conceptualmente, se puede decir que una misión consiste de:

- Una **entrada** (con las operaciones necesarias para comenzar la misión, como por ejemplo, el despegue y la búsqueda de la altitud necesaria del punto de partida).
- La **ejecución de tareas** (con las operaciones requeridas para cumplir cada tarea de la misión, como por ejemplo, buscar un objeto en un área). Se proveen sólo un conjunto de tareas limitado.
- Un **salida** (que consiste en las operaciones requeridas para concluir con la misión, como por ejemplo, el aterrizaje).

Entonces, para un usuario, una misión FLYAQ resulta de un área geográfica, una entrada, una salida, y un conjunto tareas a realizar mientras los drones atraviesan distintos puntos del área. Se asume que al principio de la misión cada dron estará estacionado en su ubicación de origen, y que al final de la misión cada drone aterriza en una ubicación específica.

La generación automática del código se encarga de la lógica que cada dron debe tener para cumplir la misión, calculado las trayectorias, previniendo colisiones entre drones y obstáculos, etc. De esto se desprende un aspecto interesante de este artículo: la extensibilidad del DSL utilizado para definir las misiones, la cual permite a los programadores personalizar el lenguaje de acuerdo a sus necesidades, agregando tareas adicionales específicas adaptadas al dominio en cuestión.

3.2 Conclusiones del estado del arte

Los artículos *Un enfoque dirigido por modelos para la creación de sistemas robóticos* [3] y *Component-Based Robotic Engineering (Part I & II)* [5] [6] han permitido establecer una firme base acerca del concepto MDD y las ventajas que conlleva su aplicación a sistemas robóticos, ilustrando la manera de separar las funcionalidades en componentes y definir las mismas con un alto nivel de abstracción.

El artículo *Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines* [7] fue analizado como un ejemplo práctico. Por esta razón, el trabajo de esta tesina fue desarrollado con las herramientas utilizadas en este mismo artículo: EMF (para la definición del lenguaje), Acceleo (para la transformación de código) y ROS (como plataforma específica).

Por último, el paper *Automatic generation of detailed flight plans from high-level mission descriptions* [4] también es un ejemplo práctico que tiene el mismo objetivo de esta tesina. Lo interesante de este es la interfaz gráfica que le provee a los usuarios, la extensibilidad del lenguaje, la aceptación de la plataforma y sobre todo, que es una aplicación que es utilizada en la vida real.

Capítulo 4

Descripción de tecnologías

En este capítulo se describen las tecnologías utilizadas. Primero se hace un breve comentario de la plataforma Eclipse [13] para luego explicar dos de sus extensiones: EMF (para la definición del modelo) y Acceleo (para la generación del código). Para finalizar el capítulo se trata ROS (Robot Operating System), un framework para la programación de robots.

Cada tecnología mencionada tiene la justificación de la elección al finalizar su respectiva sección.

4.1 Eclipse

Eclipse es una plataforma de desarrollo, diseñada para ser extendida de forma indefinida a través de plug-ins. Fue concebida desde sus orígenes para convertirse en una plataforma de integración de herramientas de desarrollo. No tiene en mente un lenguaje específico, sino que es un IDE genérico, aunque cuenta con mucha popularidad entre la comunidad de desarrolladores del lenguaje Java.

Básicamente proporciona herramientas para la gestión de espacios de trabajo, escribir, desplegar, ejecutar y depurar aplicaciones. Dispone de un editor de texto con resaltado de sintaxis, la compilación es en tiempo real, brinda pruebas unitarias, asistentes para creación de proyectos, clases, tests, etc, y refactorización.

El IDE fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Este IDE es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

4.1.1 ¿Por qué Eclipse?

- La arquitectura plug-in, que permite extender la funcionalidad del IDE con cualquier extensión deseada.
- La existencia de plug-ins necesarios para la realización de esta tesina, como son EMF y el proyecto Acceleo.
- Facilidad de uso e instalación, además de un gran respaldo de la comunidad. Los autores de esta tesina están familiarizados con el IDE.

4.2 Eclipse Modeling Framework (EMF)

El proyecto EMF es un framework para modelado, que permite la generación automática de código para construir aplicaciones, a partir de datos estructurados. Comenzó como la imple-

mentación del metalenguaje MOF y su uso se ha extendido de gran manera. Se apoya en el hecho de que cualquier programa trabaja con un modelo de datos (definido mediante clases Java, UML, esquemas XML, etc). EMF compone la base de todas las herramientas de Model Driven Development que existen en la plataforma Eclipse.

La principal característica que posee el framework es que permite usar un modelo como punto de partida para la generación de código. De este modo, se puede refinar el modelo iterativamente y automatizar la regeneración del código, al mismo tiempo que permite al usuario modificar el código generado. De esta manera, el desarrollador puede poner su foco en el diseño del modelo y dejar en manos del framework los detalles de implementación.

El corazón del framework es el meta-modelo Ecore, que se utiliza para describir los modelos (archivos .ecore). Es una implementación similar a Essential Meta-Object Facility (EMOF), que a su vez es una especificación del estándar Meta-Object Facility (MOF) de la OMG. La especificación de un modelo EMF está dada por:

- Atributos del objeto.
- Relaciones (asociaciones) entre objetos.
- Métodos (operaciones) disponibles del objeto.
- Constantes sobre objetos y relaciones.

Básicamente, consiste de un subconjunto de los elementos del diagrama de clases de UML.

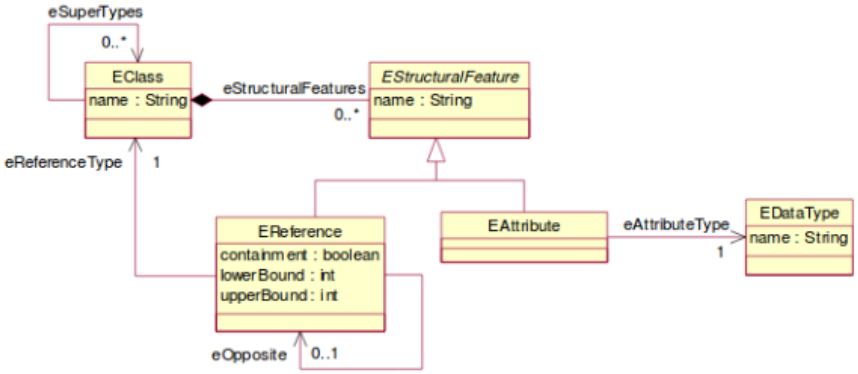

Figura 4.1: Componentes del meta-modelo Ecore.

Además el framework unifica las tecnologías JAVA, XML y UML, pudiendo definir un modelo mediante un diagrama Ecore, interfaces y anotaciones Java, diagramas UML (de clase) y esquemas XML, ya que las mencionadas tecnologías brindan la misma información pero las representan o visualizan de distinta manera. Y, a partir de un modelo, puede generar las otras representaciones así como el código de implementación.

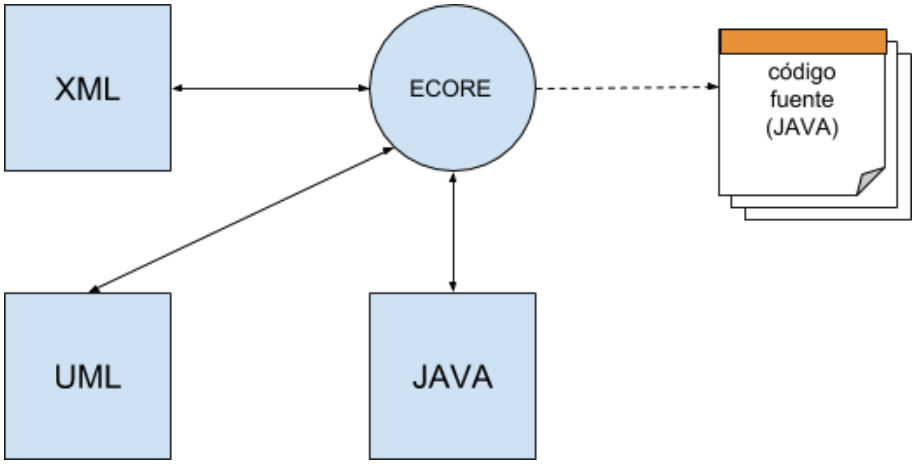

Figura 4.2: Un modelo Ecore puede ser definido mediante esquemas XML, diagramas UML e interfaces y anotaciones JAVA. Además, un modelo Ecore puede ser transformado a cualquiera de las tres mencionadas tecnologías. El código JAVA siempre es generado a partir del modelo Ecore.

La persistencia (serialización) de todos los archivos manejados por el framework (entre los que se encuentran los modelos Ecore) se hacen a través de archivos XMI (XML Metadata Interchange). Todo lo necesario para el manejo de este tipo de archivos se encuentra incluido en el framework.

4.2.1 ¿Por qué EMF?

EMF es una herramienta con enfoque práctico:

- No requiere la definición de un modelo muy abstracto.
- Combina el modelo con la programación para obtener lo mejor de cada una
- Aumenta la productividad gracias a la generación de código automática
- Es la base del paradigma Model Driven Development en la plataforma Eclipse.

4.3 Acceleo

Acceleo es un generador de código de la fundación Eclipse, que permite implementar la transformación de modelo a texto en enfoques dirigidos por modelos. Se puede utilizar para generar código (JAVA, PHP, C++, etc) a partir de cualquier metamodelo, ya sea Ecore, UML o un DSL particular, siempre y cuando este sea compatible con EMF. La herramienta se encuentra disponible como un plug-in para Eclipse, de código abierto y con licencia pública de GNU. Requiere el uso de Java 5 o superior, y tiene soporte para Linux, Mac OS X, y Windows XP en adelante.

Acceleo usa un enfoque basado en templates, en dónde el texto es generado a partir de elementos que posee algún modelo específico. La herramienta funciona partiendo de un modelo y un módulo, donde se encuentran las transformaciones a realizar para la generación del código. La figura 4.3 ilustra esta situación.

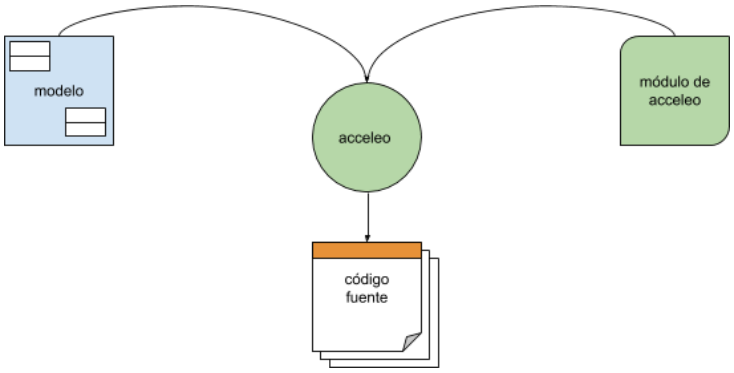

Figura 4.3: Funcionamiento de la herramienta. A partir de un modelo y una "plantilla" de transformación, el motor de Acceleo genera el código fuente.

Los archivos con extensión *.mtl* (Model Transformation Language) describen un módulo de Acceleo, dónde se define el código que se generará para una plataforma específica, es decir, la transformación del modelo a texto. Un generador de código Accelo está formado por uno o varios módulos. A su vez, estos están compuestos por templates.

Los templates son básicamente un conjunto de directivas Acceleo que describen la información necesaria para generar el código fuente a partir del modelo. Por ejemplo, la directiva para generar un archivo de texto se representa con la etiqueta [file]. Los templates tienen una visibilidad con un alcance similar a los lenguajes orientados a objetos, por ejemplo, pueden ser invocados sólo desde el módulo que los contiene o desde un módulo externo. Un módulo puede extender otro módulo y también puede importar otro módulo para acceder a sus templates.

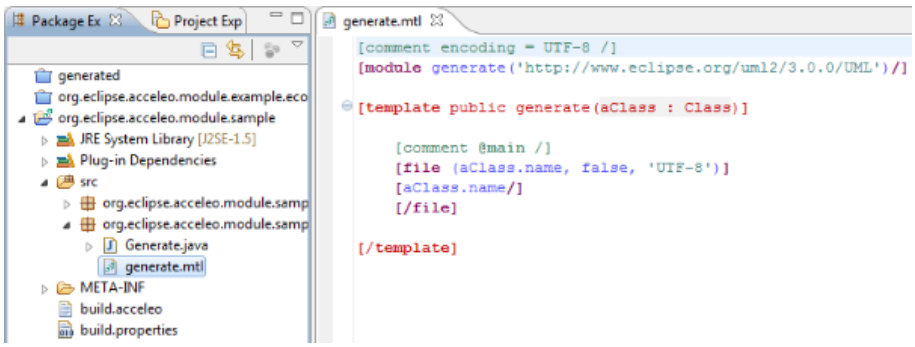

Figura 4.4: Vista de un módulo (generate.mtl) de Acceleo. Se aprecian las directivas de Acceleo, como la definición de un módulo, de un template, la generación de un archivo y el uso del metamodelo.

Como se puede ver en la figura 4.4, las directivas de Acceleo se identifican por etiquetas entre corchetes. El archivo generate.mtl es un módulo de nombre generate y que recibe como entrada un modelo UML (línea 2). Contiene un template público llamado también generated, que recibe una entidad Class del metamodelo UML (línea 4). Dentro del template se genera un archivo con la directiva file (líneas 7 a 9), cuyo nombre y contenido es el atributo name de la entidad Class recibida como parámetro. La línea 6 indica que el módulo es el puntos de entrada (anotación @main) de la generación.

Además de las directivas, se puede generar texto estático, es decir, sin la intervención del modelo. Así como la etiqueta file, la herramienta ofrece directivas para estructuras de control (for, if), comentarios, la utilización de métodos de las entidades del modelo, de clases Java como servicios y más.

Una característica interesante que ofrece este plugin, es la posibilidad de la generación de

bloques para el usuario, es decir, porciones del código que son sólo generadas una vez y preservadas para las generaciones subsecuentes. Esto le permite al usuario enfocarse sólo en las parte del código que le interesa mientras deja el resto del archivo en control del generador. La herramienta ofrece varias alternativas, una de ellas, es el uso de la directiva protected. La figura 4.5 muestra el código de un módulo donde: el generador produce una clase Java a partir de una clase de un modelo UML, pero no genera el cuerpo de los métodos de esa clase porque el comportamiento es completado por el usuario (y cómo este puede involucrar otras referencias, también se deja como bloque de usuario los imports de la clase). Entonces, si se modifica la entidad en el modelo (por ejemplo, agregando una atributo nuevo), al momento de regenerar el código, el código escrito por el usuario no se pierde.

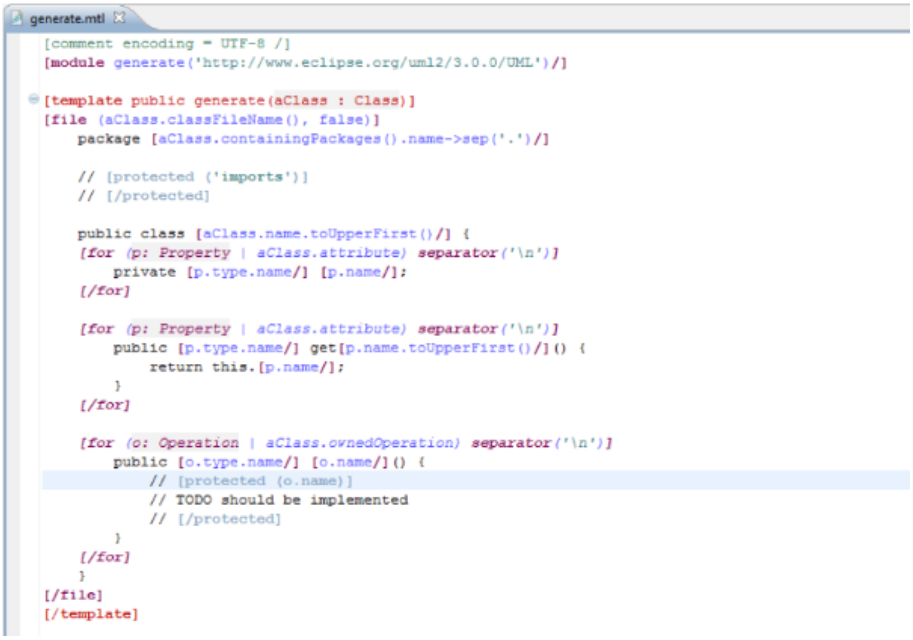

Figura 4.5: Ejemplo de un módulo de Acceleo con bloques de usuario (líneas 8 y 24), identificados por el uso de la directiva protected.

4.3.1 ¿Por qué Acceleo?

- La herramienta se integra fácilmente con Eclipse (simplemente hay que acceder al mercado de la plataforma e instalar el correspondiente plug-in) y, sobre todo, al framework EMF.
- A partir del modelo de entrada, se puede generar fácilmente código para cualquier lenguaje. Es muy sencilla e intuitiva de utilizar.
- Está basada en un estándar internacional de la OMG.
- Hace simple el mantenimiento de un generador de código.

4.4 El Sistema Operativo Robot (ROS)

El Sistema Operativo Robot, más conocido como ROS (en inglés, Robot Operating System) es un framework muy utilizado en el ámbito de la robótica. Su filosofía es la de desarrollar piezas de software que puedan ser utilizadas para programar diferentes robots, realizando pequeños cambios en el código. Entonces, la idea es crear funcionalidad que pueda ser compartida y aplicada en otros robots fácilmente (sin requerir mucho esfuerzo).

ROS originalmente fue desarrollado en el año 2007 por el Laboratorio de Inteligencia Artificial de Stanford [20] (SAIL) con el apoyo del proyecto Stanford AI Robot. A partir del 2008, el instituto de investigación Willow Garage [21] continúa desarrollando el framework con la colaboración de más de 20 instituciones.

Al pasar el tiempo, muchos laboratorios de investigación comenzaron a desarrollar proyectos con ROS, agregando su propio hardware y compartiendo nuevas funcionalidades. Además, varias empresas empezaron a adaptar sus productos para ser compatibles con ROS. Normalmente, estas nuevas funcionalidades son publicadas con un gran cantidad de código, documentación, ejemplos y simuladores, para permitirle al programador una rápida adaptación y comprensión.

El framework es Open Source y fue publicado bajo licencia BSD (Berkeley Software Distribution). Es gratis para uso comercial y para investigación. Esta enfocado a sistemas Unix (pero no todas las distribuciones se encuentran soportadas) y a futuro se planea lograr la portabilidad a Mac OS y Windows (al momento de escribir esta tesina, el desarrollo para estas plataformas estaba indicado como experimental).

Como aclaración, en esta sección sólo se explicarán los conceptos que maneja este framework, dejando de lado la instalación, ejemplos de código y demás.

4.4.1 Conceptos generales

ROS provee las funcionalidades estándar de un sistema operativo:

- Abstracción de hardware.
- Implementación de los algoritmos y las funcionalidades más comúnmente utilizadas.
- Manejo de paquetes.
- Independencia de lenguaje de programación.
- Conjunto de comandos de consola.

La arquitectura de ROS es una topología centralizada, donde el procesamiento ocurre en los nodos, que se comunican (inicialmente) entre sí utilizando un esquema Publish/Subscribe. Por ejemplo, un driver de un sensor puede ser visto como un nodo de la red, que publica los datos del sensor como mensajes en un tópico (tema o canal). Estos mensajes son consumidos por otros nodos, que pueden filtrar esa información, registrarla en un log, o incluso utilizarla para ubicar y guiar al robot por un mapa. En la figura 4.6 se ejemplifica esta situación.

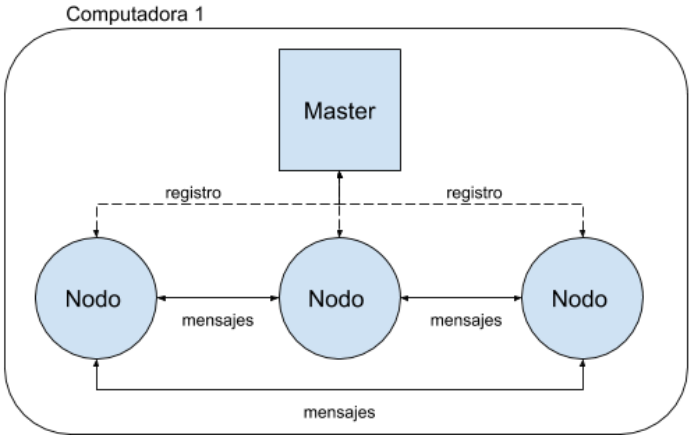

Figura 4.6: Arquitectura de ROS, en un sola computadora.

El nodo central de la topología se denomina Master, que provee el registro de nombres. Es el que permite la identificación y comunicación entre los nodos. Esta comunicación se realiza publicando/consumiendo mensajes en un tópico. Cada mensaje tiene un tipo definido. Toda esta información está registrada en el nodo Master. El framework también ofrece la posibilidad de utilizar servicios (request/reply). Todos los elementos nombrados serán explicados en detalle en la siguiente sección.

Por último, los nodos no necesitan estar en un mismo sistema, sino que pueden estar en múltiples computadores o en diferentes arquitecturas. Esto hace de ROS un framework muy flexible, ya que se puede tener un nodo ejecutando sobre la computadora del robot (por ejemplo, una placa Arduino) consumiendo datos de una cámara y publicando mensajes (con la imagen procesada) y otro nodo en otra computadora (por ejemplo, una notebook, un servidor o incluso un dispositivo con Android) que consume esa información y la visualiza.

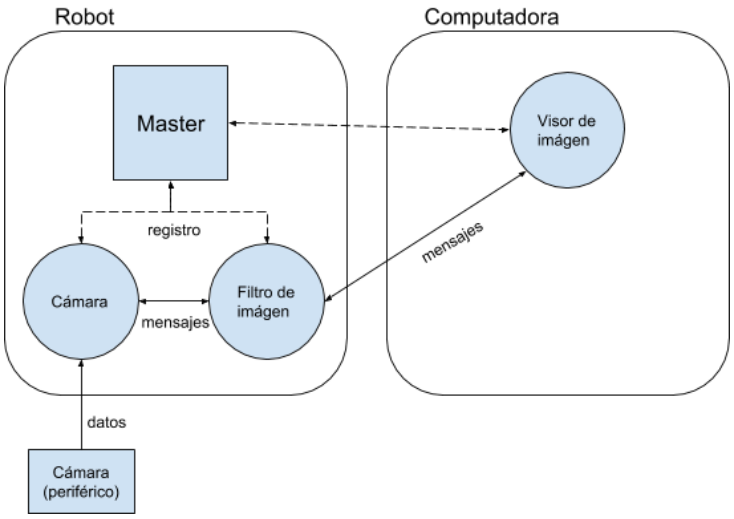

Figura 4.7: Arquitectura de ROS, ahora en dos computadoras.

4.4.2 La arquitectura en detalle

La arquitectura de ROS ha sido diseñada y dividida en tres niveles:

1. El nivel del sistema de archivos (The Filesystem Level).
2. El nivel del grafo computacional (The Computation Graph Level).
3. El nivel de la comunidad (The Community Level).

En el primer nivel, se define como el framework está organizado internamente, la estructura de carpetas y los archivos necesarios para que funcione.

En el segundo nivel es donde ocurre la comunicación entre los procesos y el sistema. Se explican todos los conceptos y formas que tiene ROS para iniciar la topología, administrar los procesos y la comunicación entre estos.

En el tercer nivel, se explican los conceptos que se utilizan para compartir el conocimiento, algoritmos y código.

4.4.3 El nivel del sistema de archivos

Al igual que un sistema operativo, un programa de ROS está dividido en carpetas. Cada una de ellas posee ciertos archivos que describen la funcionalidad.

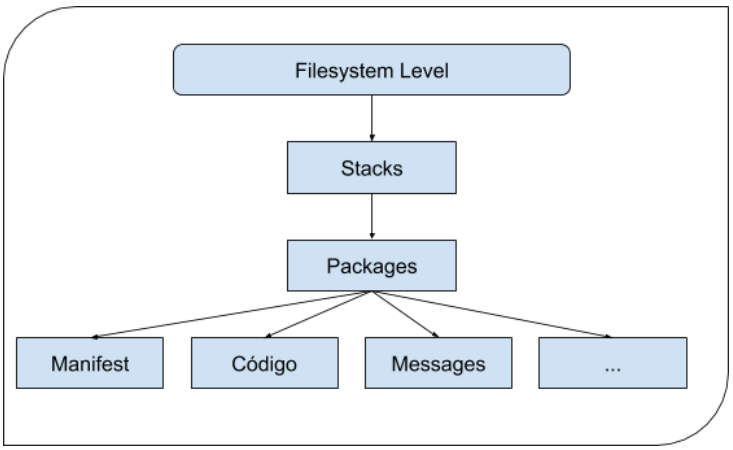

Figura 4.8: Componentes del nivel del sistema de archivos de ROS.

Los Packages (o Paquetes, en castellano) son la unidad básica de la organización del framework. Contienen librerías, procesos ejecutables (nodos), scripts, archivos de configuración, u otros artefactos. Por ejemplo, los archivos fuente se encuentra bajo la carpeta /src.

Cada paquete contiene un Manifest. Estos archivos proveen información (metadatos) acerca de un paquete, como por ejemplo la licencia y los parámetros de compilación. Pero su rol más importante es indicar la dependencia entre paquetes.

Finalmente, los Meta-Packages (antes llamados Stacks) representan una colección de paquetes, que forman una librería de alto nivel. Al final del documento, hablaremos de la Navigation Stack, una librería para la navegación autónoma del robot.

La estructura básica y los archivos indispensables de un package son los siguientes:

- CMakeList.txt: el archivo con las directivas de compilación.
- manifest.xml: el archivo manifest del package.
- bin/: directorio que contiene los programas compilados.
- src/: directorio que contiene los código fuentes (escritos en C++, Python, o JAVA).

- scripts/: directorio que contiene los scripts (Bash, Python).
- include/package_name/: este directorio incluye los headers de librerías referencias en el package (indicadas en el manifest).
- msg/ y srv/: directorios que contiene las definiciones de los types de messages y services (respectivamente).

Para crear, modificar y trabajar con packages, ROS nos provee ciertos comandos:

- rospack: se utiliza para obtener información acerca de un paquete o buscar uno en el sistema.
- roscreate-pkg: se utiliza para crear un nuevo paquete.
- rosmake: se utiliza para compilar un paquete.
- rosdep: se utiliza para instalar todas las dependencias de un paquete.

Además, para desplazarnos entre paquetes y sus carpetas y archivos, el framework nos provee un paquete especial llamado rosbash, que contiene comandos muy similares a los de Unix. Algunos de ellos son: roscd (cambiar de directorio), rosed (editar un archivo), roscp (copiar archivos), rosls (listar archivos de un package).

4.4.4 El nivel del grafo computacional

ROS crea una red donde todos los procesos están conectados entre sí. Cada uno de ellos puede acceder a la red, interactuar con otros nodos, ver la información que se está compartiendo, y transmitir nueva información a la red. Cada uno de los componentes indicados en la figura 4.9 aporta información a la topología, pero de diferente manera.

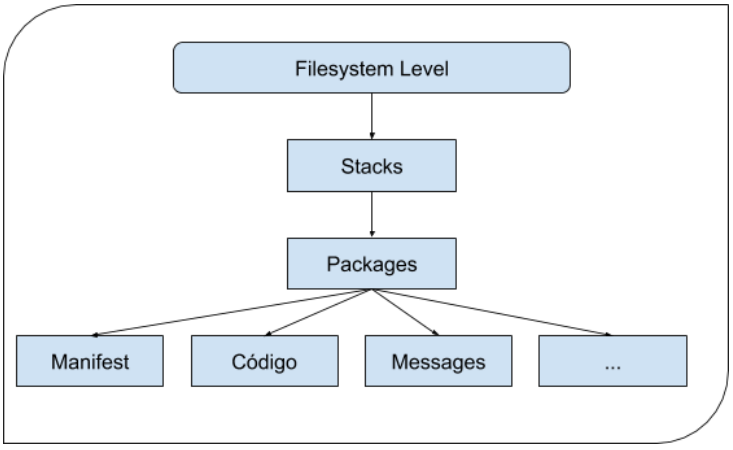

Figura 4.9: Componentes del nivel de grafo computacional de ROS.

Los Nodes (Nodos en castellano) representan a los procesos. Realizan una función específica, desde controlar el motor de una rueda, leer información de un sensor láser, capturar una imagen desde un cámara y hasta planificar una posible ruta de navegación. Cada nodo está escrito en alguna librería cliente de ROS, como roscpp (C++) o rospy (Python).

El nodo central de la topología se denomina Master (también llamado roscore). Posee el registro de nombres de todo el sistema: identificación de los nodos, canales de mensajería y servicios. El objetivo es permitir que cada nodo puede localizar a cualquier otro. Además, si hay algún cambio en la topología (por ejemplo, se incorpora un nuevo nodo), envía una notificación a cada nodo indicando la información actualizada. El Parameter Server es el almacenamiento

(diccionario) compartido por toda la red, administrado por el nodo Master. Los nodos lo utilizan para guardar o recuperar información en tiempo de ejecución.

Los nodos se comunican unos a otros enviando Messages (Mensajes en castellano). Estos son estructuras de datos simples, formados por diferentes campos (similar a un registro). Existen mensajes primitivos (int8, string, array[]) o definidos por el usuario.

Para comunicarse, los nodos publican mensajes en Topics (en castellano Tópicos, también llamados temas o canales). Cuando un nodo quiere recuperar mensajes de un canal, debe subscribirse a este. Entonces, se forma un esquema de envío de mensajes N-a-N (muchos a muchos), es decir, un nodo puede publicar/suscribirse a múltiples canales. Por lo tanto, no es necesaria una conexión directa entre los nodos y esto significa que la producción y el consumo de datos está desacoplado.

Cada tópico tiene un tipo asociado, definido por el tipo del mensaje que se envía a través de él. Los nodos que se suscriben a ese canal, sólo pueden consumir mensajes de ese tipo. La información puede ser transmitida utilizando tanto el protocolo TCP/IP (llamado TCPROS) o UDP. Todo esto es transparente para el programador.

Por último, existe otro tipo de comunicación llamado Services (Servicios en castellano). Es la implementación del paradigma cliente/servidor que provee ROS. Existe un primer mensaje, llamado request y otro llamado reply. Un nodo ofrece un servicio (servidor) a través de un nombre, y otro (cliente) lo utiliza enviando un mensaje request. Luego, espera a recibir el mensaje reply. Funciona como un llamado a un procedimiento remoto.

4.4.5 El nivel de la comunidad

ROS posee distintas distribuciones, es decir, colecciones con diferentes stacks, que los usuarios pueden descargar e instalar. Además, existe una red de repositorios donde diferentes usuarios (que pueden ser individuos, instituciones y empresas) publican nuevas funcionalidades o desarrollos sobre determinados robots.

El framework se encuentra documentado en un foro llamado ROS Wiki [17], donde cualquier usuario puede ingresar (registrando una cuenta) y contribuir con tutoriales, su propia documentación, correcciones, actualizaciones, sugerencias y más.

4.4.6 ¿Por qué ROS?

En primer lugar, ROS es un framework que tiene una gran aceptación por parte de la comunidad, no sólo por el lado de los programadores, sino que los desarrolladores de hardware y grandes empresas construyen componentes compatibles con el framework. Además ya existen robots compatibles con la plataforma utilizados por los usuarios en tareas de la vida cotidiana (por ejemplo, los robots aspiradora).

Con su filosofía de “no reinventar la rueda”, el framework resuelve una gran cantidad de problemas comunes a la hora de programar robots. En los próximos capítulos se discutirá el caso del Stack de Navegación, una solución, para resolver la navegación autónoma de un robot. Lo importante es que, el framework, sigue las ideas expuestas por los artículos comentados en el capítulo anterior, es decir, de separar los componentes del sistema robótico, modelarlos de manera abstracta y reutilizar todo el código que sea posible, siempre respetando la interfaz de cada componente.

Estas dos razones son el porqué de la utilización de ROS como plataforma específica durante el desarrollo de esta tesina.

Capítulo 5

DSL para robots con misión predeterminada

En este capítulo se presenta nuestra solución al problema planteado con la definición de un DSL (Domain Specific Language). En la primer sección se especifican las pautas para la definición del metamodelo, es decir, cómo se representa gráficamente y cómo es descrita la sintaxis abstracta. Luego se presenta el metamodelo, con una explicación conceptual, la especificación de la sintaxis abstracta y un ejemplo de sintaxis concreta.

El lenguaje fue definido mediante EMF, el plug-in de Eclipse descrito en el capítulo 3. En particular, el metamodelo está definido con el meta-lenguaje ECORE.

5.1 Pautas para la definición del metamodelo

5.1.1 Pautas sobre el diagrama

Para la representación gráfica del metamodelo (5.2) se ha utilizado UML, con algunas modificaciones para simplificar la lectura. Estas se detallan a continuación:

- Los nombres de las clases aparecen con la primera letra en mayúscula, mientras que los atributos, operaciones, referencias y asociaciones aparecen con la primera letra en minúscula.
- Las clases son representadas por cajas rectangulares con tres secciones (divididas por una línea) conteniendo el nombre de la clase, los atributos y las operaciones, respectivamente, desde arriba hacia abajo. Si el nombre de una clase se encuentra en itálica, quiere decir que la clase es abstracta. Las operaciones de getters y setters no son mostradas en el diagrama, pero si el resto.
- Los valores de los atributos pueden ser datatype, primitives o enumeration.
- Todas las asociaciones del metamodelo tienen un nombre pero, para facilitar la lectura, estas no se muestran en el diagrama. En cambio, se agrega el nombre del extremo de la asociación como atributo de la clase, con el carácter / (barra invertida) como prefijo. El tipo de ese atributo es el nombre de clase referenciada en ese extremo de la asociación. La figura 5.1 ilustra esta representación. Por último, la multiplicidad de la asociación aparece en el diagrama, así como el sentido y los indicadores de composición (rombo negro) y agregación (rombo blanco).

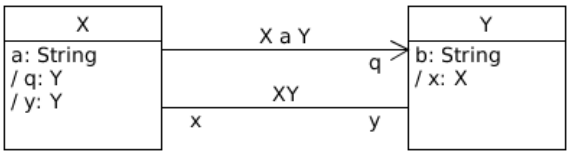

Figura 5.1: Representación de las asociaciones. La asociación q (por ejemplo, una composición) es representada como un atributo de la clase X, con el carácter / (barra invertida) como prefijo y la clase referenciada como tipo. Lo mismo ocurre con la asociación XY, dónde cada extremo se convierte en atributo de la misma manera.

5.1.2 Pautas sobre la especificación de la sintáxis abstracta

Los siguientes apartados describen cómo se definen formalmente los elementos del metamodelo y sus características. Para nuestro caso, las Clases son los elementos fundamentales. Una Clase puede tener atributos, referencias y operaciones. Además, estas pueden estar heredar información de otra Clase o relacionarse con otras mediante asociaciones. Entonces, cada una es definida en términos de su nombre, su ascendencia (superclases), atributos, referencias, operaciones, restricciones y si es abstracta o concreta.

Nombre de la Clase [abstracta]

Cada Clase es introducida por una sección cuyo título es su nombre, que puede ser seguido de la palabra abstracta, si las clase es justamente abstracta. Además, se provee una descripción de lo que representa.

Superclase

Esta sección incluye la herencia de la Clase, es decir, nombra la superclase.

Atributos

Esta sección lista todos los atributos de una Clase, excepto aquellos que son heredados. Las siguiente tabla se utiliza para definir las características de un atributo.

atributo

Descripción	
Tipo	Define el tipo del atributo.
Multiplicidad	Define la multiplicidad del atributo, que consiste en los límites inferior y superior e indicadores de [esOrdered] y [esUnique]. En el caso de una Collection, se agrega el indicador [esOrdered] si se trata de un set ordenado y el indicador de [esUnique] si cada valor es único. Caso contrario, no se agrega el indicador.
Alcance	Indica si el atributo es de instancia o de clase.

Referencias

Esta sección lista todas las relaciones de la clase con otras, excepto aquellas que son heredadas. La siguiente notación se utiliza para definir las características de una referencia. Si se trata de una relación de composición, se agrega la palabra composición al nombre de la relación.

referencia { composición }

Descripción de la referencia	
Clase	Nombre de la clase referenciada.
Multiplicidad	Se define al igual que la multiplicidad en los Atributos.
Inversa	Indica si existe una relación inversa desde la clase referenciada.

Operaciones

Esta sección lista todas las operaciones definidas en la interfaz de la clase, excepto aquellas que son heredadas. La siguiente notación se utiliza para definir las características de una operación.

operación

Comentario de la operación.	
Tipo de retorno	Define el tipo y la multiplicad de retorno.
Parámetros	Se define una descripción, el tipo y la multiplicidad de todos los parámetros que no son de retorno.

Restricciones

Esta sección lista todas las restricciones asociadas a la clase.

5.2 Metamodelo del DSL

En esta sección se presenta el metamodelo del DSL, descrito de acuerdo a las pautas presentadas en la sección anterior. Primero se comenta una visión general del mismo, luego se introduce el diagrama UML y finalmente se describe su sintaxis abstracta junto con un ejemplo de sintaxis concreta.

5.2.1 Visión general

Conceptualmente, un **Robot** tiene que cumplir con una misión pre-determinada, representada justamente por la entidad **Mision**. Ésta posee un **título**, un **autor** y un resumen (una pequeña descripción sobre qué hace la misión, es decir, una **visión general** de esta). La misma se realiza recorriendo ciertas posiciones (**Objetivos**) de un **Mapa** y realizando ninguna, una o más acciones (**Tareas**) cada vez que se alcanza dicha ubicación del mapa. Las acciones posibles a realizar están listadas e identificadas para cada misión.

El mapa no es más que el espacio conocido por el robot, es decir, la representación del mundo real en un plano con dos ejes de coordenadas (x,y), donde la unidad de medida es el metro. La entidad **Punto** representa un punto (x1,y1) del plano.

La lista de **tareas disponibles** está compuesta por un conjunto de **TareaPersonalizada**, que representan a las tareas (definidas por el usuario) disponibles para asignar a un objetivo. La lista de **objetivos** representan al conjunto (ordenado) de ubicaciones a donde el robot debe trasladarse. Cada elemento de ese conjunto está representado con la entidad **Objetivo**. Ésta, cuenta con una ubicación en el espacio (un **Punto**), un **comentario** y un estado de **obligatoriedad** que indica que, si el robot no puede alcanzar la correspondiente ubicación del mapa, entonces la misión debe ser cancelada. Por último, cada Objetivo tiene una serie de **acciones** a ejecutar, representadas por la entidad **Tarea**. Cada una tiene un **nombre** único que la identifica y una breve **descripción** sobre la funcionalidad de la acción que se realiza.

Existen dos tipos de **Tareas**: aquellas cuya funcionalidad está presente en cualquier robot y aquellas cuya funcionalidad depende exclusivamente del sistema robótico. Por ejemplo, la acción de "dormir"(es decir, no hacer nada) durante una cierta cantidad de tiempo (en segundos), se encuentra representada en la entidad **TareaDormir**. Mientras que las acciones dependientes y definidas por el usuario, están representadas por la entidad **TareaPersonalizada**.

Un **Robot** registra su **posición** y **orientación inicial** (mediante los y procede a ejecutar la misión mediante las operaciones de su interfaz, de la siguiente manera:

1. Toma el primero de los **Objetivos** como el objetivo actual.
2. Toma la **ubicación** del objetivo actual y se mueve hacia ese punto del **mapa**.
3. Una vez que llega hacia la **ubicación**, procede a ejecutar cada una de sus **tareas** (en orden). Si el robot no pudo llegar al objetivo y este estaba marcado como **obligatorio**, la misión se cancela.
4. Luego de ejecutar todas la **tareas**, el robot prosigue a tomar el siguiente **objetivo** de la lista como objetivo actual y vuelve al paso 2.
5. Si no hay más objetivos en la lista, el **Robot** verifica la cantidad de veces que debe **repetir** el recorrido de la misión.
6. Si debe repetir la misión, vuelve al paso 1. En caso contrario, termina la misión.

5.2.2 Especificación del metamodelo y su sintaxis abstracta

La figura 5.2 muestra el diagrama UML con las clases, sus atributos y las relaciones correspondientes entre ellas. A continuación se detallan todas las clases del metamodelo, ordenadas alfabéticamente, tal como fue explicado en la sección anterior de este capítulo.

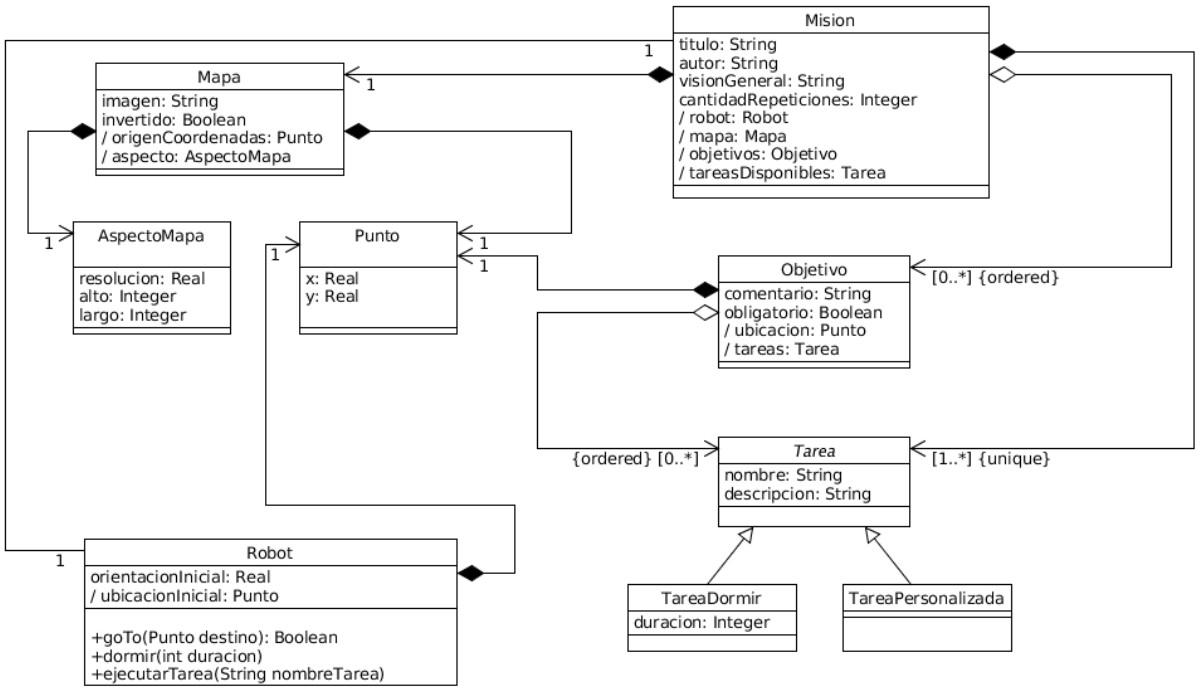

Figura 5.2: Diagrama con las entidades del metamodelo.

---

1. AspectoMapa

Esta clase representa el aspecto de un mapa. Puntualmente, agrupa la longitud de las dimensiones (alto y largo) y la relación entre metros y pixels.

Superclase

Ninguna.

Atributos

alto

Indica el alto del mapa en píxeles.	
Tipo	Integer
Multiplicidad	Exactamente una
Alcance	Instancia

largo

Indica el largo del mapa en píxeles.	
Tipo	Integer
Multiplicidad	Exactamente una
Alcance	Instancia

resolucion

Indica la relación entre metros y pixeles (metros/píxel = resolucion). Por ejemplo, una resolución de 0.1 quiere decir que 10 píxeles representan un metro.	
Tipo	Float
Multiplicidad	Exactamente una
Alcance	Instancia

Referencias

Ninguna.

Operaciones

Ninguna.

Restricciones

Ninguna.

---

2. Mapa

Esta clase modela un mapa, que no es más que una representación del mundo real conocido por el robot. Esta representación viene dada por una imagen y puede ser vista como un plano cartesiano (dos ejes), donde se identifican los obstáculos en los puntos del plano que se corresponden con pixeles negros y en blanco las posiciones libres (o viceversa, de acuerdo al valor del atributo invertido). Los pixeles con colores intermedios son considerados obstácul El mapa cuenta con un origen (origen de coordenadas del plano) y un aspecto (longitud de las ejes y la relación entre los metros del mundo real con los píxeles de la representación). La unidad del mapa son metros.

Para una explicación más detallada de esta entidad, se recomienda leer el anexo A.

Superclase

Ninguna.

Atributos

imagen

Indica la ruta absoluta del archivo de imagen en el sistema de archivos.	
Tipo	String
Multiplicidad	Exactamente una.
Alcance	Instancia

invertido

Indica si se deben tomar los píxeles en blanco como obstáculos y los píxeles en negro como posiciones libres. Por defecto (cuando el valor es false), los obstáculos en el mapa se corresponden con píxeles en negro y las posiciones libres con blanco.	
Tipo	Boolean
Multiplicidad	Exactamente una.
Alcance	Instancia

Referencias

aspecto (composición)

Representa el aspecto del mapa.	
Clase	AspectoMapa
Multiplicidad	Exactamente una.
Inversa	No

origenCoordenadas (composición)

Representa el origen de coordenadas del mapa.	
Clase	Punto
Multiplicidad	Exactamente una.
Inversa	No

Operaciones

Ninguna.

Restricciones

Los valores (x,y) del Punto **origenCoordenadas** deben corresponderse a un píxel válido de la imagen.

---

3. Mision

Esta clase representa el recorrido predeterminado que debe hacer el robot sobre un entorno conocido. Contiene toda la información de la misión: el mapa, el robot, las posiciones, las tareas a ejecutar y más metadatos como por ejemplo, la cantidad de repeticiones.

Superclase

Ninguna.

Atributos

titulo

Nombre e identificador de la misión. Tiene la particularidad de que está formado solo por caracteres alfanuméricos, en minúsculas, y el carácter _ (guión bajo).	
Tipo	String
Multiplicidad	Exactamente una.
Alcance	Instancia

autor

Indica el autor o los autores de la misión.	
Tipo	String
Multiplicidad	Exactamente una.
Alcance	Instancia

visionGeneral

Resumen de la misión. Describe brevemente cuales son los objetivos del recorrido y la tarea a realizar.	
Tipo	String
Multiplicidad	Exactamente una.
Alcance	Instancia

cantidadRepeticiones

Indica la cantidad de veces que se repite el recorrido de la misión. Este valor como mínimo es 1 y como máximo es n.	
Tipo	Integer
Multiplicidad	Exactamente una.
Alcance	Instancia.

Referencias

mapa (composición)

Representa el mapa de la misión.	
Clase	Mapa
Multiplicidad	Exactamente una.
Inversa	No

objetivos

Representa la lista de posiciones del mapa que el robot debe recorrer.	
Clase	Objetivo
Multiplicidad	Cero o más [esOrdered].
Inversa	No

robot (composición)

Representa el robot que va a ejecutar la misión.	
Clase	Robot
Multiplicidad	Exactamente una.
Inversa	Si

tareas (composición)

Representa una lista de tareas que pueden ser ejecutadas durante la misión.	
Clase	Tarea
Multiplicidad	Una o más [esUnique].
Inversa	No

Operaciones

Ninguna.

Restricciones

Dos misiones son consideradas iguales si poseen el mismo **título**.

El atributo **título** de la misión tiene el siguiente formato: String formado solamente por caracteres alfanuméricos en minúsculas y _ (guión bajo).

El valor del atributo **cantidadRepeticiones** tiene que ser mayor o igual a 1.

La lista de **objetivos** está ordenada. Este orden es importante, ya que indica qué posiciones se tienen que alcanzar antes que otras. Define el recorrido.

La lista de **tareasDisponibles** tiene siempre como mínimo un elemento: una instancia de TareaDormir y no contiene elementos repetidos.

---

4. Objetivo

Esta clase representa una posición del mapa que forma parte del recorrido del robot y la cual el robot debe alcanzar. Una vez que llega a dicha ubicación, se ejecutan (o no) una serie de acciones.

Superclase

Ninguna.

Atributos

comentario

Descripción breve de la ubicación y las tareas a ejecutar.	
Tipo	String
Multiplicidad	Exactamente una.
Alcance	Instancia

obligatorio

Indica si se debe cancelar la misión en caso de no alcanzar la ubicación especificada. Por defecto (cuando el valor es false), el recorrido continúa aunque el robot no haya podido llegar a dicha posición.	
Tipo	Boolean
Multiplicidad	Exactamente una.
Alcance	Instancia

Referencias

tareas

Representa la lista de tareas a ejecutar una vez alcanzada la ubicación especificada.	
Clase	Tarea
Multiplicidad	Cero o más [esOrdered].
Inversa	No

ubicacion (composición)

Representa un punto del mapa a donde el robot debe llegar.	
Clase	Punto
Multiplicidad	Exactamente una.
Inversa	No

Operaciones

Ninguna.

Restricciones

La lista de **tareas** está ordenada. Este orden es importante, ya que indica qué tareas se ejecutan antes que otras.

La **ubicacion** debe ser un punto válido perteneciente al mapa y ser relativa al origen de coordenadas del mapa.

---

5. Punto

Representa una punto del plano, dada por coordenadas (x,y).

Superclase

Ninguna.

Atributos

x

Indica el valor de la coordenada en x.	
Tipo	Float
Multiplicidad	Exactamente una.
Alcance	Instancia

y

Indica el valor de la coordenada en y.	
Tipo	Float
Multiplicidad	Exactamente una.
Alcance	Instancia

Referencias

Ninguna.

Operaciones

Ninguna.

Restricciones

Ninguna.

---

6. Robot

Esta clase representa al robot que va a realizar la misión. Esta representación sólo contempla las características comunes para cualquier robot necesarias para realizar el recorrido y ejecutar las acciones.

Para una explicación más detallada de esta entidad, se recomienda leer el anexo A.

Superclase

Ninguna.

Atributos

orientacionInicial

Valor expresado en radianes, que indica la orientación del robot al momento de comenzar la misión (de acuerdo al Mapa y a los puntos cardinales). Se considera el valor 0 como la orientación Norte, continuando en sentido horario.	
Tipo	Float
Multiplicidad	Exactamente una.
Alcance	Instancia

Referencias

ubicacionInicial (composición)

Indica la posición del robot al momento de comenzar la misión.	
Clase	Punto
Multiplicidad	Exactamente uno.
Inversa	No

Operaciones

goTo

El robot se mueve desde la posición actual hasta una posición del mapa recibida como parámetro. Durante este trayecto, el robot evade cualquier tipo de obstáculo, ya sea estático (identificado previamente en el mapa) o dinámico (objeto no identificado en el mapa, que convierte una posición libre en obstáculo). Luego de ejecutar esta operación, se retorna si se pudo llegar a la posición destino.	
Tipo de retorno	Boolean. Exactamente uno. Si el robot pudo alcanzar la posición solicitada, se retorna true. Si el robot no pudo llegar a la posición, se retorna false.
Parámetros	destino: Punto. Exactamente uno. Indica la posición del mapa a donde el robot debe llegar.

dormir

El robot espera (no se mueve ni ejecuta ninguna acción) durante una cantidad de segundos recibida como parámetro. Similar a la función sleep de un proceso del sistema operativo.	
Tipo de retorno	Nulo.
Parámetros	duracion: Integer. Exactamente uno. Indica la cantidad de segundos a esperar. Este valor debe ser mayor a cero.

ejecutarTarea

El robot ejecuta una tarea, cuya identificación recibe como parámetro. Una vez terminada la ejecución de la tarea, sin importar el resultado, el robot queda posicionado en el mismo punto del mapa previo a la ejecución.	
Tipo de retorno	Nulo.
Parámetros	nombreTarea: String. Exactamente uno. Nombre identificador de la tarea. Se corresponde con el atributo nombre de la clase Tarea.

Restricciones

La **ubicacionInicial** es relativa al origen de coordenadas del mapa.

Durante la ejecución del método **goTo**, el robot evade cualquier obstáculo.

Luego de ejecutar el método **ejecutarTarea**, independientemente del resultado de la ejecución, el robot queda posicionado en el mismo punto del mapa y en la misma orientación que tenía previo a la ejecución de la tarea.

---

7. Tarea [abstracta]

Esta clase representa una acción a ejecutar por el robot durante la misión.

Superclase

Ninguna.

Atributos

nombre

Nombre e identficador de una Tarea. Tiene la particularidad de que está formado solo por caracteres alfanuméricos, en minúsculas, y el carácter _ (guión bajo).	
Tipo	String
Multiplicidad	Exactamente una.
Alcance	Instancia

descripcion

Resumen de la funcionalidad de la tarea. Describe brevemente que trabajo realiza el robot durante la ejecución de esta.	
Tipo	String
Multiplicidad	Exactamente una.
Alcance	Instancia

Referencias

Ninguna.

Operaciones

Ninguna.

Restricciones

Dos tareas son consideradas iguales si poseen el mismo **nombre**.

El atributo **nombre** de la tarea tiene el siguiente formato: String formado solamente por caracteres alfanuméricos en minúsculas y _ (guión bajo).

---

8. TareaDormir

Superclase

Tarea

Atributos

duracion

Indica la cantidad de segundos que el robot debe esperar.	
Tipo	Integer
Multiplicidad	Exactamente una.
Alcance	Instancia

Referencias

Ninguna.

Operaciones

Ninguna.

Restricciones

El valor del atributo **nombre**, heredado de la clase Tarea, es siempre el String *"dormir"*.

El valor del atributo **descripcion**, heredado de la clase Tarea, es siempre el String *."El robot suspende la ejecución de la misión durante una cantidad de segundos"*.

---

9. TareaPersonalizada

Esta clase representa cualquiera acción a ejecutar por el robot, que no sea dormir. Se utiliza para generar un template, cuyo código deberá ser completado por el usuario. *El dominio no especifica puntualmente cuales son las acciones que puede realizar el robot, y por esta razón se ha modelado una acción de esa manera.*

Superclase

Tarea

Atributos

Ninguno.

Operaciones

Ninguna.

Restricciones

Ninguna.

5.2.3 Ejemplo de sintaxis concreta del DSL

La figura 5.3 ilustra una misión definida mediante una posible sintaxis concreta del metamodelo. Se puede apreciar que la entidad Mapa es representada con una imagen, mientras que los objetivos son representados con color azul (no obligatorio) y rojo (obligatorio). A la derecha de cada marcador se ubica un cuadro de texto con un número que representa el ordenamiento de los objetivos. Por último, la posición inicial del robot es representada con un marcador de color verde.

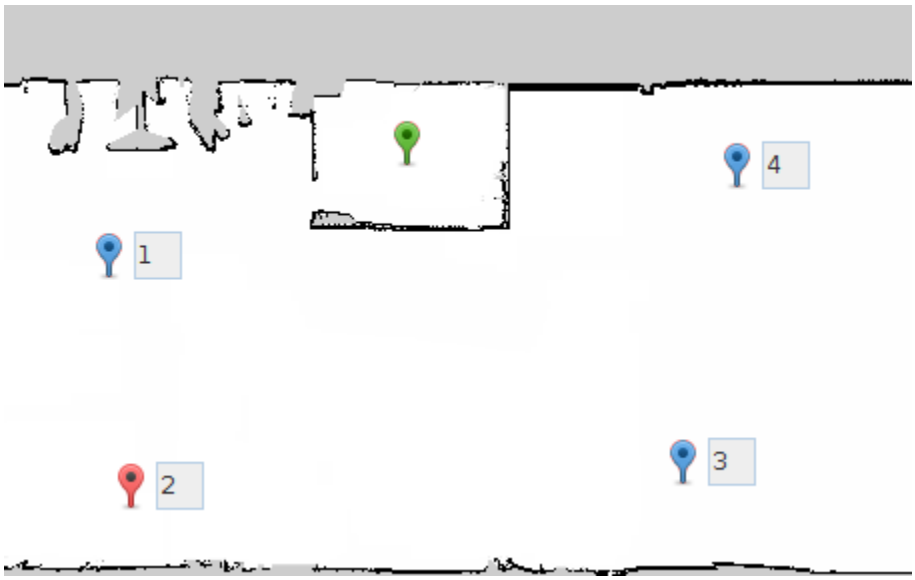

Figura 5.3: Ejemplo de sintaxis concreta para el metamodelo.

En el anexo B de este documento se puede consultar como ejemplo, la sintaxis concreta que define la herramienta gráfica desarrollada para esta tesina.

Capítulo 6

Generación de código para un robot simulado

En este capítulo se presenta un ejemplo de la generación de código para un robot TurtleBot simulado, a partir del DSL definido en el capítulo anterior. En las próximas páginas se explica cómo simular el robot y su entorno. Luego, se detalla cómo se genera el código a partir del modelo, mediante el plug-in Acceleo y finalmente se comentan cómo se ejecuta el programa obtenido.

Uno de los conceptos utilizados en la generación de código es el Stack de Navegación de ROS. Para una descripción de esta funcionalidad que ofrece el framework, se recomienda consultar el anexo A de esta tesina.

6.1 Simulando el robot y su entorno

No siempre es posible contar con un robot real e incluso tampoco trabajar con ciertas condiciones del entorno durante el desarrollo de un sistema robótico (ya sea por cuestiones económicas o físicas). Por esta razón la simulación es un herramienta indispensable en el ámbito de la robótica. Una buena herramienta de simulación hace posible la prueba rápida de algoritmos, favorece al diseño de robots, y sirve de entrenamiento para que el sistema pueda enfrentar situaciones de la vida real.

6.1.1 URDF y Xacro

URDF (acrónimo de Unified Robot Description Format) es un archivo XML, de determinado formato, que describe un robot, sus partes (y las relaciones entre ellas), dimensiones, movimientos y demás. El framework ROS utiliza este tipo de archivos para simular un modelo en tres dimensiones de un robot.

Para definir, por ejemplo, un robot con cuatro ruedas, la cantidad de líneas de un archivo URDF puede llegar a ser bastante grande. Además, si quisiéramos agregarle a ese robot cámaras, brazos u otros componentes, el archivo crecería aún más y sería mucho más complicado entender y mantener ese código. Por esta razón ROS también soporta archivos Xacro, que permiten, entre otras cosas: Definir constantes y macros, para evitar repetir el mismo valor (o componente) y emplear expresiones complejas utilizando operaciones básicas (+, -, *, /).

6.1.2 Gazebo

Gazebo [22] es un simulador de robots y entornos. Permite simular una gran cantidad de robots, sensores y objetos en el mundo tridimensional complejo de manera eficiente. Más importante aún, es que la información generada/simulada por la herramienta (datos de los sensores, colisión de objetos, etc) es muy precisa y cercana a la realidad. Está disponible para su descarga y utilización de manera gratuita. Es compatible con sistemas operativos Linux (en el sitio oficial se recomienda Ubuntu).

El simulador provee una librería de bajo nivel, escrita en C (*libgazebo*), que le permite a otras herramientas y a otros desarrolladores la integración con Gazebo. El framework ROS cuenta con un package para trabajar con este simulador. En la figura 6.1 se muestra una captura de la herramienta.

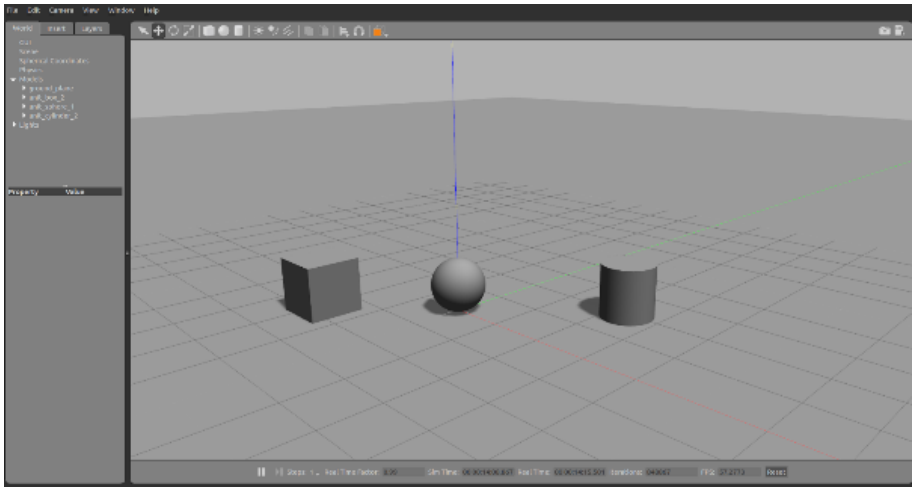

Figura 6.1: Captura del simulador Gazebo.

El término *world* es utilizado para describir una colección de robots y objetos (como edificios, mesas y luces), con un conjunto de parámetros para configurar el cielo, la iluminación y ciertas propiedades físicas. La herramienta permite agregar gráficamente y programáticamente objetos al mundo simulado, llamados *models*. Estos son entidades físicas del entorno que cuentan con ciertas características dinámicas, estáticas y visuales. Además, es posible incorporar *plug-ins* para agregarle funcionalidad a estos objetos. Pueden representar desde un simple cubo, un sensor láser, y hasta un brazo de un robot. Incluso la superficie del entorno (es decir, el suelo) es justamente un *model*. Las dos entidades descritas anteriormente se definen utilizando una notación XML.

Gazebo ofrece una base de datos de objetos, disponible para toda la comunidad de desarrolladores, con modelos y mundos listos para ser descargados y utilizados para la simulación. Además, los usuarios pueden crear y publicar sus objetos, con la restricción de que los mantengan actualizados.

Al instalar ROS, este viene integrado el simulador Gazebo, a través de un paquete del framework llamado justamente *gazebo*. De esta forma, los modelos escritos en URDF o Xacro se visualizan en el simulador. Además, el framework provee paquetes con modelos de simuladores reales, configurados de tal manera que el usuario sólo tiene que especificar el entorno en dónde va a operar el robot.

6.1.3 El robot TurtleBot

A efectos prácticos, para esta tesina se ha instalado el paquete *turtlebot_gazebo*, que ofrece un modelo simulado de un TurtleBot [23]. Este robot consiste de un base móvil, sensores de distancia, una placa controladora (SBC, Single Board Computer) y un kit de extensión para agregar más componentes de hardware. Está preparado, principalmente, para ser compatible con ROS, pero puede ser programado con otros lenguajes ya que tanto el hardware como el software es Open Source. La figura 6.2 muestra las distintas versiones del robot TurtleBot.

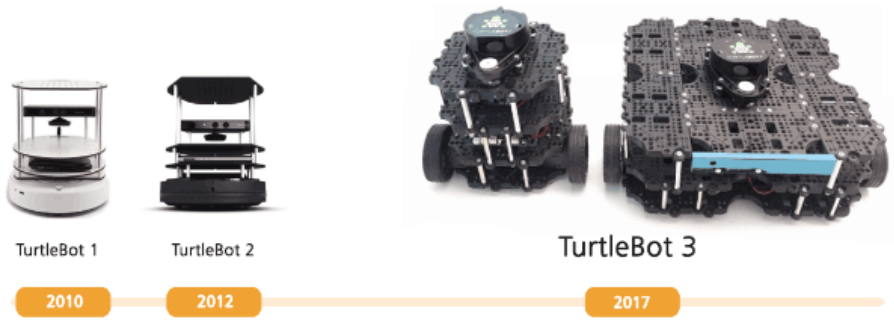

Figura 6.2: Fotografías comparativas de las distintas versiones del TurtleBot.

Para iniciar la simulación de un TurtleBot en Gazebo, basta con ejecutar el siguiente comando indicando como parámetro el path de un archivo *.world* (el entorno, creado con la herramienta gráfica del simulador):

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch world_file:={$PATH_ARCHIVO}
```

Para nuestro ejemplo, se ha utilizado el archivo *circuito.world*. La siguiente figura 6.3 es una captura de la simulación.

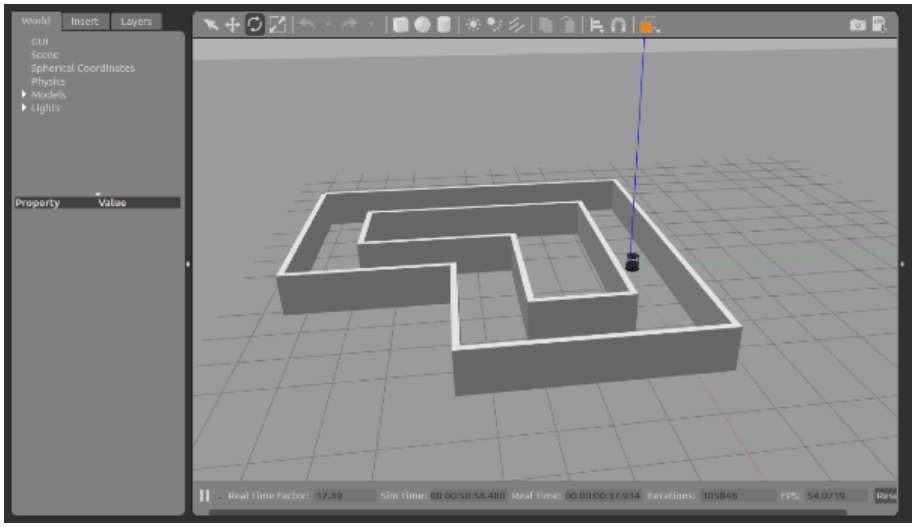

Figura 6.3: Captura de un TurtleBot 2 en el mundo circuito.world, simulado con Gazebo.

Además, el paquete *turtlebot_gazebo*, ofrece una configuración del Stack de Navegación para un TurtleBot simulado, ya lista para ser ejecutada. Hay que recordar que este tipo de robots cumple con los requisitos especificados por el DSL del capítulo anterior. Para inicializar la navegación autónoma, se ejecuta el siguiente comando:

```
$ roslaunch turtlebot_gazebo amcl_demo.launch map_file:={$MAPA}
  initial_pose_x :={$X) initial_pose_y :={$Y) initial_pose_a :={$ORIENTACION)
```

Donde hay que indicar el path a un archivo *.yaml* que representa un mapa para el framework de ROS y la posición y orientación inicial del robot en el mapa. Las posiciones son relativas al origen del mapa y la orientación se ingresa en radianes. La navegación en este caso se basa en el algoritmo AMCL, que es una aproximación de otro algoritmo llamado Monte Carlo Localization (MCL) [8]. Básicamente, se intenta localizar al robot (es decir, estimar su posición actual) en un entorno conocido (mapa) a trávez de la información enviada por sus sensores.

Para obtener un mapa del entorno *circuito.world*, se ha utilizado otro paquete de ROS llamado *turtlebot_teleop* para controlar manualmente al robot (utilizando el teclado o en nuestro caso un joystick de Xbox 360) y, a través de la lectura del sensor láser, construir la imagen del mapa y los metadatos necesario. El algoritmo GMapping es el que se encarga de construir una matriz de ocupación de acuerdo a la lectura de un láser y la odometría que envía el robot. Los comandos empleados fueron los siguientes:

```
$ roslaunch turtlebot_teleop xbox360_teleop.launch
```

```
$ roslaunch turtlebot_gazebo gmapping_demo.launch
```

Para visualizar toda esta información, ROS viene integrado con otra herramienta llamada Rviz (ROS Visualization). Con ella se puede ver los canales (topics) sistema en tiempo real. En la figura 6.4 se muestra una captura de la herramienta, dónde se visualiza la odometría del robot (identificada con una imagen del robot), la distancia y detección de obstáculos emitida por los sensores láser (identificadas con un color rojo y verde) y la matriz de ocupación (representada con una imagen). Para ejecutar la herramienta, hay que ejecutar el siguiente comando en la terminal y ver solamente los componentes referidos a la navegación autónoma del robot):

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

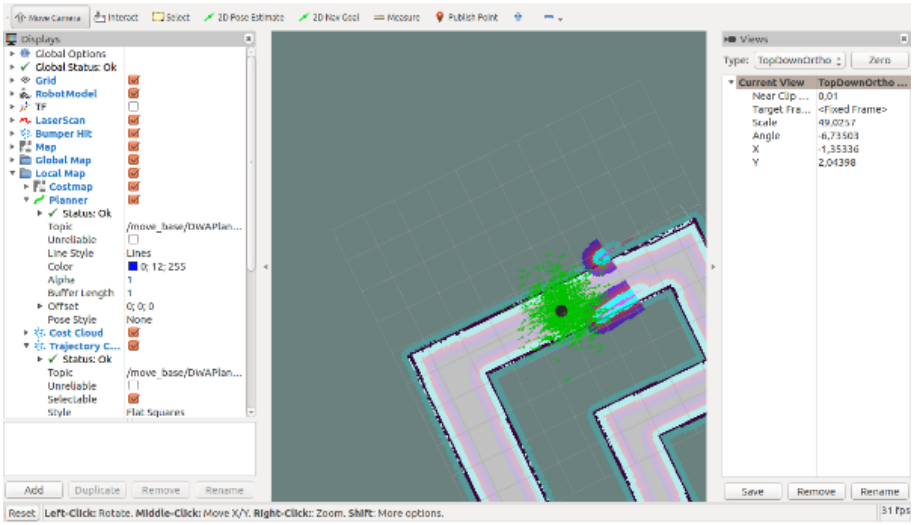

Figura 6.4: Captura de la herramienta Rviz. Se puede observar a la izquierda la información de todos los componentes de la topología (por ejemplo, el sensor láser), en el centro la representación gráfica de estos componentes y en el menú superior algunas acciones como por ejemplo Publish Point, que envía al robot una posición a alcanzar.

Una vez construido el mapa, se guarda el archivo *circuito.yaml* para ser utilizado más adelante, con el siguiente comando:

```
$ rosrun map_server map_saver -f {$PATH}
```

Ahora que se cuenta con un mapa del mundo *circuito.world*, se ejecutan los siguientes comandos para inicializar el Stack de Navegación:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch world_file:={$MUNDO}
```

```
$ roslaunch turtlebot_gazebo amcl_demo.launch map_file:={$MAPA}
  initial_pose_x :={$X) initial_pose_y :={$Y) initial_pose_a :={$ORIENTACION)
```

Para indicarle al robot que se mueva hacia determinada posición, podemos hacerlo programáticamente o gráficamente. Para lo segunda, en Rviz se puede seleccionar un punto del mapa, manteniendo apretado el botón derecho del mouse, para indicar un objetivo en el mapa (aparecerá una flecha de color verde para especificar la orientación del robot). Para hacerlo programáticamente, hay que enviar una petición a un servicio del sistema de navegación. En la próxima sección se explica cómo hacerlo desde un programa escrito en C++.

En este punto, ya se dispone con un robot simulado (TurtleBot) junto con un mundo/entorno (*circuito.world*) y su mapa asociado (*circuito.yaml*). Además, se dispone de una configuración del Stack de Navegación listo para ser ejecutada.

6.2 Generación de código para un TurtleBot simulado

En primer lugar, se define un nuevo *nodo* en la topología de ROS, el cual se va encargar de enviar los objetivos al Stack de Navegación y ejecutar las tareas. Para cada una de ellas se implementa un nodo que ofrece un *servicio*, el cual es invocado por el primer nodo. Luego, se crea el *mapa* de acuerdo al formato especificado por el framework. Por último, se inicializa el Stack de Navegación con el nuevo mapa como parámetro y una vez que el componente está disponible, se ejecuta el nodo con la misión.

Para organizar todo estos elementos, se define un nuevo *package* de ROS, cuyo nombre viene dado por el atributo título de la entidad Misión. Además de los nodos mencionados, es necesario definir ciertos archivos dentro del package, que contienen información acerca de este como las dependencias y directivas de compilación, entre otros.

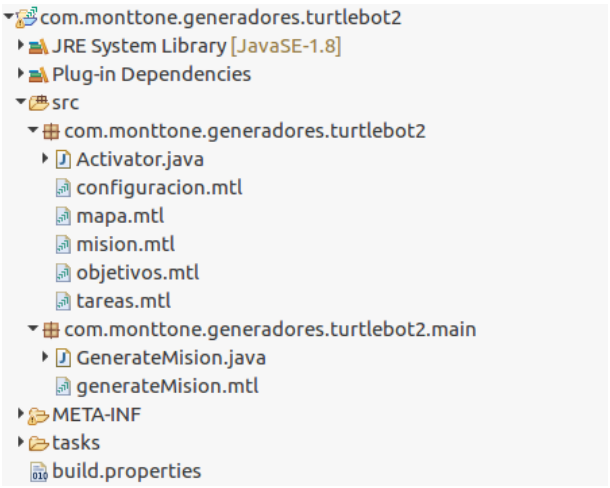

Figura 6.5: Estructura del proyecto Acceleo para la generación de código para un TurtleBot simulado.

El proyecto Acceleo *com.monttone.generadores.turtlebot2*, desarrollado para esta tesina, contiene los módulos que generan cada archivo necesario del package para le ejecución de la misión.

En la figura 6.5, que presenta las estructura de archivos del proyecto, se puede apreciar los diferentes módulos que se encargan de un determinado aspecto del código: *mapa.mtl* genera el mapa, *configuracion.mtl* genera la información adicional del package (metadatos, directivas de compilación y archivos lanzadores), *objetivos.mtl* y *tareas.mtl* se encargan de generar el código que depende de un objetivo y una tarea respectivamente y por último *misión.mtl* genera el código del nodo misión.

El módulo *generateMision.mtl* es el punto de entrada de la generación (comentario main, línea 21, 6.6). Las líneas 15 a 18 importan los módulos mencionados anteriormente, cuyos templates públicos son invocados dentro del propio template *generateMision*. Este módulo, a trávez del método main de la clase GenerateMision es el que una aplicación externa debe invocar para la generación de código indicando como parámetro la ubicación del modelo XMI y del directorio a dónde se crearán los archivos.

```
12
13 [module generateMision('http://www.example.org/misiones')]
14
15 [import com::monttone::generadores::turtlebot2::configuracion/]
16 [import com::monttone::generadores::turtlebot2::mapa/]
17 [import com::monttone::generadores::turtlebot2::mision/]
18 [import com::monttone::generadores::turtlebot2::tareas/]
19
20 [template public generateMision(aMision : Mision)]
21 [comment @main/]
22
23 [manifest(aMision)/]
24 [makefile(aMision)/]
25 [launcher(aMision)/]
26
27 [mapa(aMision)/]
28 [generarTareas(aMision)/]
29
30 [nodoMision(aMision)/]
31
32 [/template]
33
```

Figura 6.6: Módulo principal de Acceleo para la generación del código para el Turtlebot simulado.

En las próximas secciones se detalla cómo se genera cada componente de la solución, con capturas del código de Acceleo y una descripción de su funcionamiento.

6.2.1 El mapa

Para definir un mapa en ROS, se necesita crear un archivo YAML y completar los atributos solicitados por el framework, tal como fue explicado en las secciones anteriores. A continuación se describe el módulo *mapa.mtl*, cuyo único template llamado *mapa* (línea 14, fig. 6.7) recibe como entrada una entidad Mision:

```
11
12 [module mapa('http://www.example.org/misiones')]
13
14 [template public mapa(aMision : Mision)]
15 [file (aMision.titulo.concat('/maps/mapa.yaml'), false, 'UTF-8')]
16 image: [aMision.mapa.imagen /]
17 resolution: [aMision.mapa.aspecto.resolucion /]
18 origin: ['['/]-[aMision.mapa.origenCoordenadas.x/], -[aMision.mapa.origenCoordenadas.y/], 0.0[ ']'/]
19 [if (aMision.mapa.invertido)]
20 negate: 1
21 [else]
22 negate: 0
23 [/if]
24 occupied_thresh: 0.9
25 free_thresh: 0.1
26 [/file]
27 [/template]
```

Figura 6.7: Módulo de generación mapa.mtl (Turtlebot simulado).

Primero se genera un archivo llamado *mapa.yaml* bajo el directorio *maps/* del package (línea 15, fig. 6.7). Los valores de los miembros *image*, *resolution* y *origin* (primeras dos posiciones) del archivo YAML se corresponden con los atributos *imagen*, *resolución* y *ubicacionInicial* (x para la primer posición e y para la segunda) del **Mapa** de la misión (líneas 16 a 18, fig. 6.7). Vale aclarar que el tercer valor del miembro *origin* es completado con 0.0 (sin rotación) porque no se ha modelado la rotación en el mapa.

Luego, si el atributo *invertido* del Mapa es verdadero, se asigna el valor 1 al miembro *negate* (línea 19, fig. 6.7).

Por último, se asignan los valores 0.9 y 0.1 a *occupied_thresh* y *free_thresh* respectivamente, para indicar que el mapa sólo toma como obstáculos pixeles completamente en negro y como posiciones libres píxeles completamente en blanco. Es decir, para no tener en cuenta colores ni escalas intermedias de grises y en definitiva ignorar esta funcionalidad del framework.

6.2.2 Las tareas

Cada **TareaPersonalizada** se transforma en un *nodo* de ROS, que ofrece un *servicio* a la topología. Entonces, se genera un archivo C++ (bajo el directorio *src/* del package) cuyo nombre es el *nombre* de la Tarea más el sufijo *'_srv'*. La figura 6.8 muestra el código del módulo *tareas.mtl*, cuyo template *generarTareas* recibe como entrada una entidad Mision y se encarga de generar los archivos mencionados:

```
12
13 [template public generarTareas(aMision : Mision)]
14 [for (tarea : Tarea | aMision.tareasDisponibles)]
15 [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
16 [file (aMision.titulo.concat('/src/').concat(tarea.nombre).concat('_srv.cpp'), false, 'UTF-8')]
17 #include <ros/ros.h>
18 #include <std_srvs/Trigger.h>
19 //[protected ('for the declarations and definitions of '+ tarea.nombre)]
20 // TODO: Completar las declaraciones/definiciones de la tarea, si corresponde.
21 //[/protected]
22
23 /**
24     Cuerpo del servicio '[tarea.nombre/]_srv'.
25     [tarea.descripcion/]
26
27     @param req, Trigger
28     @param res, Trigger
29     @return true
30 */
31 bool service_callback(std_srvs::Trigger::Request &req, std_srvs::Trigger::Response &res);
32
33 int main(int argc, char **argv) {
34     ros::init(argc, argv, "[tarea.nombre/]_srv");
35     ros::NodeHandle n;
36
37     ros::ServiceServer service = n.advertiseService("[tarea.nombre/]_srv", service_callback);
38
39     ROS_INFO("La tarea '[tarea.nombre/]' esta disponible.");
40     ros::spin();
41 }
42
43 bool service_callback(std_srvs::Trigger::Request &req, std_srvs::Trigger::Response &res) {
44     //[protected ('for the body of '+ tarea.nombre)]
45     // TODO: Completar la funcionalidad de la tarea.
46     //[/protected]
47     return true;
48 }
49 [/file]
50 [/if]
51 [/for]
52 [/template]
```

Figura 6.8: Módulo de generación de las tareas (Turtlebot simulado).

Las primeras dos líneas del template recorren la lista de tareas disponibles de la misión, ignorando aquellas que no sean TareasPredefinidas. En la línea 37 se publica el servicio a la topología y se asigna la función *service_callback* como controladora.

El tipo del servicio es Trigger, predefinido del framework (*package std_srvs*). Su particularidad es que la petición (request) y la respuesta (response) no contiene datos. Los servicios de este tipo son llamados disparadores ya que solo se utilizan para invocar a un servicio sin ningún tipo de parámetro ni intercambio de datos. Al modelar las TareasPersonalizadas de una forma abstracta (de acuerdo al dominio planteado), el tipo Trigger resulta conveniente para modelar esta invocación a una tarea.

Se recuerda que el usuario debe completar el código de la función *service_callback*. El código será preservado frente a la re-generación del código gracias a las directivas *protected* del template (líneas 44 a 46). Existe otro bloque *protected* para la declaración y definiciones que se necesiten en el cuerpo de la función (líneas 19 a 21).

Las líneas 34 y 40 son parte de la interfaz de ROS para inicializar un nodo (registrarlo en la topología) y para loopear (evitando la finalización del programa y permitiendo la atención de los llamados al servicio), respectivamente.

La TareaDormir es representada con una funcionalidad que provee el framework, explicada en el siguiente apartado.

6.2.3 El nodo misión

El módulo *mision.mtl* genera el nuevo nodo de la topología, que ejecuta la **Misión**. Para ello, se precisa definir el nodo mediante el archivo *mision.cpp* ubicado bajo el directorio *src/*. De esto se encarga el template *nodoMision* (línea 19, fig. 6.9), que recibe como parámetro una entidad Mision.

```
12 [module mision('http://www.example.org/misiones')]
13 [import com::monttone::generadores::turtlebot2::objetivos/]
14
15 [template public nodoMision(aMision : Mision)]
16 [file (aMision.titulo.concat('/src/mision.cpp'), false, 'UTF-8')]
17 [declaraciones(aMision)/]
18
19 int main(int argc, char** argv){
20     ros::init(argc, argv, "mision");
21     ros::NodeHandle n;
22     ros::spinOnce();
23
24     std_srvs::Trigger trigger_req;
25     [for (tarea : Tarea | aMision.tareasDisponibles )]
26     [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
27     ros::ServiceClient [tarea.nombre/]_client = n.serviceClient<std_srvs::Trigger>("[tarea.nombre/]_srv");
28     [/if]
29     [/for]
30
31     [for (tarea : Tarea | aMision.tareasDisponibles )]
32     [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
33     while(![tarea.nombre/]_client.waitForExistence(ros::Duration(5.0))){};
34     [/if]
35     [/for]
36
37     ROS_INFO("La mision '[aMision.titulo/]' ha comenzado.");
38
39     for( int x = 0; x < [aMision.cantidadRepeticiones/]; x++ ) {
40     [for (objetivoActual : Objetivo | aMision.objetivos)]
41         [procesarObjetivo(objetivoActual, aMision)/]
42     [/for]
43     }
44
45     ROS_INFO("La mision '[aMision.titulo/]' ha sido ejecutada correctamente.");
46     return 0;
47 }
48
49 [moveToGoal(aMision)/]
50 [/file]
51 [/template]
```

Figura 6.9: Módulo de generación del nodo misión (Turtlebot simulado).

En primer lugar, se incluye otro template del módulo llamado *declaraciones* (línea 17, fig. 6.9), que genera las referencias requeridas por ROS, las necesarias para utilizar el Stack de Navegación, para invocar las tareas (el tipo de servicio Trigger) y las definiciones de las variables y funciones. A continuación se muestra el código del template:

```
64 [template private declaraciones(aMision : Mision)]
65 #include <ros/ros.h>
66 #include <move_base_msgs/MoveBaseAction.h>
67 #include <actionlib/client/simple_action_client.h>
68 #include <std_srvs/Trigger.h>
69 #include <string>
70
71 bool goal_reached;
72
73 /**
74     Invoca al Stack de Navegación para mover al robot
75     hacia una determinada ubicación(x,y).
76
77     @param x_goal, coordenada en x de la ubicación destino
78     @param y_goal, coordenada en y de la ubicación destino
79     @return true/false dependiendo de si el robot llegó o no
80 */
81 bool move_to_goal(double x_goal, double y_goal);
82 [/template]
```

Figura 6.10: Template para las declaraciones y definiciones del archivo C++ (Turtlebot simulado).

En la línea 25 (fig. 6.9), por cada TareaPredefinida de la lista de *tareasDisponibles* de la misión, se define un cliente para cada uno de los servicios ofrecidos por los nodos que representan la tarea (línea 27, fig. 6.9). Y para sincronizar el nodo misión con el resto de los nodos tarea (es decir, que el primero espere hasta que todos los demás estén disponibles) se ejecuta la sentencia del framework *waitForExistence* (línea 33, fig. 6.9).

Una vez que las tareas se encuentran disponibles para ser ejecutadas, se realiza una iteración cuyo número está definido por el atributo *cantidadRepeticiones* de la Misión (línea 39, fig. 6.9). Dentro de este loop se recorre la lista de *objetivos* de la Misión y para cada uno se invoca al template *procesarObjetivo* (línea 41, fig. 6.9) del módulo *objetivos.mtl*, que recibe como entrada una entidad **Objetivo**:

```
12
13 [template public procesarObjetivo(aObjetivo : Objetivo, aMision : Mision)]
14 /*
15     [aObjetivo.comentario/]
16 */
17
18 goal_reached = move_to_goal([aObjetivo.ubicacion.x/], [aObjetivo.ubicacion.y/]);
19 if(goal_reached) {
20     ROS_INFO("El robot ha llegado al punto ([aObjetivo.ubicacion.x/], [aObjetivo.ubicacion.y/].)");
21     [procesarTareasObjetivo(aObjetivo)/]
22 }else{
23     ROS_INFO("El robot NO pudo llegar al punto ([aObjetivo.ubicacion.x/], [aObjetivo.ubicacion.y/]).");
24     [if (aObjetivo.obligatorio)]
25         ROS_INFO("El objetivo era obligatorio. Se cancela la mision");
26         return 1;
27     [/if]
28 }
29 [/template]
30
```

Figura 6.11: Template para el tratamiento de un objetivo (Turtlebot simulado).

Primero se incrusta el comentario del Objetivo (línea 15, fig. 6.11). Luego, se ejecuta la función *move_to_goal*, que recibe como parámetro la posición en x y en y del *objetivo*. El cuerpo de la función será explicado más adelante, pero en líneas generales, se invoca al Stack de Navegación, que se encarga de mover al robot hasta la posición (x,y) del mapa. Se recuerda que la posición de la entidad Objetivo es relativa al origen del mapa, como así lo requiere la interfaz del framework.

Si el robot pudo alcanzar el punto determinado del mapa, la función retorna true y en caso contrario, false. Si ocurre lo segundo y si el atributo *obligatorio* del Objetivo es verdadero (línea 24, fig. 6.11), se cancela la misión (con la sentencia *return*). Ahora, si el resultado de la función es verdadero, se genera el código para ejecutar las Tareas de ese Objetivo, mediante el template *procesarTareasObjetivo* (línea 21, fig. 6.11) del módulo *objetivos.mtl*, que recibe como entrada una entidad Objetivo:

```
30
31 [template private procesarTareasObjetivo(aObjetivo : Objetivo)]
32 [for (tarea : Tarea | aObjetivo.tareas)]
33     [if (tarea.nombre.equalsIgnoreCase('dormir'))]
34         ros::Duration([tarea.oclAsType(TareaDormir).duracion/]).sleep();
35     [else]
36         if (![tarea.nombre/]_client.call(trigger_req)) {
37             ROS_ERROR("La tarea '[tarea.nombre/]' no pudo ser ejecutada.");
38         }
39     [/if]
40 [/for]
41 [/template]
```

Figura 6.12: Template para el tratamiento de las tareas de un objetivo (Turtlebot simulado).

Básicamente, se recorre la lista de **Tareas** y si el atributo identificador *nombre* de la tarea es el literal *'dormir'* (que indica justamente que se trata de una **TareaDormir**) se ejecuta el método *sleep*, de la clase Duration del framework, que recibe como parámetro la cantidad de segundos mediante el atributo *duracion* de la TareaDormir (línea 34, fig. 6.12).

Si el nombre no es *'dormir'*, se trata de una **TareaPersonalizada** y por lo tanto se genera una línea que invoca al servicio del nodo tarea determinado (línea 36, fig. 6.12). La función de invocación se llama *call*: recibe como parámetro el tipo de dato definido para el servicio (en este caso, Trigger) y retorna true o false dependiendo de si se pudo invocar al servicio o no (no indica el resultado de la tarea, sino si hubo algún problema de conexión entre los nodos).

Para finalizar esta sección, se presenta el cuerpo de la función *move_to_goal*, similar al ejemplo del uso del Stack de Navegación explicado en el anexo A, exceptuando que la posición en x (*x_goal*) y en y (*y_goal*) se encuentra parametrizada (líneas 95 y 96, fig. 6.13). El template *moveToGoal* pertenece al módulo *mision.mtl*.

```
84 [template private moveToGoal(aMision : Mision)]
85 bool move_to_goal(double x_goal, double y_goal){
86
87    actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac("move_base", true);
88    while(!ac.waitForServer(ros::Duration(5.0))){}
89
90    move_base_msgs::MoveBaseGoal goal;
91
92    goal.target_pose.header.frame_id = "map";
93    goal.target_pose.header.stamp = ros::Time::now();
94
95    goal.target_pose.pose.position.x =  x_goal;
96    goal.target_pose.pose.position.y =  y_goal;
97    goal.target_pose.pose.position.z =  0.0;
98    goal.target_pose.pose.orientation.x = 0.0;
99    goal.target_pose.pose.orientation.y = 0.0;
100   goal.target_pose.pose.orientation.z = 0.0;
101   goal.target_pose.pose.orientation.w = 1.0;
102
103   ac.sendGoal(goal);
104   ac.waitForResult();
105
106   return (ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED);
107 }
108 [/template]
```

Figura 6.13: Template del cuerpo de la función move_to_goal.

6.2.4 Otros archivos

Para completar la estructura del *package*, es necesario generar más archivos mediante el módulo *configuración.mtl*. El template *manifest* (fig. 6.14) genera el archivo *manifest.xml*, que contiene los metadatos (información, dependencias, etc). Recibe como entrada una entidad Mision.

```
12
13 [template public manifest(aMision : Mision)]
14 [file (aMision.titulo.concat('/manifest.xml'), false, 'UTF-8')]
15 <package>
16   <description brief="[aMision.titulo/]">[aMision.visionGeneral/]</description>
17   <author>[aMision.autor/]</author>
18   <license>BSD</license>
19   <review status="unreviewed" notes=""/>
20   <url>http://ros.org/wiki/[aMision.titulo/]</url>
21   <depend package="move_base_msgs"/>
22   <depend package="actionlib"/>
23   <depend package="roscpp"/>
24 </package>
25 [/file]
26 [/template]
27
```

Figura 6.14: Template de la información del package (Turtlebot simulado).

Los tags se completan con los atributos de la entidad recibida como parámetro (*titulo*, *visionGeneral*, *autor*, líneas 16, 17 y 20). Como dependencias en las líneas 22 y 23 se nombra a *actionlib* y *move_base_msgs* (necesarias para el Stack de Navegación).

Luego, el template *makefile* (fig. 6.15) genera otros dos archivos necesarios para la compilación: Makefile (directivas comunes) y CMakeList (directivas específicas de ROS). También reciben como entrada una entidad Mision.

```
27
28 [template public makefile(aMision : Mision)]
29 [file (aMision.titulo.concat('/Makefile'), false, 'UTF-8')]
30 include $(shell rospack find mk)/cmake.mk
31 [/file]
32
33 [file (aMision.titulo.concat('/CMakeLists.txt'), false, 'UTF-8')]
34 cmake_minimum_required(VERSION 2.4.6)
35 include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
36
37 # Set the build type.  Options are:
38 #  Coverage       : w/ debug symbols, w/o optimization, w/ code-coverage
39 #  Debug          : w/ debug symbols, w/o optimization
40 #  Release        : w/o debug symbols, w/ optimization
41 #  RelWithDebInfo : w/ debug symbols, w/ optimization
42 #  MinSizeRel     : w/o debug symbols, w/ optimization, stripped binaries
43 #set(ROS_BUILD_TYPE RelWithDebInfo)
44
45 rosbuild_init()
46
47 #set the default path for built executables to the "bin" directory
48 set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
49 #set the default path for built libraries to the "lib" directory
50 set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
51
52 #uncomment if you have defined messages
53 #rosbuild_genmsg()
54 #uncomment if you have defined services
55 rosbuild_gensrv()
56
57 rosbuild_add_executable([aMision.titulo/] src/mision.cpp)
58 [for (tarea : Tarea | aMision.tareasDisponibles)]
59 [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
60 rosbuild_add_executable([tarea.nombre/]_srv src/[tarea.nombre/]_srv.cpp)
61 [/if]
62 [/for]
63 rosbuild_add_boost_directories()
64 rosbuild_link_boost(${PROJECT_NAME} system thread)
65 [/file]
66 [/template]
67
```

Figura 6.15: Template de las directivas de compilación (Turlebot simulado).

Del archivo CMakeLists.txt (línea 33), se aprecian las directivas para compilar los nodos que van a formar parte de la topología (*rosbuild_add_executable*, línea 57), indicando el nombre y el archivo fuente. En nuestro caso, el nodo misión y los nodos que representan una TareaPredefinida. Los últimas dos líneas (63 y 64) incluyen librerías del sistema operativo requeridas por el Stack de Navegación.

Por último, se genera un archivo lanzador mediante el template *launcher* (fig. 6.16). La primera línea se encarga de ejecutar cada nodo con la TareaPredefinda, mientras que la última pone en ejecución el nodo misión.

También se incluye otro archivo lanzador en la línea 85, que a su vez se encarga de ejecutar todo lo necesario para el Stack de Navegación. Cómo parámetro se especifica la ubicación del archivo YAML que modela el mapa (referenciado por el nombre del package, línea 86), la *posición inicial* del robot (líneas 87 y 88) y la *orientación inicial* del robot (en radianes, línea 89). Estos últimos tres valores son obtenidos a partir de la entidad **Robot**.

```
75
76 [template public launcher(aMision : Mision)]
77 [file (aMision.titulo.concat('/launch/iniciar_mision.launch'), false, 'UTF-8')]
78 <launch>
79 [for (tarea : Tarea | aMision.tareasDisponibles)]
80     [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
81     <node name="[tarea.nombre/]_srv" pkg="[aMision.titulo/]" type="[tarea.nombre/]_srv"/>
82     [/if]
83 [/for]
84
85     <include file="$(find turtlebot_gazebo)/launch/amcl_demo.launch">
86         <arg name="map_file" value="$(find [aMision.titulo/])/maps/mapa.yaml"/>
87         <arg name="initial_pose_x" value="[aMision.robot.ubicacionInicial.x/]"/>
88         <arg name="initial_pose_y" value="[aMision.robot.ubicacionInicial.y/]"/>
89         <arg name="initial_pose_a" value="[aMision.robot.orientacionInicial/]"/>
90     </include>
91
92     <node name="[aMision.titulo/]" pkg="[aMision.titulo/]" type="[aMision.titulo/]"/>
93 </launch>
94 [/file]
95 [/template]
96
```

Figura 6.16: Template del archivo lanzador (Turtlebot simulado).

6.3 Ejecutando la misión

El programa escrito en Acceleo genera un package de ROS. Este posee un archivo launcher que se encarga de ejecutar:

- Los nodos que representan las tareas y que ofrecen un servicio que puede ser invocado por cualquier nodo de la topología.
- El Stack de Navegación, configurado con la posición y orientación inicial del robot y que administra el mapa también generado por el programa.
- Un nodo misión.

Este nodo misión, se encarga de mover al robot a ciertas posiciones del mapa (a traves del Stack de Navegación) y ejecutar lo servicios de los nodos correspondientes a esa posición del mapa. Este circuito se repite una cantidad determinada de veces.

Entonces, para ejecutar la misión, primero se debe completar el código de las TareasPredefindas (es decir, los archivos del directorio *src/* terminados en *'_srv'*). Luego se ejecuta el siguiente comando (ubicado en el directorio raíz del package):

```
$ rosmake
```

Una vez que la compilación ha terminado correctamente, se ejecuta el archivo lanzador de la siguiente manera. Supongamos que el título de la Misión es *'mi_primera_mision'*. Vale aclarar que previamente debe estar inicializado Gazebo con el mundo y el robot simulado (de la misma manera en que debería estar un robot real conectado a la computadora).

```
$ roscore
$ roslaunch mi_primera_mision iniciar_mision
```

De esta forma, se ejecutan todos los nodos mencionados anteriormente y comienza efectivamente la misión.

Capítulo 7

Generación de código para un robot real

Como alternativa al ejemplo de la simulación, presentado en el capítulo anterior, a continuación se presenta la generación de código para el robot iRobot Create 1. La figura 7.1 es una captura del robot utilizado para este ejemplo. La primer sección de este capítulo comenta las características del robot, tanto las físicas como aquellas que interesan al programador. En la segunda sección se detalla cómo se genera el código a partir del modelo, mediante el plug-in Acceleo y finalmente se describe paso a pso cómo ejecutar el programa obtenido.

Figura 7.1: iRobot Create 1.

7.1 iRobot Create 1

Esta serie de robots fue fabricada en el año 2007 por la compañía iRobot [25]. Este modelo consiste en una modificación de la serie Roomba, el robot aspiradora de la misma empresa. Create 1 reemplaza todo el hardware de la aspiradora con una placa que provee un puerto serial

(DB-25), entrada y salida digital, entrada y salida analógica y una fuente de energía eléctrica. Hay que destacar que este modelo no es compatible por defecto con ROS, es decir, no es *plug-and-play* como si lo es un TurtleBot. La figura 7.2 representa un diagrama con los componentes del robot. De este se destaca la placa nombrada anteriormente (llamada *Cargo Bay Conector*), las dos ruedas (movimiento hacia atrás o adelante) y los bumpers o chocadores (*Cliff Sensors*).

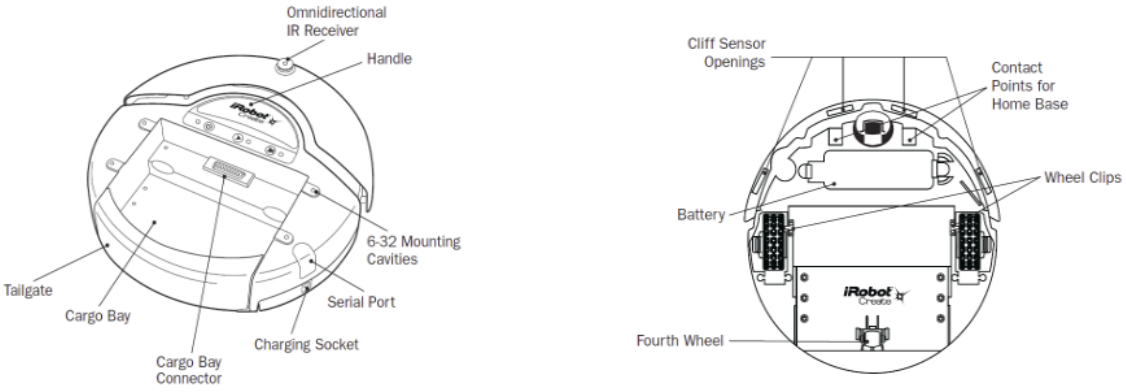

Figura 7.2: Diagrama con las principales partes del robot. A la izquierda, el robot visto desde arriba. A la derecha, el robot visto por debajo.

La idea de este robot es que sea sencillo de programar, para que los usuarios puedan dotar fácilmente de nueva funcionalidad al robot, adaptarlo a sus necesidades particulares y sobre todo, no preocuparse de la programación a bajo nivel. Para lograr esto, el robot cuenta con una interfaz (*iRobot Create's Open Interface*, OI [26]) que provee un conjunto de comandos sencillos para, entre otras cosas, mover y girar al robot, reproducir sonidos y leer información de los sensores. Para agregar otros componentes de hardware (brazo, cámaras u otros sensores), el usuario debe conectarlos al *Cargo Bay* y programar la funcionalidad desde cero.

Básicamente, para programar este robot, se debe conectarlo a una computadora (mediante un cable serial), abrir un programa que sea capaz de enviar datos a través de este protocolo y configurarlo de acuerdo a la especificación del manual (57600 baudios, 8 bits de datos, 1 bit de parada y sin control de flujo). Como ejemplo, enviando la siguiente serie de comandos, se consigue que el robot se mueva hacia adelante:

```
128 131
137 0 100 128 0
```

Para una completa referencia sobre la interfaz, se puede consultar el manual desde la página de iRobot [26].

7.1.1 Compatibilidad con ROS

De acuerdo a la sección anterior, se puede escribir un programa en cualquier lenguaje de programación que, a trávez del envío y recepción de datos por el cable serial, implemente algoritmos complejos complejos en base a la interfaz del robot. Autonomy Lab [27], un laboratorio de investigación ubicado en Canadá, ha implementado un driver para hacer compatible varios modelos de robots con ROS (C++). Entre estos modelos, se encuentra iRobot Create 1. El repositorio del proyecto (*create_autonomy* [28]) se puede descargar de manera gratuita.

De esta manera, se puede conectar al robot con el sistema ROS (mediante el cable serial), ejecutar los drivers de Autonomy Lab y programarlo en un nivel mayor de abstracción, además de reutilizar todos los componentes (algoritmos complejos) que ofrece el framework.

Para realizar la conexión, se debe configurar e instalar el driver como se explica en la documentación de Autonomy Lab y luego ejecutar el siguiente comando:

```
$ roslaunch ca_driver create_1.launch
```

7.1.2 La navegación en iRobot Create 1

Inicialmente, la idea de la implementación del componente de navegación fue emplear el Stack de Navegación. Pero el problema que presenta el iRobot Create 1 es la carencia de un láser y este es un requisito indispensable para poder trabajar con la nombrada funcionalidad del framework. Las alternativas que se barajaron para solucionar este problema fueron: conseguir y configurar (o adaptar) un sensor láser, simular un láser en el sistema, utilizar los bumpers como detección de colisiones o implementar los algoritmos ignorando el láser. Las primeras tres opciones no funcionaron.

Al implementar el componente de navegación desde cero, sin utilizar un láser o los bumpers, se han encontrado ciertas limitaciones, que se enumeran a continuación. Se recuerda que la resolución del problema de la navegación para este robot particular no es el objetivo de esta tesina.

1. **El robot no detecta, y por lo tanto no esquiva, obstáculos dinámicos**. Esto se debe principalmente a la falta de sensores y a la complejidad del algoritmo que resuelve el problema de *"detectar obstáculo y calcular una ruta para evitarlo"*.
2. **El manejo de la localización del robot (es decir, dónde está ubicado y a que orientación apunta en el mapa) es simple**. Se basa en un manejo manual del estado o configuración actual del robot: es decir, no se utiliza un algoritmo como MCL, que ubica al robot en el mapa de acuerdo a varios valores. Si bien el driver *create_autonomy* calcula la ubicación del robot de acuerdo al movimiento de los motores de las ruedas (publicándolo en el canal y */odom*), en algunas pruebas que se realizaron, estos valores no eran acertados y por lo tanto no confiables.
3. **El manejo del movimiento del robot no es preciso**, existe un pequeño margen de error en la distancia recorrida (en cm). Además, solo puede moverse hacia adelante o hacia atrás, y girar siempre a 90 grados.

Teniendo en cuenta estas limitaciones, se construye una arquitectura, similar al Stack de Navegación y a lo recomendado en los artículos del capítulo 3). La figura 7.3 representa los componentes que intervienen en nuestra solución.

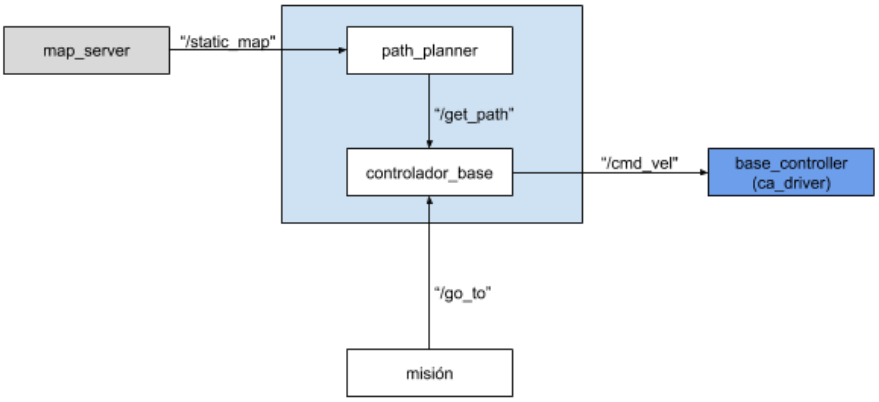

Figura 7.3: Diagrama con componentes y relaciones del sistema de navegación. Con gris se identifican los componentes provistos por ROS, con azul los componentes provistos por el driver del Create 1 (nodo ca_driver) y con blanco los componentes propios de la navegación.

- Se utiliza el *nodo map_server* de ROS para administrar el mapa (al igual que el Stack de Navegación). Específicamente, se invoca al servicio *static_map* que retorna una estructura de datos que contiene los metadatos del mapa y la matriz de ocupación.
- Se define un componente llamado *path_planner*, que dado dos puntos del mapa, calcula la ruta óptima evitando los obstáculos del mapa. Este nodo utiliza el servicio de *map_server*. Para procesar una matriz de ocupación, se necesita un algoritmo de caminos para grafos. En nuestro caso, se optó por el algoritmo de Djikstra (al igual que el Stack de Navegación). Este componente ofrece un servicio a la topología, llamado *get_path* que retorna el camino (si existe) entre dos puntos del mapa recibidos como parámetro.
- Se define un componente llamado *controlador_base*, que dado dos puntos del mapa, se encarga de enviar los comandos de movimiento necesarios para que el robot alcance esos puntos. Este nodo utiliza los servicios de *path_planner*, particularmente, se invoca a get_path.
- Se utilizan, al igual que el Stack de Navegación, el canal */cmd_vel*, que representa comandos de movimiento para el robot, publicados por el nodo *controlador_base* y consumidos por el driver *create_autonomy*.

Entonces, al igual que el ejemplo anterior, la misión consiste en un nodo de ROS que invoca los servicios del *controlador_base*, indicando hacia qué punto del mapa se desea ir.

7.2 Generación de código para iRobot iCreate 1

La generación es similar al ejemplo del capítulo anterior para el TurtleBot 2 simulado.

En primer lugar, se define un nuevo *nodo* en la topología de ROS, el cual se va encargar de enviar los objetivos al componente de navegación y ejecutar las tareas. Para cada una de ellas se implementa un *nodo* que ofrece un *servicio*, el cual es invocado por el primer nodo. Luego, se crea el *mapa* de acuerdo al formato especificado por el framework. Por último, se inicializa el componente de navegación con el nuevo mapa como parámetro y una vez que el componente está disponible, se ejecuta el nodo con la misión.

Para organizar todo estos elementos, se define un nuevo *package*, cuyo nombre viene dado por el atributo *título* de la entidad Misión. Además de los nodos mencionados, es necesario definir ciertos archivos dentro del package, que contienen información acerca de este como las dependencias y directivas de compilación, entre otros.

El proyecto Acceleo *com.monttone.generadores.create1*, desarrollado para esta tesina, contiene los *módulos* que generan cada archivo necesario del *package* para le ejecución de la misión. En la figura 7.4, que presenta las estructura de archivos del proyecto, se puede apreciar los diferentes módulos que se encargan de un determinado aspecto del código: *mapa.mtl* genera el mapa, *configuracion.mtl* genera la información adicional del *package* (metadatos, directivas de compilación y archivos lanzadores), *objetivos.mtl* y *tareas.mtl* se encargan de generar el código que depende de un objetivo y una tarea respectivamente y por último los módulos *planificadorRuta.mtl*, *controladorBase.mtl* y *misión.mtl* generan, respectivamente, el código del nodo que representa el componente *path_planner*, el nodo que representa al componente *base_controller* y el código del nodo misión.

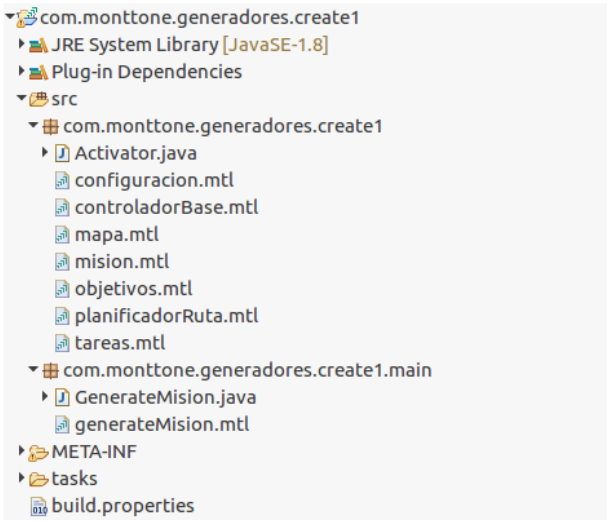

Figura 7.4: Estructura del proyecto Acceleo para la generación de código para un iCreate 1.

El módulo *generateMision.mtl* (figura 7.5) es el punto de entrada de la generación (comentario main, línea 22). Las líneas 13 a 19 importan los módulos mencionados anteriormente, cuyos templates públicos son invocados dentro del propio template *generateMision*. Este módulo, a trávez del método main de la clase *GenerateMision* es el que una aplicación externa debe invocar para la generación de código indicando como parámetro la ubicación del modelo XMI y del directorio a dónde se crearán los archivos.

```
12
13 [module generateMision('http://www.example.org/misiones')]
14 [import com::monttone::generadores::create1::configuracion/]
15 [import com::monttone::generadores::create1::mapa/]
16 [import com::monttone::generadores::create1::tareas/]
17 [import com::monttone::generadores::create1::planificadorRuta/]
18 [import com::monttone::generadores::create1::controladorBase/]
19 [import com::monttone::generadores::create1::mision/]
20
21 [template public generateMision(aMision : Mision)]
22 [comment @main/]
23
24 [manifest(aMision)/]
25 [makefile(aMision)/]
26 [launcher(aMision)/]
27
28 [mapa(aMision)/]
29 [generarTareas(aMision)/]
30
31 [nodoPlanificadorRuta(aMision)/]
32 [nodoControladorBase(aMision)/]
33 [nodoMision(aMision)/]
34
35 [/template]
36
```

Figura 7.5: Módulo principal de Acceleo para la generación del código para el Create 1.

En las próximas secciones se detalla cómo se genera cada componente de la solución, con capturas del código de Acceleo y una descripción de su funcionamiento.

7.2.1 El mapa

El módulo para generar el mapa es el mismo que fue presentando el capítulo anterior. La explicación completa puede ser consultada en la sección correspondiente del mencionado capítulo.

7.2.2 Las tareas

El módulo para generar las tareas es el mismo que fue presentando el capítulo anterior. La explicación completa puede ser consultada en la sección correspondiente del mencionado capítulo.

7.2.3 El nodo path_planner

Como se ha comentado anteriormente, se necesita de un nodo que se encargue de analizar el mapa y encontrar una posible ruta entre dos puntos de ese mapa. Este nodo recibe el nombre de planificador de ruta o *path_planner*. A continuación, no se explicará la totalidad del código del módulo *planificadorRuta.mtl*, sino las partes consideradas más importantes para nuestro objetivo. La razón es sencilla, puesto que se trata de un código dónde sólo algunas de sus líneas depende del modelo.

En primer lugar, el template *nodoPlanificadorRuta* genera un archivo C++ (bajo el directorio *src/* del *package*), con el nombre *path_planner.cpp*. La únicas línea que dependen del modelo se utiliza para referenciar al *package*. Por ejemplo, la línea 42 de la figura 7.6:

```
25
26 [template public nodoPlanificadorRuta(aMision : Mision)]
27 [definicionServicio(aMision)/]
28
29 [file (aMision.titulo.concat('/src/path_planner.cpp'), false, 'UTF-8')]
30 /**
31     path_planner.cpp
32
33     @author Mattone Nicolas - Montanari Franco
34     @version 1.0 20/05/2018
35 */
36
37 using namespace std;
38
39 #include <ros/ros.h>
40 #include <nav_msgs/GetMap.h>
41 #include <geometry_msgs/Pose.h>
42 #include <[aMision.titulo/]/get_path.h>
43 #include <vector>
44 #include "./dijkstra.cpp">
45
46 // Metadatos del mapa.
47 uint32_t width;
48 uint32_t height;
49 float resolution;
50 uint32_t origin_x;
51 uint32_t origin_y;
52 std::vector<int8_t> ocuppance_grid;
53 // Grafo que representa las posiciones libres del mapa.
54 Graph *grafo_libres;
```

Figura 7.6: Parte de la generación del código del componente path_planner.

El resto del código se encarga básicamente de: invocar al servicio *static_map* para obtener la información del mapa y luego, a partir de la matriz de ocupación, construir un grafo. Luego, se define un nuevo servicio llamado *get_path* (cuyo tipo será explicando a la brevedad). Para manejar una petición se asigna la función *service_callback*, que en pocas palabras, ejecuta el algoritmo de Dijkstra sobre el grafo construido y retorna la lista de puntos que representa el camino.

Hay que aclarar que la interfaz del servicio trabaja con puntos del mapa relativos al origen del mapa, pero internamente el programa transforma esos puntos a puntos de la matriz de ocupación (es decir, a pixels de la imagen).

Otro archivo a generar es el *get_path.srv*, bajo el directorio *srv/* del *package*. Estos archivos se utilizan para definir los datos a enviar en la petición y a retornar en la respuesta del servicio. Básicamente, se define los parámetros de la petición (*request*) y la respuesta (*response*), separados por tres guíones (línea 19). Para cada variable, se indica el tipo de dato y el nombre. El template privado *definicionServicio* (fig. 7.7) se encarga justamente de esto:

```
12
13 [template private definicionServicio(aMision : Mision)]
14 [file (aMision.titulo.concat('/srv/get_path.srv'), false, 'UTF-8')]
15 float64 x0
16 float64 y0
17 float64 x1
18 float64 y1
19 ---
20 int64 success
21 float64[']'/]] pathX
22 float64[']'/]] pathY
23 [/file]
24 [/template]
25
```

Figura 7.7: Template definicionServicio del módulo planificadorRuta.

Se observa que el servicio con tipo *get_path*, recibe como *request* cuatro valores que representa las posiciones (x,y) del punto origen (*x0*,*y0*) y del punto destino (*x1*,*y1*). Como response se

retorna si existe un camino (*success*) y dos vectores con las coordenadas en x (*pathX*) y en y (*pathY*) del camino resultado.

Por último, se genera un archivo llamado *dijkstra.cpp* bajo el directorio *src/* del *package*. Contiene una implementación del mencionado algoritmo y la definición de una clase que modela un grafo.

7.2.4 El nodo base_controller

Este nodo es el que sustituye al Stack de Navegación como punto de entrada. Nuevamente, no se explica el código completo del módulo *controladorBase.mtl*, sino sólo aquellas partes más relevantes y que dependen del modelo.

En primer lugar, el template *nodoControladorBase* genera un archivo C++ (bajo el directorio *src/* del *package*), con el nombre *base_controller.cpp*. La únicas línea que dependen del modelo se utiliza para referenciar al *package*. Por ejemplo, las líneas 36 y 37 de la figura 7.8. Las líneas 43 hasta 48 se encargar de configurar la velocidad del robot, de acuerdo a la resolución del mapa.

```
21
22 [template public nodoControladorBase(aMision : Mision)]
23 [definicionServicio(aMision)/]
24
25 [file (aMision.titulo.concat('/src/base_controller.cpp'), false, 'UTF-8')]
26 /**
27     base_controller.cpp
28
29     @author Mattone Nicolas - Montanari Franco
30     @version 1.0 20/05/2018
31 */
32
33 using namespace std;
34
35 #include <ros/ros.h>
36 #include <[aMision.titulo/]/get_path.h>
37 #include <[aMision.titulo/]/go_to.h>
38 #include <geometry_msgs/Twist.h>
39 #include <turtlesim/Pose.h>
40
41 const double PI = 3.14159265359;
42 const char CARDINALES['['/]] = "NESO";
43 const double DISTANCIA_PASO = [aMision.mapa.aspecto.resolucion/];
44 [if (aMision.mapa.aspecto.resolucion <= 0.1)]
45 const double VELOCIDAD = 0.1;
46 [else]
47 const double VELOCIDAD = [aMision.mapa.aspecto.resolucion/];
48 [/if]
49 // Configuración actual del robot.
50 double pos_actual_x;
51 double pos_actual_y;
52 uint64_t orientacion;
53 // Definición de clientes, subscriptores y publicadores.
54 ros::Publisher velocity_publisher;
55 ros::ServiceClient path_planner_client;
```

Figura 7.8: Parte de la generación del código del componente base_controller.

El nodo recibe como parámetro tres valores: la posición (eje x e y) y la orientación inicial (en radianes). Luego, se define el servicio *go_to* (cuyo tipo será explicando a la brevedad) y las suscripciones al servicio *get_path* y al canal *cmd_vel* para publicar los comandos de movimiento. Con todos estos valores, el nodo maneja la configuración actual del robot, es decir, la posición actual y la orientación actual y la actualiza de acuerdo a las respuestas del servicio *go_to*.

El código se encarga básicamente de manejar las peticiones de *go_to*. Lo que implica: la invocación a *get_path* (desde la posición actual hasta la solicitada por el usuario), el tratamiento del camino resultado recibido y su conversión a comandos de movimiento. Estos comandos están implementados en las funciones *rotar* y *avanzar_una_posicion*, y todo este manejo se encuentra la función *service_callback*.

Otro archivo a generar es el *go_to.srv*, bajo el directorio *srv/* del *package*. Define los datos a enviar en la petición y a retornar en la respuesta del servicio. El template privado *definicionServicio* se encarga justamente de esto:

```
12
13 [template private definicionServicio(aMision : Mision)]
14 [file (aMision.titulo.concat('/srv/go_to.srv'), false, 'UTF-8')]
15 float64 to_x
16 float64 to_y
17 ---
18 int64 success
19 [/file]
20 [/template]
21
```

Figura 7.9: Template definicionServicio del módulo controladorBase.

Se observa que el servicio con tipo *go_to*, recibe como *request* dos valores que representan la posiciones (*x*,*y*) del punto destino (*x1*,*y1*). Cómo response se retorna si ese punto fue alcanzado (*success*). Se puede apreciar que la interfaz del servicio cumple con la planteada en el capítulo de la definición del DSL.

7.2.5 El nodo misión

El módulo *mision.mtl* genera el nuevo nodo de la topología, que ejecuta la **Misión**. Para ello, se precisa definir el nodo mediante el archivo *mision.cpp* ubicado bajo el directorio *src/*. De esto se encarga el template *nodoMision* (línea 14, fig. 7.10), que recibe como parámetro una entidad Mision.

```
12 [import com::monttone::generadores::create1::objetivos/]
13
14 [template public nodoMision(aMision : Mision)]
15 [file (aMision.titulo.concat('/src/mision.cpp'), false, 'UTF-8')]
16 [declaraciones(aMision)/]
17
18 int main( int argc, char** argv ) {
19     ros::init(argc, argv, "mision");
20     ros::NodeHandle n;
21     ros::spinOnce();
22
23     std_srvs::Trigger trigger_req;
24     [for (tarea : Tarea | aMision.tareasDisponibles )]
25     [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
26     ros::ServiceClient [tarea.nombre/]_client = n.serviceClient<std_srvs::Trigger>("[tarea.nombre/]_srv");
27     [/if]
28     [/for]
29
30     [for (tarea : Tarea | aMision.tareasDisponibles )]
31     [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
32     while(![tarea.nombre/]_client.waitForExistence(ros::Duration(5.0))){};
33     [/if]
34     [/for]
35
36     base_controller_client = n.serviceClient<[aMision.titulo/]::go_to>("go_to");
37     while(!base_controller_client.waitForExistence(ros::Duration(5.0))){ }
38
39     for( int x = 0; x < [aMision.cantidadRepeticiones/]; x++ ) {
40     [for (objetivoActual : Objetivo | aMision.objetivos)]
41         [procesarObjetivo(objetivoActual, aMision)/]
42     [/for]
43     }
44
45     ROS_INFO("La mision '[aMision.titulo/]' ha sido ejecutada correctamente.");
46     ros::spin();
47     return 0;
48 }
49
50 [moveToGoal(aMision)/]
51 [/file]
52 [/template]
```

Figura 7.10: Módulo de generación del nodo misión (Create 1).

En primer lugar, se incluyen las referencias necesarias para utilizar el servicio *go_to* (línea 16), para invocar las tareas (el tipo de servicio Trigger) y las definiciones de las variables y funciones. Esto se realiza a trávez del template *declaraciones*, cuyo código se muestra en la figura 7.11:

```
65
66 [template private declaraciones(aMision : Mision)]
67 using namespace std;
68
69 #include <ros/ros.h>
70 #include <[aMision.titulo/]/go_to.h>
71 #include <std_srvs/Trigger.h>
72
73 // Definición de clientes, subscriptores y publicadores.
74 ros::ServiceClient base_controller_client;
75
76 /**
77     Invoca al nodo base_controller para mover al robot
78     hasta la ubicación (x,y) del mapa.
79
80     @param x, coordenada en x de la ubicación.
81     @param y, coordenada en y de la ubicación.
82     @return true/false dependiendo de si el robot llegó o no
83 */
84 bool ir_a_ubicacion(double x, double y);
85 [/template]
86
```

Figura 7.11: Template para las declaraciones y definiciones del archivo C++ (Create 1).

Lo que sigue es similar al ejemplo del capítulo anterior: por cada **TareaPredefinida** de la lista de *tareasDisponibles* de la Misión, se define un cliente para cada uno de los servicios ofrecidos por los nodos que representan la tarea (línea 26, fig. 7.10). Y para sincronizar el nodo misión con el resto de los nodos tarea (es decir, que el primero espere hasta que todos los demás estén disponibles) se ejecuta la sentencia del framework *waitForExistence* (línea 32, fig. 7.10).

Una vez que las tareas se encuentran disponibles para ser ejecutadas, el nodo se subscribe al servicio *go_to* y espera hasta que este se encuentre disponible (línea 37, fig. 7.10). Entonces se realiza una iteración cuyo número está definido por el atributo *cantidadRepeticiones* de la Misión (línea 39, fig. 7.10). Dentro de este loop se recorre la lista de *objetivos* de la Misión y para cada uno se invoca al template *procesarObjetivo* (línea 41, fig. 7.10) del módulo *objetivos.mtl* (fig. 7.12), que recibe como entrada una entidad Objetivo:

```
22
23 [template public procesarObjetivo(aObjetivo : Objetivo, aMision : Mision)]
24 /*
25     [aObjetivo.comentario/]
26 */
27
28 if( ir_a_ubicacion([aObjetivo.ubicacion.x/], [aObjetivo.ubicacion.y/]) ) {
29     ROS_INFO("El robot ha llegado al punto ([aObjetivo.ubicacion.x/], [aObjetivo.ubicacion.y/])");
30     [procesarTareasObjetivo(aObjetivo)/]
31 } else {
32     ROS_INFO("El robot NO ha alcanzado el objetivo ([aObjetivo.ubicacion.x/], [aObjetivo.ubicacion.y/])");
33     [if (aObjetivo.obligatorio)]
34     ROS_INFO("El objetivo era obligatorio. Se cancela la mision");
35     return 1;
36     [/if]
37 }
38 [/template]
```

Figura 7.12: Template para el tratamiento de un objetivo (Create 1).

En el mencionado módulo, primero se incrusta el comentario del **Objetivo** (línea 25). Luego, se ejecuta la función *ir_a_ubicacion* (línea 28), que recibe como parámetro la posición en x y en y del objetivo y se encarga de invocar al servicio *go_to* del nodo *controlador_base*.

```
86
87 [template private moveToGoal(aMision : Mision)]
88 // Invoca al nodo base_controller para mover al robot
89 // hasta la ubicación (x,y) del mapa.
90 bool ir_a_ubicacion(double x, double y) {
91     [aMision.titulo/]::go_to srv;
92     srv.request.to_x = x;
93     srv.request.to_y = y;
94
95     if(base_controller_client.call(srv)) {
96         return srv.response.success;
97     } else {
98         ROS_ERROR("No se pudo establecer la comunicacion con el nodo 'base_controller'.");
99         return false;
100    }
101 }
102 [/template]
```

Figura 7.13: Cuerpo de la función moveToGoal (Create 1).

Si el robot pudo alcanzar el punto determinado del mapa, la función retorna true y en caso contrario, false. Al igual que en el capítulo anterior, si ocurre lo segundo y si el atributo *obligatorio* del Objetivo es verdadero (línea 33), se cancela la misión (con la sentencia *return*). Ahora, si el resultado de la función es verdadero, se genera el código para ejecutar las **Tareas** de ese Objetivo (línea 30). El mismo es el que fue presentado (y explicado) en el capítulo anterior.

7.2.6 Otros archivos

Para completar la estructura del *package*, es necesario generar más archivos mediante el módulo *configuración.mtl* El template *manifest* genera el archivo *manifest.xml*, que contiene los metadatos (información, dependencias, etc). El código es el mismo que fue presentado en el capítulo anterior.

Luego, el template *makefile* genera otros dos archivos necesarios para la compilación: Makefile (directivas comunes) y CMakeList (directivas específicas de ROS). También reciben como entrada una entidad Mision.

```
27
28 [template public makefile(aMision : Mision)]
29 [file (aMision.titulo.concat('/Makefile'), false, 'UTF-8')]
30 include $(shell rospack find mk)/cmake.mk
31 [/file]
32
33 [file (aMision.titulo.concat('/CMakeLists.txt'), false, 'UTF-8')]
34 cmake_minimum_required(VERSION 2.4.6)
35 include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
36
37 # Set the build type.  Options are:
38 #  Coverage       : w/ debug symbols, w/o optimization, w/ code-coverage
39 #  Debug          : w/ debug symbols, w/o optimization
40 #  Release        : w/o debug symbols, w/ optimization
41 #  RelWithDebInfo : w/ debug symbols, w/ optimization
42 #  MinSizeRel     : w/o debug symbols, w/ optimization, stripped binaries
43 #set(ROS_BUILD_TYPE RelWithDebInfo)
44
45 rosbuild_init()
46
47 #set the default path for built executables to the "bin" directory
48 set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
49 #set the default path for built libraries to the "lib" directory
50 set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
51
52 #uncomment if you have defined messages
53 #rosbuild_genmsg()
54 #uncomment if you have defined services
55 rosbuild_gensrv()
56
57 rosbuild_add_executable([aMision.titulo/] src/mision.cpp)
58 [for (tarea : Tarea | aMision.tareasDisponibles)]
59 [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
60 rosbuild_add_executable([tarea.nombre/]_srv src/[tarea.nombre/]_srv.cpp)
61 [/if]
62 [/for]
63 rosbuild_add_executable(path_planner src/path_planner.cpp)
64 rosbuild_add_executable(base_controller src/base_controller.cpp)
65 [/file]
66 [/template]
67
```

Figura 7.14: Template de las directivas de compilación (Create 1).

Del archivo CMakeLists.txt, se aprecian las directivas para compilar los nodos que van a formar parte de la topología (*rosbuild_add_executable*), indicando el nombre y el archivo fuente. En nuestro caso, el nodo *misión*, *path_planer*, *base_controller* y los nodos que representan una **TareaPredefinida**. Además, al trabajar con tipos de servicios definidos por el usuario, se descomenta la línea *rosbuild_gensrv()* tal como dicta la documentación del framework.

Por último, se genera un archivo lanzador mediante el template *launcher* 7.15. La línea 79 se encarga de ejecutar el nodo *map_server* de ROS, indicando como parámetro la ubicación del archivo YAML que modela el mapa.

```
75
76 [template public launcher(aMision : Mision)]
77 [file (aMision.titulo.concat('/launch/iniciar_mision.launch'), false, 'UTF-8')]
78 <launch>
79     <node name="map_server" pkg="map_server" type="map_server"
80         args="$(find [aMision.titulo/])/maps/mapa.yaml"/>
81
82 [for (tarea : Tarea | aMision.tareasDisponibles)]
83     [if (not(tarea.nombre.equalsIgnoreCase('dormir')))]
84     <node name="[tarea.nombre/]_srv" pkg="[aMision.titulo/]" type="[tarea.nombre/]_srv"/>
85     [/if]
86 [/for]
87
88     <node name="path_planner" pkg="[aMision.titulo/]" type="path_planner"/>
89     <node name="base_controller" pkg="[aMision.titulo/]" type="base_controller"
90         args="[aMision.robot.ubicacionInicial.x/] [aMision.robot.ubicacionInicial.y/]
91               [aMision.robot.orientacionInicial/]"/>
92     <node name="[aMision.titulo/]" pkg="[aMision.titulo/]" type="[aMision.titulo/]"/>
93 </launch>
94 [/file]
   [/template]
```

Figura 7.15: Template del archivo lanzador (Create 1).

La línea 82 se encarga de ejecutar cada nodo con la TareaPredefinda. Luego, las líneas 88, 89 y 92 ejecutan, respectivamente, al nodo *path_planner*, al nodo *base_controller* (indicando como parámetros la *posición* y *orientación* inicial del **Robot**) y finalmente al nodo *misión*.

7.3 Ejecutando la misión

El programa escrito en Acceleo genera un package de ROS. Este posee un archivo launcher que se encarga de:

- Ejecutar los nodos que representan las tareas y que ofrecen un servicio que puede ser invocado por cualquier nodo de la topología.
- Ejecutar los nodos que representan el componente de navegación: *path_planner* y *controlador_base*.
- Ejecutar el nodo que administra el mapa: *map_server*.
- Ejecutar el nodo *misión*.

Este nodo *misión*, se encarga de mover al robot a ciertas posiciones del mapa (mediante el componente de navegación) y ejecutar lo servicios de los nodos correspondientes a esa posición del mapa. Este circuito se repite una cantidad determinada de veces.

Entonces, para ejecutar la misión, primero se debe completar el código de las TareasPredefindas (es decir, los archivos del directorio *src/* terminados en *'_srv'*). Luego se ejecuta el siguiente comando (ubicado en el directorio raíz del *package*):

```
$ rosmake
```

Una vez que la compilación ha terminado correctamente, se ejecuta el archivo lanzador de la siguiente manera. Supongamos que el título de la Misión es *'mi_primera_mision'*. Vale aclarar que previamente debe estar conectado el Create 1 a la computadora, con el driver inicializado.

```
$ roscore
$ roslaunch mi_primera_mision iniciar_mision
```

De esta forma, se ejecutan todos los nodos mencionados anteriormente y comienza efectivamente la misión.

Como adjunto a este tesina, se presenta un video de una misión específicada con el *DSL para robots con misión predeterminada* (a trávez de la herramienta desarrollada para esta tesina, llamada *Editor de Misiones*) y ejecutada por un robot iRobot Create 1. La figura 7.16 compara el archivo de imagen que representa el mapa con el entorno real.

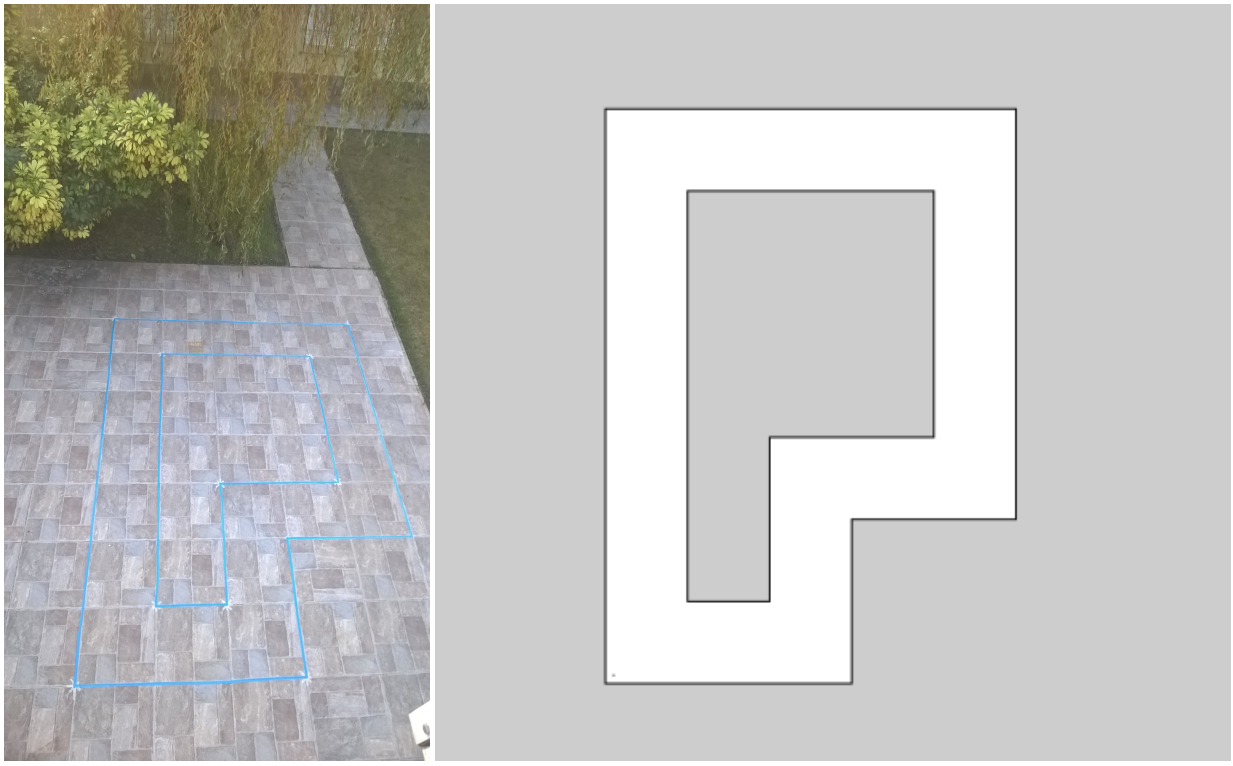

Figura 7.16: Entorno real de la misión de demostración para esta tesina (a la izquierda) y su representación en imagen (a la derecha). Se pueder ver identifcado el circuito mediante cintas de color celeste que representan paredes, y en el mapa con píxeles en negro.

Capítulo 8

Conclusiones y trabajos futuros

8.1 Conclusiones

En esta tesina se expone la potencia de MDD, introduciendo el concepto desde lo más básico para luego, dado un problema particular, solucionarlo y demostrar el alcance de este paradigma con ejemplos prácticos.

El problema a atacar pertenece al dominio de la robótica, lo que requirió un estudio intensivo de la misma ya que ninguno de los integrantes de este trabajo contaban con experiencia en ese ámbito ni en MDD. Pero, gracias a los avances y a la cantidad de artículos y trabajos enfocados en aplicar MDD al domino de la robótica, se pudo resolver el problema de forma práctica.

La alternativa Eclipse ha demostrado ser muy efectiva, tanto para la definición del metamodelo del DSL (a trávez del plug-in EMF), el desarrollo de la herramienta (Swing) y la generación de código (mediante Acceleo). Esto se logra gracias al enfoque sencillo y práctico de la plataforma para manejar la definición de lenguajes, la generación automática de código a partir de modelos y el desarrollo de una interfaz gráfica.

Además, se debe reconocer que el framework ROS es una excelente opción como plataforma específica para programar sistemas robóticos. Aplica todos los conceptos que facilitan la programación de robots (como la separación de componentes, reutilización de código, abstracción de hardware, etc) y permite a los programadores trabajar fácil y rapidamente. Pero, el aspecto que quieren destacar los autores de esta tesina, es que el framework organiza en un sólo lugar conceptos y resoluciones de muchos problemas generales de la robótica (como la interacción entre componentes y la navegación de un robot). De esta forma, no se está re-inventado la rueda constantemente y se agrupa el conocimiento del dominio en un área en un proyecto respaldado por la comunidad (entre las que se incluyen universidades y laboratorios con mucho prestigio).

Entonces, gracias a los avances en el campo (MDD y robótica) y a las herramientas disponibles en el mercado (EMF, Acceleo y ROS), se logró definir el *DSL para robots con misión predeterminada* y además acompañarlo con una herramienta gráfica que permite a los usuarios definir recorridos en un mapa e indicarle al robot que ejecute acciones sobre ciertos puntos del recorrido, además de ofrecerle la posibilidad al usuario de definir (y programar) sus propias acciones. Para demostrar el uso y el alcance del DSL, se ha utilizado una simulación de un robot popular en el mercado actual (Turtlebot 2) y un robot real (iRobot Create 1), con resultados éxitos (que pueden ser mejorados en trabajos futuros).

Por lo mencionado anteriormente, se llega a la conclusión de que, no sólo se ha resuelto el problema planteado, sino que MDD ha demostrado ser un paradigma que puede ser aplicado y utilizado de forma práctica, con resultados reales y concretos. El trabajo de esta tesina sirve para demostrar esta afirmación.

8.2 Trabajos futuros

En base al trabajo realizado a lo largo de la tesina y a los resultados obtenidos, se proponen las siguientes líneas como trabajos futuros:

1. **Mejorar la herramienta gráfica *Editor de Misiones***: En concreto, se propone migrar la aplicación desarrollada en Java (Swing) a un sistema web, similar a FLYAQ [4]. De esta forma, aprovechando las ventajas de HTML y Javascript, se lograría una interfaz flexible, intuitiva y por ende más accesible para cualquier tipo de usuario al momento de especificar las misiones.

2. **Extender el metamodelo del *DSL para robots con misión predeterminada***: Se pretende identificar más acciones comunes a cualquier sistema robótico, del estilo de la tarea *dormir*, es decir, subclases directas de la entidad Tarea. Se recuerda que el usuario no debe completar el código de estas tareas, sino que la transformación queda a cargo de la plataforma específica. Entonces, se puede investigar cuáles son aquellas funcionalidades que gran parte de los sistemá robóticos tengan incorporados para luego agregarlas al metamodelo.

3. **Implementar una generación de código para otras plataformas específicas diferentes a ROS**. En este informe se ha destacado las ventajas de dicho framework, pero no se ha trabajado con otras alternativas. Se propone ver cómo se generaría el código, a partir del DSL presentado en esta tesina, para diferentes plataformas específicas.

4. **Restringir el metamodelo del *DSL para robots con misión predeterminada* a un dominio similar al planteado en esta tesina, pero más específico**: La principal limitación de este DSL es que obliga a los usuarios casuales (es decir, sin ningún tipo de conocimiento informático) a programar la funcionalidad de aquellas tareas indicadas como personalizadas. Entonces, se propone tomar un dominio incluido en el problema de esta tesina y específicar un metamodelo, que tome como base el metamodelo del capítulo 5, y resuelva el problema sin el uso de la entidad TareaPersonalizada. Para completar esto, se puede desarrollar una herramienta gráfica. De esta forma, el código generado no requerirá de ningún tipo de participación del usuario y además se demostraría la reusabilidad del *DSL para robots con misión predeterminada.*

Apéndice A

Notas sobre el metamodelo y ROS

En la primera sección de este anexo se comenta el por qué de la representación elegida para ciertas entidades del modelo del DSL presentado en esta tesina. Luego, en la segunda sección, se describe una funcionalidad muy utilizada del framework ROS: el Stack de navegación.

A.1 Justificación técnica del modelo

En esta sección se explica en un mayor nivel de detalle algunas entidades del modelo. Esta explicación debe verse como una justificación de cada elemento a un nivel más técnico, de acuerdo a los conceptos y soluciones que existen actualmente en el mercado.

A.1.1 El mapa

Para nuestro modelo, el mundo del robot está representado por una imagen y una serie de atributos para enriquecer la información. Una imagen no es más que una **matriz** de pixels (dos dimensiones), cuya dimensión viene dada por el alto y el largo de la mísma. Cada píxel se corresponde con un elemento de la matriz y tiene un valor que identifica al color.

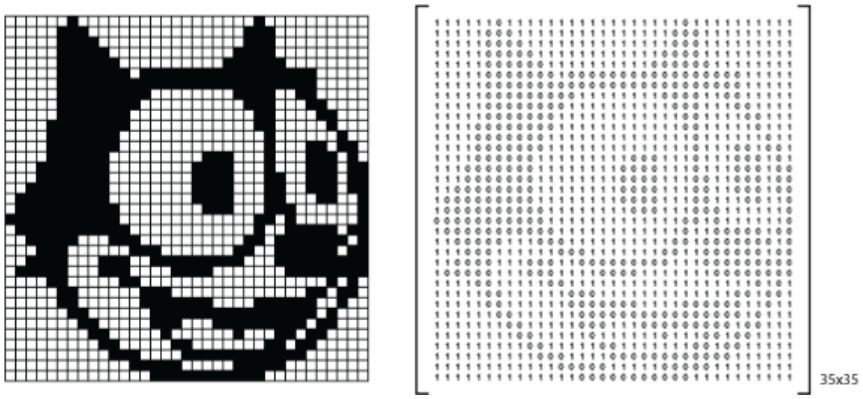

Figura A.1: Relación entre una imagen y su representación como matriz. A la izquierda se muestra una imagen de 35x35, y a la derecha una matriz que la representa. El alto y el largo de la imagen definen las dimensiones de la matriz. Los píxeles se corresponden a posiciones de la matriz y los colores a valores numéricos. En este ejemplo, la imagen es de 35x35.

El mapa del modelo tiene como unidad, el **metro**, pero la unidad de una imagen es un píxel. **El atributo resolución indica justamente la relación que hay entre metros y pixels**. Por ejemplo, si tenemos una imagen de 960 x 480 pixels y se indica un resolución de 0.05, esto significa que el mapa representa un mundo de 48 x 24 metros.

Ahora, un elemento p de la matriz $imagen$, dónde $p = imagen(p_x, p_y)$ representa el punto $(p_x * resolucion, p_y * resolucion)$ del mapa. Por ejemplo, si tenemos una imagen de 4000x4000 y una resolución de 0.05, el pixel $imagen(2000, 2000)$ equivale al punto $(100, 100)$ del mapa.

Pero de esta forma, el origen del mapa siempre sería el primer pixel de la imagen. Por lo tanto, se define un origen de coordenadas para el mapa, es decir, un elemento $p_0 = imagen(p0_x, p0_y)$ que representa el punto (0,0). Entonces, un elemento $p = imagen(p_x, p_y)$ representa el punto $((p_x - p0_x) * resolucion, (py_0 - p0_y) * resolucion)$ del mapa. **Se calcula un valor relativo al origen de coordenadas** y de esta forma, se construye un plano cartesiano a partir de una imagen.

Si bien la imagen puede ser a color, para el procesamiento del mapa sólo se tienen en cuenta los píxeles de color negro y blanco. ¿Por qué? **Para identificar si un punto determinado del mapa se encuentra ocupado o no, dependiendo del color del pixel**: Un pixel en blanco representa un lugar libre, mientras que uno en negro indica que hay un obstáculo en ese punto. Los obstáculos que se identifican son estáticos. El mapa debe ser visto, no sólo como sistema de referencia y localización, sin también como una **matriz de ocupación**, una estructura de datos muy utilizada en los sistemas de navegación autónoma de un robot.

Por último, el atributo **invertido** de la entidad Mapa indica si se debe tomar un pixel en blanco como obstáculo y uno negro como libre (en el caso de ser verdadero) o viceversa (en caso de ser falso).

Esta representación de un mapa que se propone, se encuentra basada en la generación de mapas por medio de robots. En ella, básicamente, un operario controla el movimiento del robot a través de un entorno cerrado y a partir de la lectura de los datos de ciertos sensores (láser, colisiones) y una serie de algoritmos, se obtiene una matriz que representa ese entorno. Esta matriz puede ser fácilmente convertida en una imagen.

Por ejemplo, la figura A.2 muestra una imagen generada por el framework de ROS. No es una imagen completamente en blanco y negro, sino que permite diferentes colores. Pero, al momento de identificar si una posición está libre o no, se aplica una función de probabilidad (basada en el canal RGB del píxel) que asigna un valor a esa posición. Entonces, si ese valor se encuentra en un rango determinado (definido en los atributos del mapa), la posición es considerada como libre, ocupado o desconocida. El formato de los mapas de ROS será explicado en las siguientes secciones. Hay que aclarar que una posición desconocida indica que el algoritmo no pudo determinar si existe o no un obstáculo.

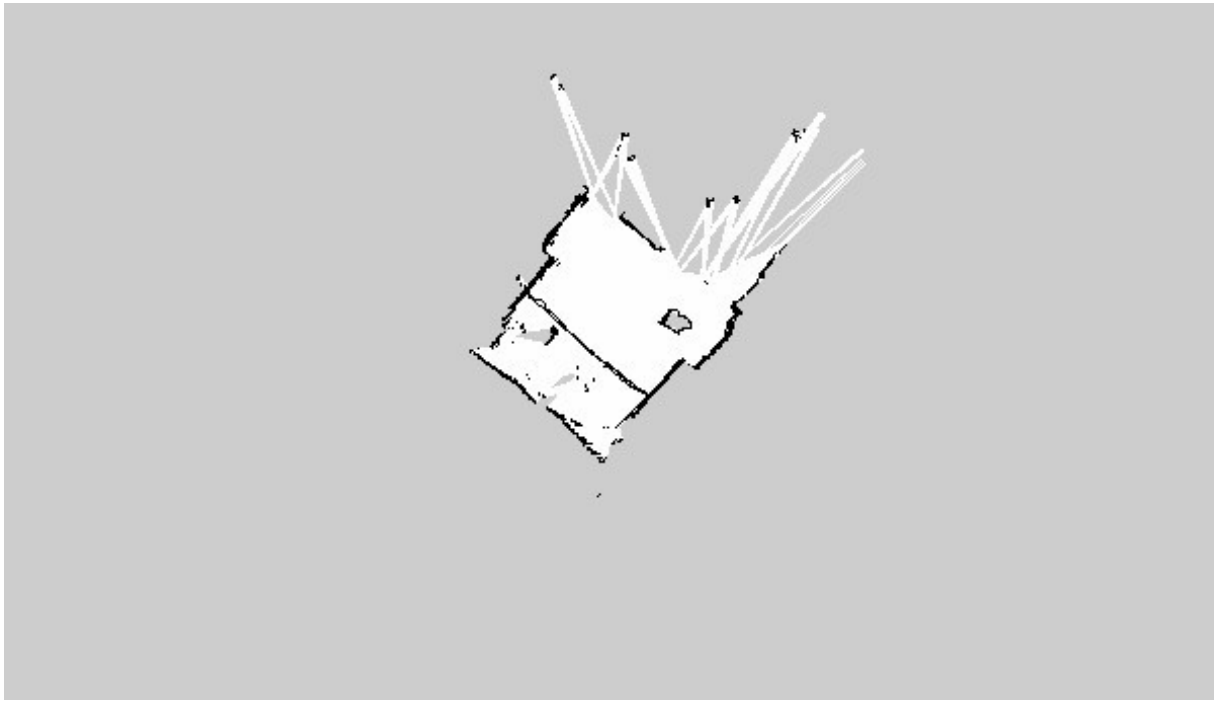

Figura A.2: Mapa por un robot a trávez del framework ROS.

Otro ejemplo es *2D Robot Mapping Software for autonomous navigation* [24], una herramienta que si bien obtiene una matriz de ocupación, el resultado final no es un archivo de imagen (sino un archivo .csv). Pero, la representación visual del mapa es similar a la planteada en este capítulo.

Básicamente cada punto tiene un valor de probabilidad y es representado en la interfaz gráfica con un color determinado, de la siguiente manera: Verde oscuro para los obstáculos (probabilidad entre [0.8-1]), verde claro para desconocido (probabilidad entre [0.51-0.79]), blanco para libre ([0-0.49]) y gris para los puntos sin explorar (probabilidad igual a 0.5).

Además, la herramienta define la resolución de un mapa en centímetros por pixel y permite especificar el origen de coordenadas. La figura 4-4 una captura de la herramienta.

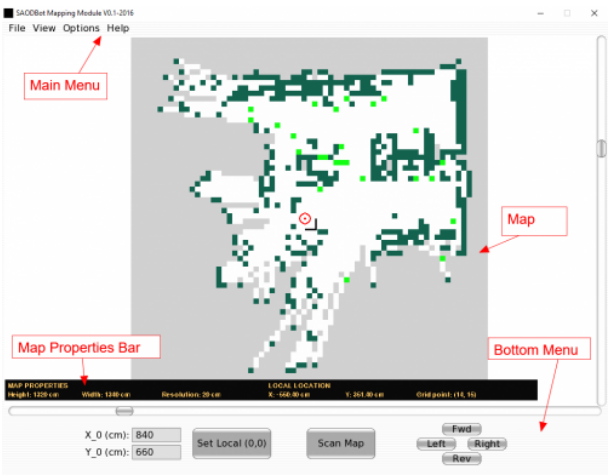

Figura A.3: Captura de la herramienta *2D Robot Mapping Software for autonomous navigation* [24].

A.1.2 El robot

Para poder relacionar el mundo real con el robot, en nuestro modelo se solicitan tanto la **posición** inicial del mismo en el mapa y su **orientación**. Si bien algoritmos de localización como Monte Carlo (MCL) [8] pueden estimar la posición inicial en un mapa conocido, esta operación en mapas muy grandes puede llegar a tardar mucho tiempo [9]. Para solucionar este problema, estos algoritmos permiten como parámetro de entrada opcional la posición inicial del robot. De hecho, en el lenguaje definido en el paper *Automatic Generation of detailed Flight Plans from High-Level Mission Descriptions* [4] se asume que al principio de la misión cada dron se encuentra estacionado en una ubicación de origen. Por esta razones y porque de alguna forma es natural conocer desde dónde empieza un recorrido, se definen estos atributos en la entidad Robot.

Además, el modelo deja de lado los componentes físicos del robot: medidas (largo, ancho y alto), peso, tipo de cuerpo (cuadrado, circular), cantidad de ruedas, movimiento diferencial y muchas otras cuestiones. **Este tipo de información muy específica queda separado y se completa al momento de generar el código**.

Pero, obligatoriamente, el robot debe cumplir con la siguientes operaciones en su interfaz:

- **goTo**: El robot se mueve desde la posición actual hasta una posición del mapa recibida como parámetro. Durante este trayecto, el robot evade cualquier tipo de obstáculo, ya sea estático (identificado previamente en el mapa) o dinámico (objeto no identificado en el mapa, que convierte una posición en obstáculo). Luego de ejecutar esta operación, se retorna si se pudo llegar a la posición destino.
- **dormir**: El robot espera (no se mueve ni ejecuta ninguna acción) durante una cantidad de segundos recibida como parámetro. Similar a la función sleep de un proceso del sistema operativo.
- **ejecutarTarea**: El robot ejecuta una tarea, cuya identificación recibe como parámetro. Una vez terminada la ejecución de la tarea, sin importar el resultado, el robot queda posicionado en el mismo punto del mapa previo a la ejecución.

Estas operaciones no tienen que ser exactamente iguales a como fueron presentadas. Por ejemplo, ejecutarTarea podría ser un método llamado tomarFotografía() o incluso un llamado a un servicio de algún nodo del sistema que se encarga de comunicarse con la cámara y obtener la imagen.

Por supuesto, para lograr esto, el sistema robótico debería estar formado por otras componentes. Éstas deberían encargarse, entre otras actividades de:

- Administrar el mapa y mantener la ubicación actual del robot.
- Planificar una ruta, para evitar los obstáculos estáticos (identificados en el mapa).
- Identificar y esquivar los obstáculos dinámicos (utilizando diferentes sensores).
- Control del movimiento del robot (por ejemplo, la velocidad y orientación del motor de una rueda).

Una simple implementación de goTo podría ser: obtener el mapa (por ejemplo, convertir la imagen en una matriz de enteros), planificar la ruta desde la posición actual del robot hasta el objetivo (mediante el algoritmo de Dijkstra) y mover al robot una determinada distancia (esquivando obstáculos dinámicos con el sensor láser o los bumpers del robot).

Pero en definitiva, con las operaciones nombradas, sería suficiente para trabajar con el DSL propuesto.

A.2 El Stack de Navegación de ROS

El Stack de Navegación de ROS [18] es un conjunto de algoritmos que utilizan la información de los sensores y odometría del robot, permitiendo controlarlo mediante el envío de un mensaje estándar hacia un nodo específico de la topología (llamado *controlador base*). De esta forma, se lo puede mover de un una posición a otra, evitando que el robot se atasque, choque o se pierda.

Esta librería puede ser utilizada fácilmente con cualquier robot que sea compatible con el framework de ROS. Pero requiere que el usuario describa ciertos archivos de configuración y algunos nodos. Específicamente, el sistema robótico debe satisfacer los siguientes requerimientos:

- Poseer un sensor láser. Se utiliza para crear el mapa del entorno y para la localización (por ejemplo, evitar obstáculos). El robot debe publicar esta información para que pueda ser consumida por el sistema.
- La forma del robot debe ser cuadrada o circular.
- El movimiento de las ruedas del robot debe ser diferencial u holonómico.
- El robot debe publicar información sobre las relaciones entre sus diferentes componentes físicos (llamados "frames").
- El robot debe publicar su odometría, para ser consumida por el sistema.

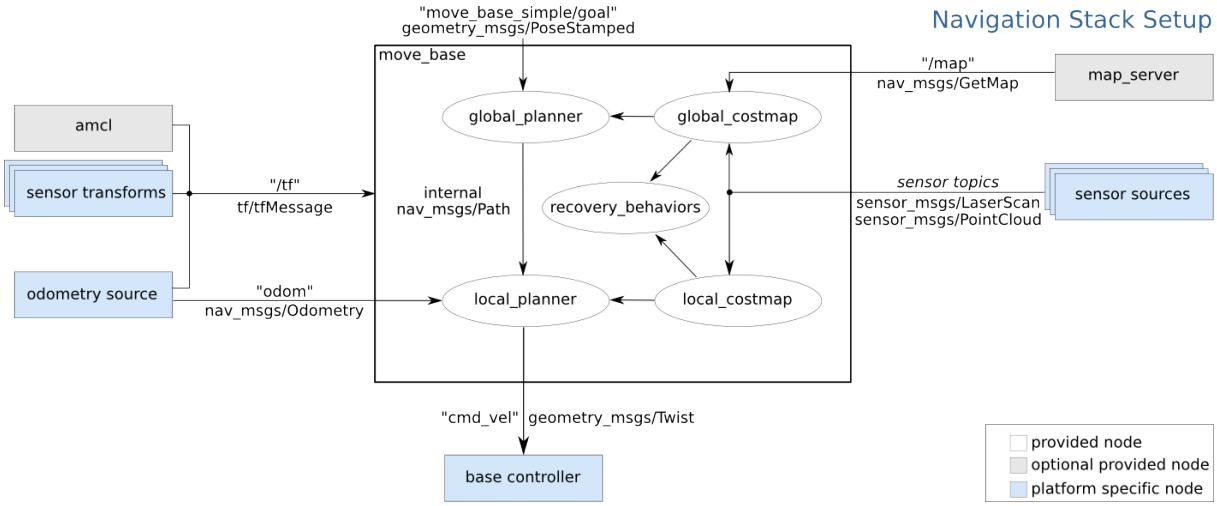

Figura A.4: Diagrama de la documentación oficial del framework [18], con los componentes que forman el Stack de Navegación. En color blanco se identifica el framework en sí, en celeste los nodos opcionales y en azul los componentes específicos que debe completar el usuario.

La figura A.5 muestra cómo se encuentra organizado el Stack de Navegación. Se aprecian tres grandes grupos, identificados por diferentes colores: en blanco se marcan los componentes provistos por el framework (con la lógica para el movimiento autónomo del robot, como por ejemplo, la planificación de ruta), en celeste los nodos opcionales (también provistos por ROS) y en azul los nodos que el usuario debe implementar (específicos de la plataforma).

Toda la documentación del framework puede ser consultada en el sitio oficial [18]. En ella se explica cómo implementar cada requisito. A continuación, en forma resumida, se explican algunos conceptos asociados a cada requisito, extraídos y traducidos de la mencionada documentación:

TF: Árbol de transformación

Un robot está formado por distintas componentes: un cuerpo, ruedas, brazos, sensores, etc. Si seleccionamos un componente como centro del robot, podemos armar un árbol dónde ese centro

corresponde a la raíz. En un nivel abstracto, un árbol de transformación TF (por su traducción en inglés, transform tree) define las relaciones en cuanto a traslación y rotación entre los diferentes nodos del árbol (teniendo como referencia el nodo raíz).

Consideremos el ejemplo de un robot que tiene una base movible con un láser montado sobre la misma. Podemos definir dos coordenadas (o *frames*, como los llama el framework ROS), una correspondiente al centro de la base (*base_link*) y otra al centro del sensor láser (*base_laser*). Ahora supongamos que el sensor lee cierta información, que queremos utilizar para mover al robot cierta distancia. Pero para poder hacerlo, se necesita alguna forma de transformar la información de un frame a otro, es decir, definir la relación que hay entre uno y otro.

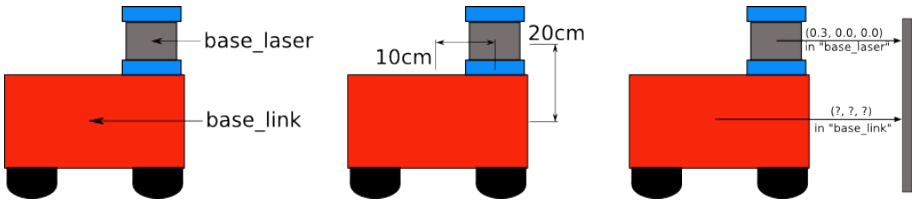

Figura A.5: Relaciones entre los componentes base_link y base_laser de un robot.

Supongamos que el láser está montado 10 cm hacia delante y 20 cm arriba de la base, como muestra la figura 5-2. Entonces, para transformar la información de un frame a otro, se puede aplicar la relación (x: 0.1m, y: 0.0m, z: 0.2m) para base-láser y (x: -0.1m, y: 0.0m, z: -0.20m) para láser-base.

El Stack de Navegación necesita que el usuario implemente un nodo que publique las relaciones entre los frames del robot.

La odometría

La odometría se utiliza para estimar la posición relativa de un robot móvil. Se habla de una estimación de posición ya que saber exactamente la posición de un robot es imposible, porque los métodos utilizados para calcular esa posición no cuentan con una precisión absoluta. Un ejemplo simple es calcular la odometría de acuerdo a la corriente enviada al motor de la rueda: si no se tiene en cuenta el "bloqueo" del robot por parte de una pared, este cálculo de la odometría indicaría que el robot está en una posición x cuando en realidad está en una posición anterior (de la que nunca pudo salir por el bloque).

Las posiciones estimadas a partir de la odometría no tienen nada que ver con las coordenadas del entorno (mapa o mundo): son relativas al punto de inicio. Definimos como punto de inicio aquel punto en el que se comienza a calcular la posición mediante odometría. Es decir, aquel punto en el que encendemos el robot y comienza la ejecución del programa que estima la posición. Este punto de inicio puede estar en cualquier punto de nuestro mapa.

El Stack de Navegación necesita que el usuario implemente un nodo que publique la odometría del robot.

El nodo base_controller

El Stack de Navegación asume que se puede enviar comandos de movimiento al robot, mediante el envío de mensajes de tipo *geometry_msgs/Twist* al canal *cmd_vel*.

Entonces, debe existir un nodo suscripto al canal *cmd_vel* que es capaz de tomar el mensaje y convertirlo a una directiva más específica que mueva la base del robot (es decir, el pulso eléctrico que gire la rueda a una determinada velocidad). El usuario es el encargado de proveer este nodo a la topología.

El mapa

El Stack de Navegación no precisa obligatoriamente de un mapa para operar, pero para el caso de esta tesina, es necesario contar con un mapa conocido.

Para ROS, un mapa está formado por un par de archivos: un archivo YAML que describe los metadatos y el nombre de la imagen. El archivo de la imagen representa la matriz de ocupación, es decir, qué posiciones del mundo están libres y cuáles están ocupadas. Los píxeles en blanco indican posiciones libres, los negros ocupadas y los intermedios son desconocidos. Para estos pixels desconocidos, se utiliza una función (llamada *probabilidad de ocupación*) para decidir si estos son libres u ocupados. Cuando se comunica la información del mapa a la topología de ROS, la ocupación se representa por un entero (en el rango [0,100]) donde 0 significa que esa posición está completamente libre y 100 que está completamente ocupada. El valor -1 se refiere a una posición desconocida.

La *probabilidad de ocupación* se calcula de la siguiente manera:

```
occ = (255 - color_avg) / 255.0
```

Dónde *color_avg* es el promedio de todos los canales de ese pixel. Por ejemplo, si el color de la imagen está representado por 24 bits, un píxel con el color *0x0a0a0a* tiene una probabilidad de 0.96. Mientras que otro con el color *0xeeeeee* tiene 0.07. El framework soporta la mayoría de los formatos de imagen. Pero por ejemplo, el formato PNG no es soportado en OS X.

La estructura del archivo YAML es la siguiente:

```
image: testmap.png
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

- *image*: ruta (absoluta o relativa) con la ubicación de la imagen en el sistema de archivos.
- *resolution*: resolución del mapa (metros / pixel).
- *origin*: la posición (x,y, yaw) del píxel inferior izquierdo del mapa (el origen de coordenadas del mapa), dónde yaw representa la rotación (en sentido de las agujas del reloj). 0 significa que no hay rotación.
- *occupied_thresh*: determina el valor mínimo para que un pixel sea considerado como ocupado.
- *free_thresh*: determina el valor máximo para que un pixel sea considerado como libre.
- *negate*: indica si la interpretación de blanco/negro y libre/ocupado es al revés.

ROS provee un nodo, llamado *map_server*, que lee un mapa del sistema de archivos y lo dispone para toda la topología. Al momento de ejecutarlo, hay que especificar un mapa (el archivo YAML) como parámetro.

Otros archivos de configuración

El Stack de Navegación utiliza dos archivos para guardar información sobre los obstáculos en el entorno. Uno de ellos, llamado *global_costmap*, es empleado para la planificación global (es decir, crear rutas a largo plazo sobre todo el entorno) y el otro, llamado *local_costmap* para la planificación local y evasión de obstáculos.

De acuerdo a ciertos parámetros en el archivo de configuración, se afecta al contenido de los dos archivos. Por ejemplo, dependiendo del valor de la variable *obstacle_range* se guarda o no información de los obstáculos (si estos están a determinado distancia son evaluados, sino son ignorados).

Además existe otro archivo YAML de configuración, llamado *base_local_planner_params*, el cual indica los límites de la velocidad del robot y de la aceleración.

Archivos lanzadores (launch files)

El framework ROS provee un package especial llamado *roslaunch*. Este paquete permite leer archivos lanzadores, de formato XML y extensión *.launch*. Además contiene más herramientas para ayudar al usuario a utilizar este tipo de archivos.

Muchos paquetes de ROS vienen con archivos lanzadores, que pueden ser ejecutados con el siguiente comando:

```
$ roslaunch package_name file.launch
```

Estos archivos ejecutan un conjunto de nodos (e incluso otros archivos lanzadores) y pueden estar parametrizados. De esta manera, los usuarios no tienen que ir ejecutando de a uno cada nodo de sus paquete o sistema. A continuación se presenta un ejemplo del Stack de Navegación:

```
<launch>
    <master auto="start"/>

    <!-- Run the map server -->
    <node name="map_server" pkg="map_server" type="map_server" args="$(find
    my_map_package)/my_map.pgm my_map_resolution"/>
    <!--- Run AMCL -->
    <include file="$(find amcl)/examples/amcl_omni.launch" />

    <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
        <rosparam file="$(find my_robot_name_2dnav)/costmap_common_params.yaml" command="load"
        ns="global_costmap" />
        <rosparam file="$(find my_robot_name_2dnav)/costmap_common_params.yaml" command="load"
        ns="local_costmap" />
        <rosparam file="$(find my_robot_name_2dnav)/local_costmap_params.yaml" command="load"
        />
        <rosparam file="$(find my_robot_name_2dnav)/global_costmap_params.yaml" command="load"
        />
        <rosparam file="$(find my_robot_name_2dnav)/base_local_planner_params.yaml" command="
        load" />
    </node>
</launch>
```

Figura A.6: Ejemplo de un archivo lanzador de ROS. En este caso, se utiliza para invocar a todos los componentes del Stack de Navegación.

Al ejecutar este archivo lanzador, se están ejecutando los nodos *map_server* del paquete *map_server* y *move_base* del paquete *my_nav_conf* (junto con los archivos de configuración *.yaml* como parámetro para ese nodo). Además, se invoca a otro lanzador (*amcl_omni.launch*).

Ejemplo del uso del Stack de Navegación

Una vez configurado y ejecutado el Stack de Navegación, se puede escribir el siguiente programa para mover al robot. Este programa será compilado y ejecutado, formando parte de la red de ROS como un nodo.

```
1  #include <ros/ros.h>
2  #include <move_base_msgs/MoveBaseAction.h>
3  #include <actionlib/client/simple_action_client.h>
4
5  typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
6
7  int main(int argc, char** argv){
8    ros::init(argc, argv, "simple_navigation_goals");
9
10   //tell the action client that we want to spin a thread by default
11   MoveBaseClient ac("move_base", true);
12
13   //wait for the action server to come up
14   while(!ac.waitForServer(ros::Duration(5.0))){
15     ROS_INFO("Waiting for the move_base action server to come up");
16   }
17
18   move_base_msgs::MoveBaseGoal goal;
19
20   //we'll send a goal to the robot to move 1 meter forward
21   goal.target_pose.header.frame_id = "base_link";
22   goal.target_pose.header.stamp = ros::Time::now();
23
24   goal.target_pose.pose.position.x = 1.0;
25   goal.target_pose.pose.orientation.w = 1.0;
26
27   ROS_INFO("Sending goal");
28   ac.sendGoal(goal);
29
30   ac.waitForResult();
31
32   if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
33     ROS_INFO("Hooray, the base moved 1 meter forward");
34   else
35     ROS_INFO("The base failed to move forward 1 meter for some reason");
36
37   return 0;
38 }
39
```

Figura A.7: Programa de ejemplo para mover al robot un metro hacia adelante mediante el Stack de Navegación.

En la *línea 2* se incluyen las referencias a ciertas funciones, servicios, tipos de datos del Stack de Navegación. *move_base_msgs* es un paquete que contiene distintos tipos de Mensajes (datos) utilizados para comunicarse con el nodo *move_base*, definidos de acuerdo a cierta interfaz. Esa interfaz es llamada *actionlib* (referenciada en la *línea 3*) que no es más que un conjunto de paquetes (*stack*) que provee una forma estándar para interactuar con ciertas funcionalidades del framework como mover la base de un robot de un punto a otro, realizar un escaneo con un láser, detectar cierto objeto (como el picaporte de una puerta) y muchas más. Específicamente, nos interesa el primer ejemplo, la función sendGoal, (*línea 28*), que recibe un cierto tipo de dato (*MoveBaseGoal*) que representa un punto en el mapa e intenta mover al robot a una posición/orientación del mundo.

Justamente esta función que provee la interfaz abstracta del framework, cumple con la interfaz definida por el *DSL para robots con misión predeterminada* para la entidad Robot. El Stack de Navegación es una de las funcionalidades más importantes que ofrece ROS, ya que esta colección de algoritmos que solucionan los problemas más comunes sobre la navegación de un robot. De esta manera, los desarrolladores se ahorran, no sólo mucho tiempo en re-implementar estas soluciones, sino en ignorar la implementación y trabajar con un sistema robótico desde un punto de vista más abstracto.

Apéndice B

Manual de usuario - Editor de Misiones

En este anexo se detalla un manual para el uso de la herramienta *Editor de Misiones*, desarrollada para completar el DSL presentado en esta tesina. Además se muestra, en su totalidad, cómo se concretiza la sintaxis abstracta descrita en el capítulo 5 en esta herramienta, describiendo de este modo las relaciones entre ambas sintaxis.

B.1 Creación de un nuevo mapa y edición de un mapa existente.

Para la creación de un nuevo **Mapa**, se deben seguir los siguientes pasos:

1. Seleccionar *Editar* - *Editor de Mapas*, como se muestra en la figura a continuación.

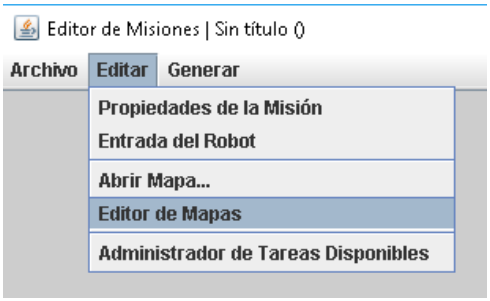

Figura B.1: Opción a seleccionar para abrir el editor de mapas.

2. Dentro de la ventana que se abre, seleccionar *Archivo* - *Nuevo Mapa*, acción que creará una nueva instancia de la clase **Mapa**, con los atributos y referencias inicializados como se muestra en la siguiente tabla:

Tipo	Nombre	Valor
Atributo	imagen	null
Atributo	invertido	false
Referencia	origenCoordenadas	Punto(0,0)
Referencia	aspecto	instanciaAspectoMapa

Donde *instanciaAspectoMapa* será una nueva instancia de la clase textbfAspectoMapa, con los atributos *alto*, *ancho* y *resolucion* inicializados en *0*, *0* y *0.0* respectivamente.

3. Presionar el botón *Seleccionar archivo imagen...* y escoger una imagen PNG que represente a un mapa, como se puede ver en la B.2.

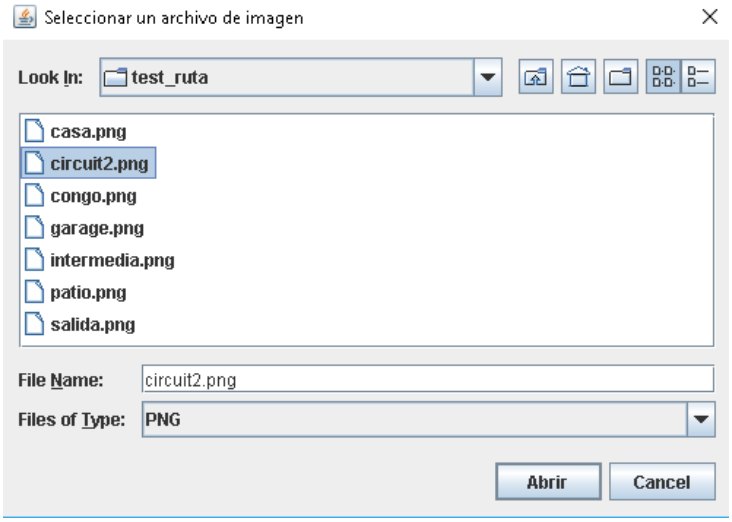

Figura B.2: Selección de una imagen PNG como mapa.

Al cliquear *Abrir*, el atributo imagen de la instancia de **Mapa** se fijará con la ruta completa al archivo elegido, en este caso será algo similar a ".../test_ruta/circuit2.png"; además, se modificarán los atributos *alto* y *ancho* de *instanciaAspectoMapa* con el alto y el ancho de la imagen escogida.

4. Se pueden modificar los demás atributos del mapa, en este ejemplo modificamos la resolucion para que sea de 0.5.

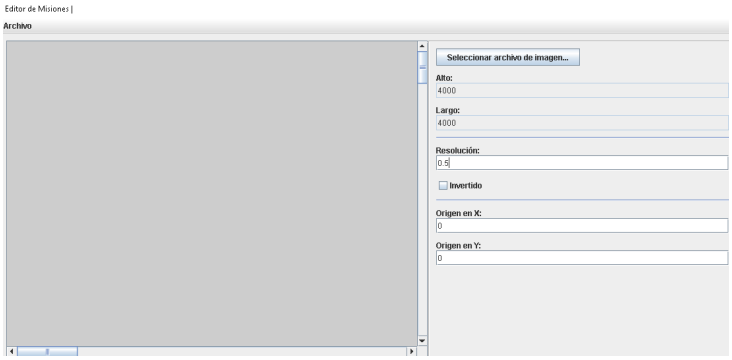

Figura B.3: Ejemplo de modificación de los atributos de un mapa.

Con los cambios realizados anteriormente, se modifica el atributo *resolucion* de *instanciaAspectoMapa* al valor *0.5*.

Una vez realizados todos los cambios deseados, se tiene que seleccionar *Archivo - Guardar Mapa* y elegir un nombre de archivo que tenga la extension ".xmi". Si se desea cambiar de nombre, se puede escoger *Archivo - Guardar Mapa Como...* y poner el nuevo nombre del archivo.

Si se desea editar un mapa, se debe seleccionar *Archivo - Abrir Mapa...* y abrir el archivo deseado, mediante lo que se cargará en una instancia de **Mapa** todos los atributos y referencias con los valores extraídos del archivo. Luego, se procederá con la modificación de los parámetros deseados y se deberá guardar el mapa como se explicó anteriormente.

B.2 Creación de una nueva misión.

Para la creación de una nueva **Mision** vacía se debe seleccionar *Archivo - Nueva Misión*, tal como se muestra en la imagen que está a continuación.

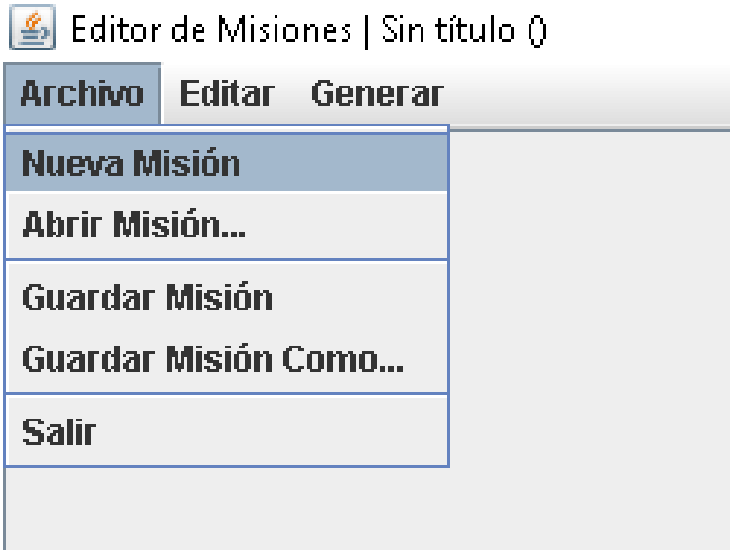

Figura B.4: Creación de una nueva misión vacía.

Una vez realizado el paso anterior, se creará una nueva instancia de la clase **Mision** con los atributos y referencias seteados del siguiente modo:

Tipo	Nombre	Valor
Atributo	titulo	"Sin título"
Atributo	autor	"Desconocido"
Atributo	visionGeneral	
Atributo	cantidadRepeticiones	1
Referencia	robot	*nuevoRobot*
Referencia	mapa	null
Referencia	objetivos	lista vacía
Referencia	tareasDisponibles	*listaTareas*

Como se muestra en la tabla anterior, la nueva misión aún no tendrá título, ni autor, ni una visión general seteadas, mientras que sí tendrá seteada por defecto la cantidad de repeticiones en el valor 1, una lista de objetivos creada sin ningún **Objetivo** cargado, una lista de tareas disponibles creada sin ninguna **TareaPersonalizada** cargada (pero sí con una **TareaDormir** cargada), y un robot establecido nombrado *nuevoRobot*, que será una nueva instancia de la clase Robot, inicializada de la siguiente manera:

Tipo	Nombre	Valor
Atributo	orientacionInicial	0
Referencia	posicionInicial	Punto(0,0)

B.3 Modificación de las propiedades de una misión.

Para modificar las propiedades básicas de una misión, se debe seleccionar la opción *Editar* - *Propiedades de la Misión*, como se puede ver en la Figura B.5.

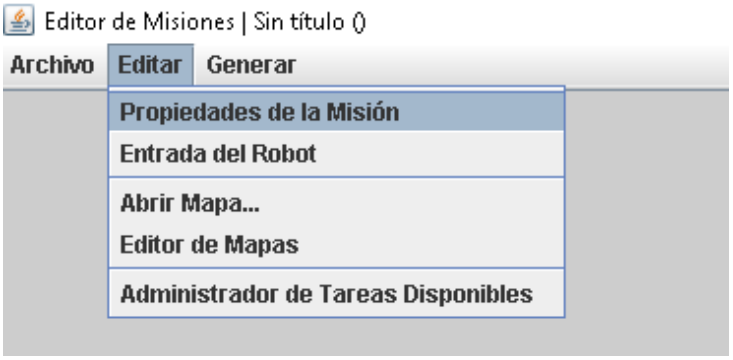

Figura B.5: Opción a elegir para editar las propiedades de la misión.

Se abrirá una ventana y se podrán modificar las propiedades básicas de la misión, como título, autor, visión general y cantidad de repeticiones. A continuación, se muestra un ejemplo.

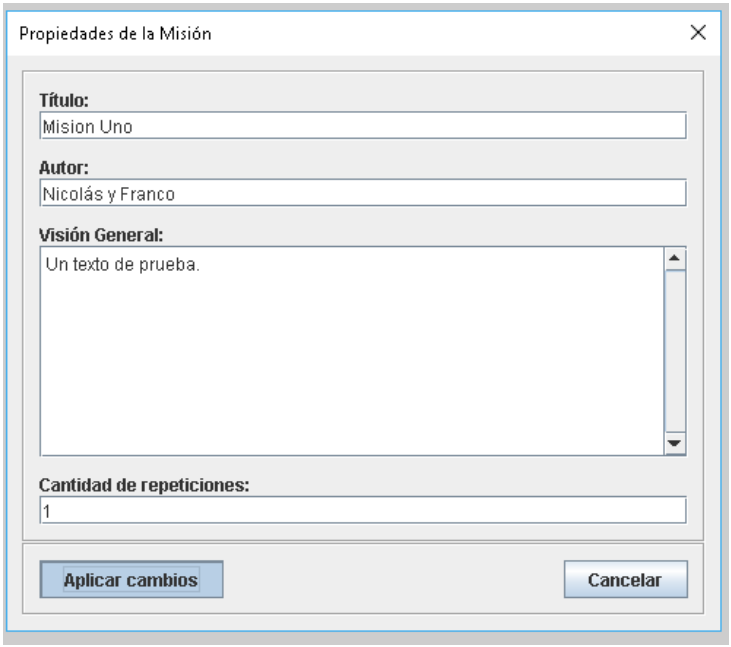

Figura B.6: Ejemplo de modificación de las propiedades de una misión.

Una vez cliqueado el botón *Aplicar Cambios*, se modificarán los atributos de la misión, quedando con los siguientes valores:

Nombre	**Valor**
titulo	"Misión Uno"
autor	"Nicolás y Franco"
visionGeneral	"Un texto de prueba."
cantidadRepeticiones	1

En caso de escoger la opción *Cancelar* no se realizará ninguna modificación.

B.4 Creación de nueva tarea personalizada.

Para la creación de una nueva tarea personalizada en la misión, se debe:
1. Seleccionar la opción *Editar - Administrador de Tareas Disponibles.*

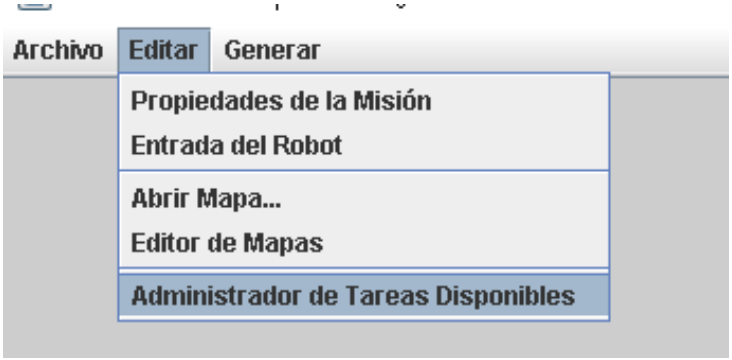

Figura B.7: Opción a seleccionar para abrir el administrador de tareas disponibles.

2. Presionar el botón *Nueva.* 3. Completar los campos como se muestra en la Figura B.8 y cliquear el botón *Confirmar.*

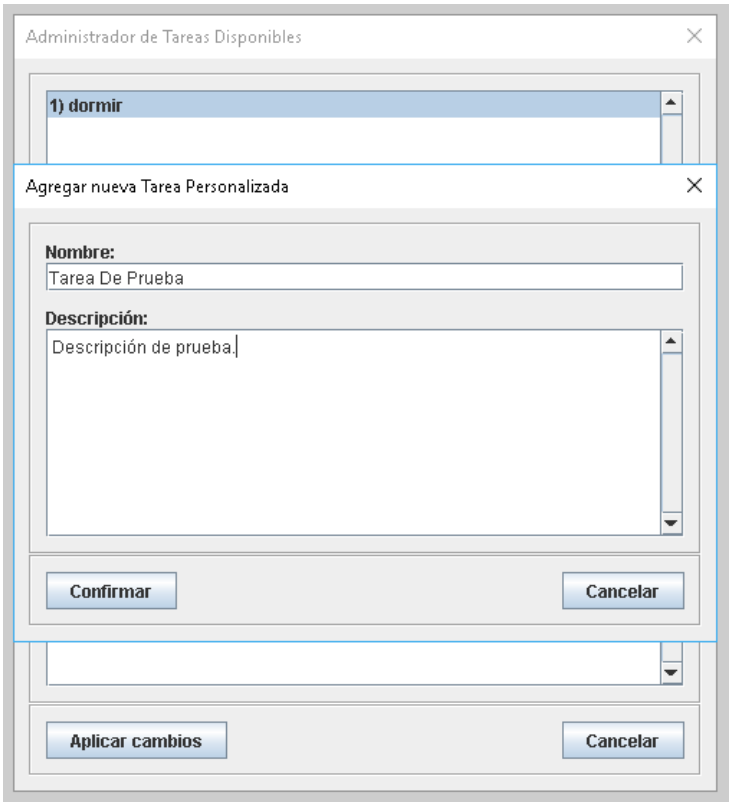

Figura B.8: Ejemplo de creación de una nueva tarea personalizada.

4. Una vez agregadas todas las tareas deseadas, se debe seleccionar el botón *Aplicar Cambios.*
Una vez hecho lo anterior se creará una nueva instancia de la clase **TareaPersonalizada** con los atributos nombre seteado en "*Tarea De Prueba*" y *descripcion* en "*Descripción de prueba.*", y se agrega esa instancia a la lista referenciada por *tareasDisponibles* de la instancia de la clase **Mision**.

B.5 Edición de tareas personalizadas.

Para la edición de las tareas personalizadas de la misión, lo primero que se debe hacer al igual que en la sección anterior es seleccionar la opción *Editar - Administrador de Tareas Disponibles.*

1. Para modificar una tarea existente, se debe seleccionar dicha tarea y cambiar los campos que se desee. Luego, presionar el botón *Actualizar*. En la Figura B.9 se puede ver un ejemplo.

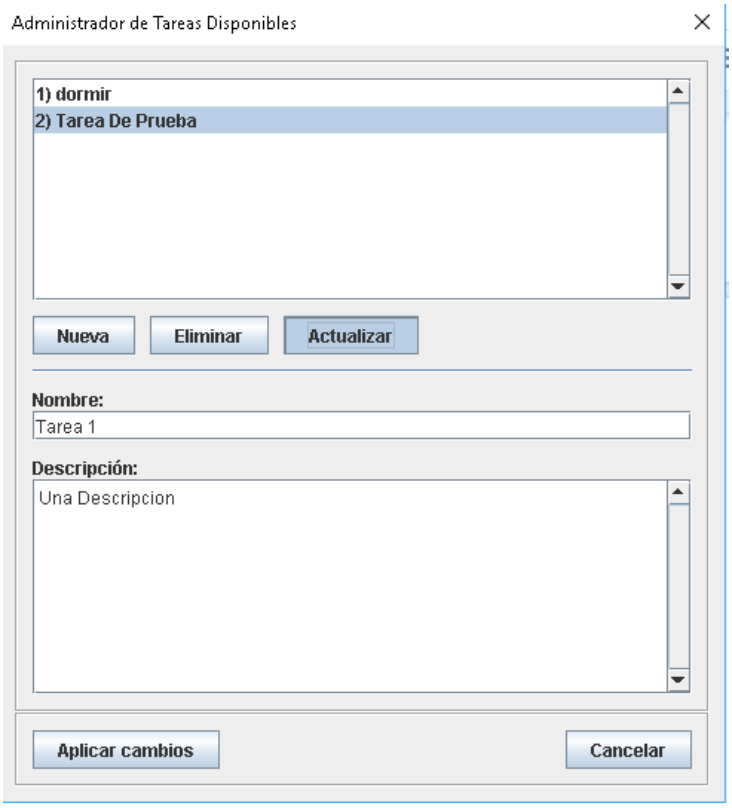

Figura B.9: Ejemplo de modificación de una tarea personalizada.

Al realizar lo expuesto por la figura, se modificarán los atributos *nombre* y *descripcion* de la *Tarea De Prueba* (instancia de la clase **TareaPersonalizada**), y quedarán con los valores "*Tarea 1*" y "*Una Descripción.*" respectivamente.

2. Para eliminar una tarea existente, se debe seleccionar la tarea a eliminar y presionar el botón *Eliminar*, tal como se puede ver en la Figura B.10. Lo que ocurrirá será la eliminación de esa instancia de **TareaPersonalizada** de la lista *tareasDisponibles* de la instancia de la clase **Mision**.

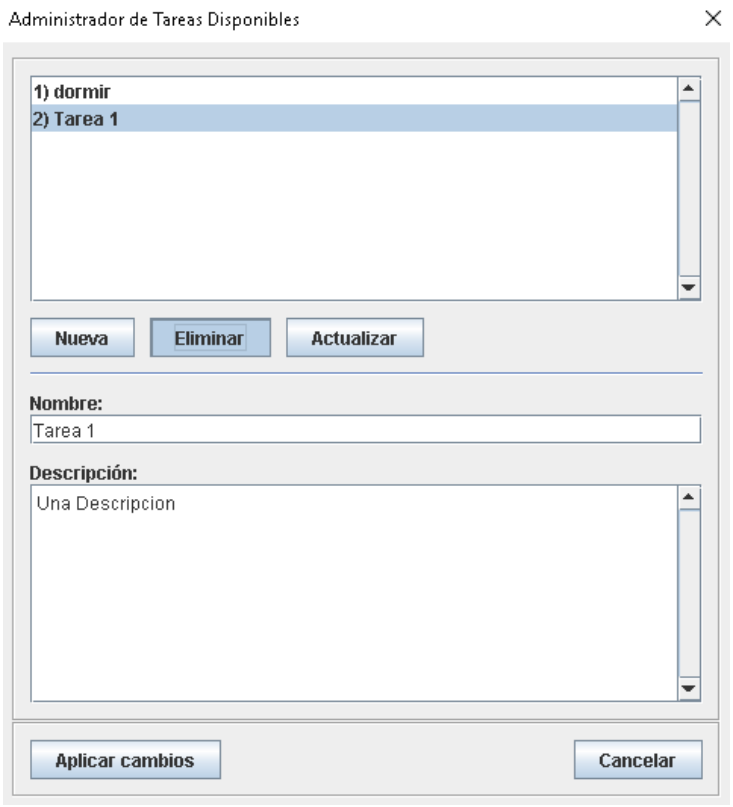

Figura B.10: Ejemplo de eliminación de una tarea personalizada.

B.6 Asignación de un mapa a una misión.

Para la asignación de un mapa válido a una misión, se debe escoger la opción *Editar - Abrir Mapa...* y seleccionar un archivo de mapa válido. En las Figuras B.11 y B.12 se muestra lo mencionado.

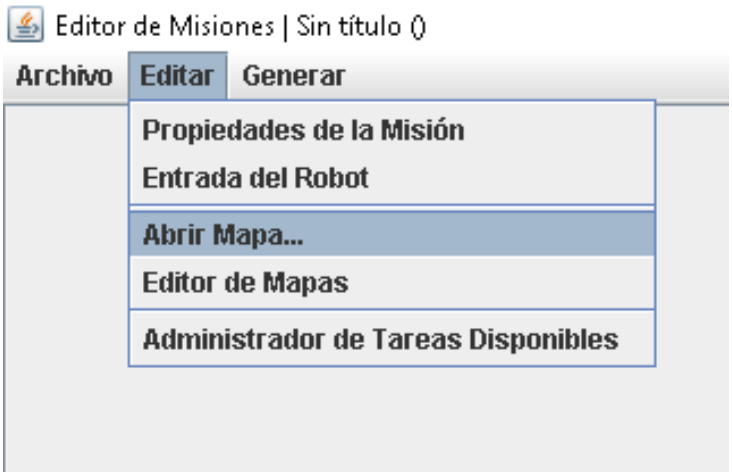

Figura B.11: Opción para designación de mapa.

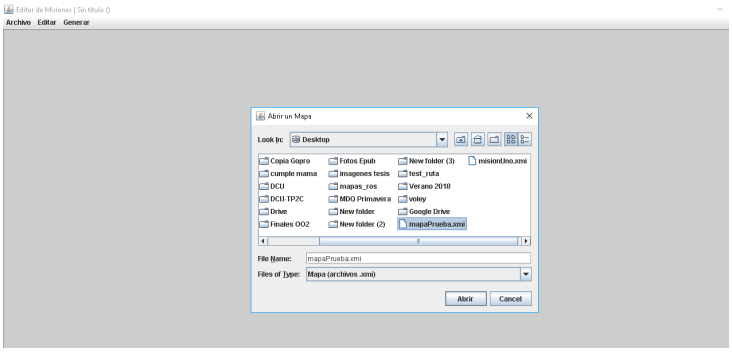

Figura B.12: Ejemplo de selección de un archivo mapa válido.

Una vez asignado el mapa, se guardará una referencia al mismo en el atributo *mapa* de la clase **Mision**.

B.7 Modificación de los parámetros iniciales del robot.

Para indicar la posición inicial del robot hay dos formas:

1. Hacer clic derecho sobre una posición del mapa y seleccionar la opción *Marcar Posición Inicial del Robot*.

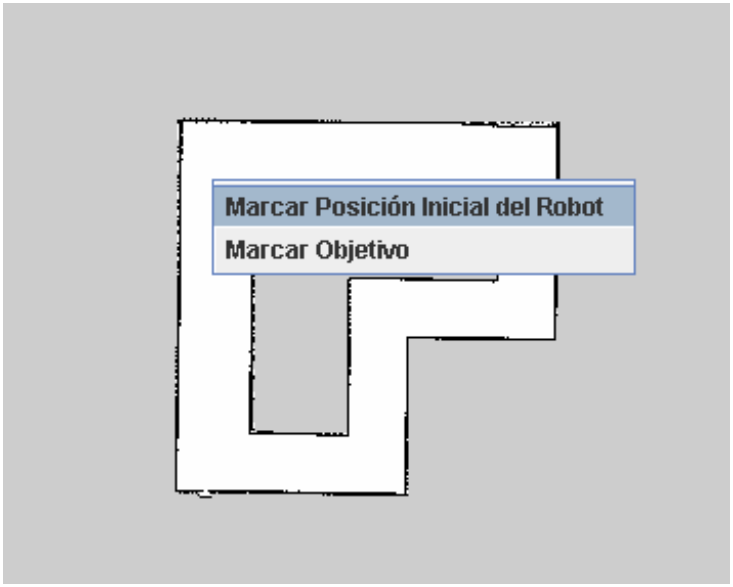

Figura B.13: Selección de posición inicial del robot mediante clic en el mapa.

2. Seleccionar la opción *Editar - Entrada del Robot* y editar los datos de la ventana presentada (la misma se visualiza en la Figura B.14). A diferencia de la opción anterior, en ésta será posible también cambiar la orientación inicial del robot.

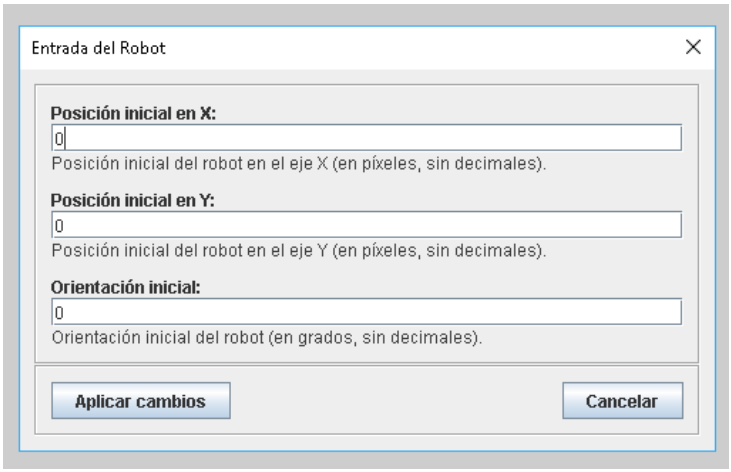

Figura B.14: Edición de los datos iniciales del robot.

3. La posición inicial del robot se guardará en la referencia *posicionInicial* (como una instancia de la clase **Punto**) del **Robot** que ejecuta la **Mision**, y la orientación inicial en el atributo *orientacionInicial*.

Una vez marcada la posición inicial, se podrá distinguir en el mapa dicha posición mediante un globo de color verde, como se muestra en la Figura B.15.

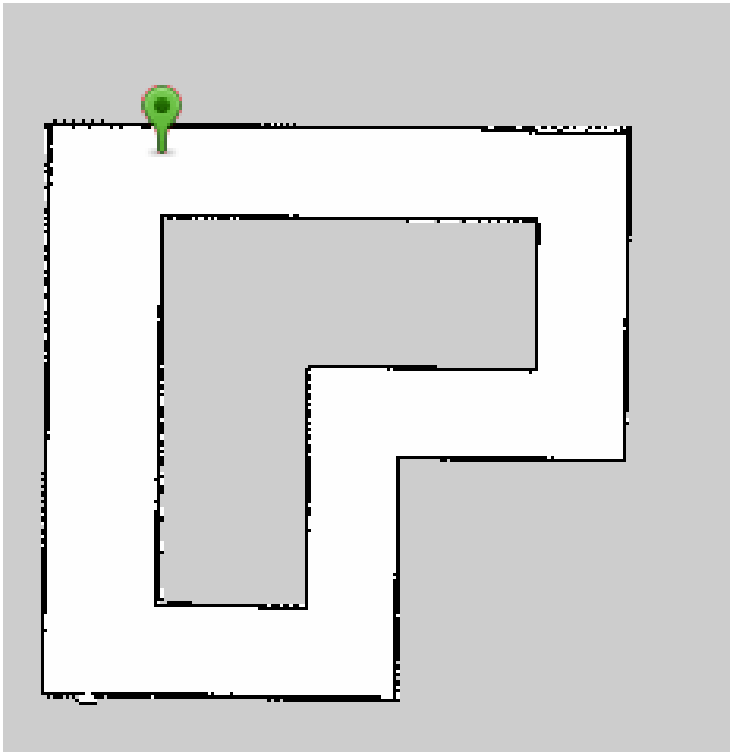

Figura B.15: Ejemplo de un mapa con la posición inicial del robot seteada.

B.8 Inserción y edición de objetivos en la misión.

Para insertar objetivos en la misión, se deberá hacer clic derecho sobre un lugar del mapa y seleccionar la opción *Marcar Objetivo*, como se puede visualizar en la Figura B.16.

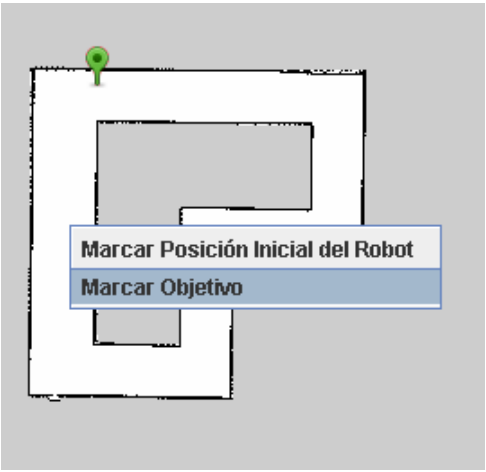

Figura B.16: Inserción de un nuevo objetivo en el mapa.

Cuando se inserte un nuevo objetivo en el mapa, se creará una nueva instancia de la clase **Objetivo** con los atributos y referencias inicializados como se muestra en la tabla de abajo, y se agrega esa instancia a la lista de objetivos (referencia *objetivos*) de la instancia de la clase **Mision**.

Tipo	Nombre	Valor
Atributo	comentario	
Atributo	obligatorio	false
Referencia	ubicacion	Punto(x_1,y_1)*
Referencia	tareas	lista vacía

Cuadro B.1: * Donde x_1 y y_1 corresponden a los valores x e y del mapa en el que se insertó el nuevo objetivo.

Si se desea que un objetivo sea obligatorio, se debe hacer clic derecho sobre dicho objetivo y seleccionar la opción *[] Obligatorio*, lo cual cambiará al atributo *obligatorio* asignándole el valor *true*. En la siguiente imagen se puede ver lo dicho anteriormente.

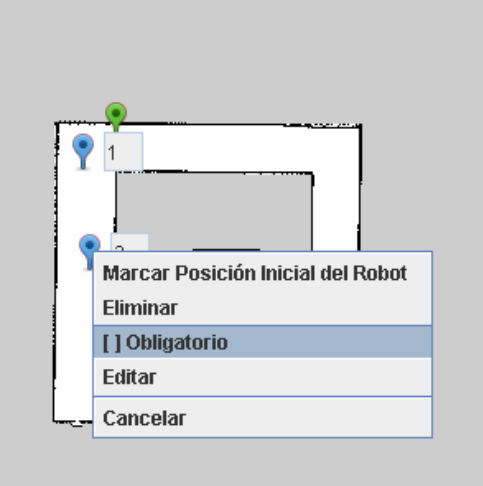

Figura B.17: Designación de un objetivo como obligatorio.

A continuación se puede apreciar una imagen con objetivos obligatorios (se distinguen al estar en color rojo) y no obligatorios (están en color azul).

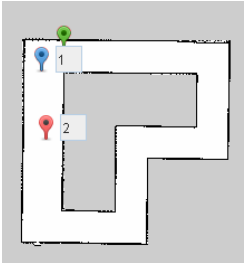

Figura B.18: Ejemplo de mapa con un objetivo obligatorio y otro no obligatorio.

Si se deseara que un objetivo que es obligatorio deje de serlo, se debe hacer clic derecho sobre dicho objetivo y seleccionar la opción *[X] Obligatorio*, lo cual cambiará al atributo *obligatorio* asignándole el valor *false*, y consecuentemente, dicho objetivo aparecerá ahora en color azul.

Para editar un objetivo, se debe seleccionar dicho objetivo con clic derecho y clickear la opción *Editar*, como se muestra en la Figura B.19.

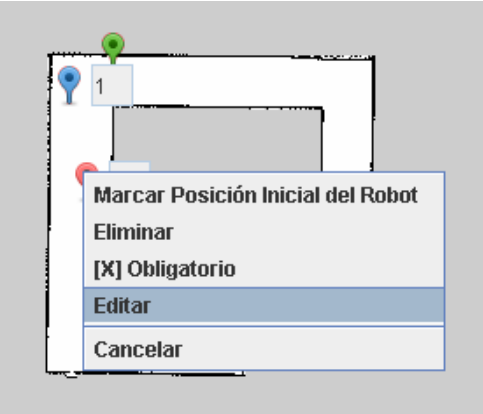

Figura B.19: Opción a seleccionar para la edición de un objetivo.

Se abrirá una ventana como la que se puede visualizar en la Figura B.20, la cuál permitirá la edición de atributos y referencias del objetivo.

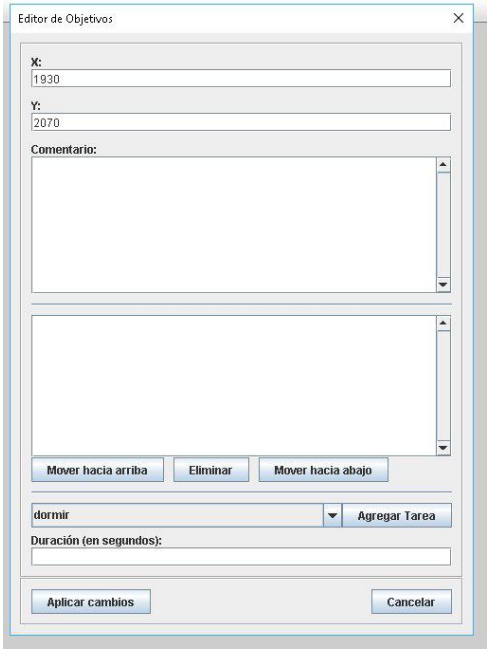

Figura B.20: Ventana para edición de un objetivo.

Si se modifican los campos x e y, se creará una nueva instancia de la clase **Punto** con los valores seteados y se asignará esa instancia a la referencia *ubicacion* del objetivo actual. Lo mismo ocurre al modificarse el campo *comentario*, provocando un cambio en el atributo *comentario* del objetivo en cuestión. Siguiendo, para agregar una nueva tarea personalizada se debe seleccionar la tarea deseada y cliquear en el botón *Agregar Tarea*.

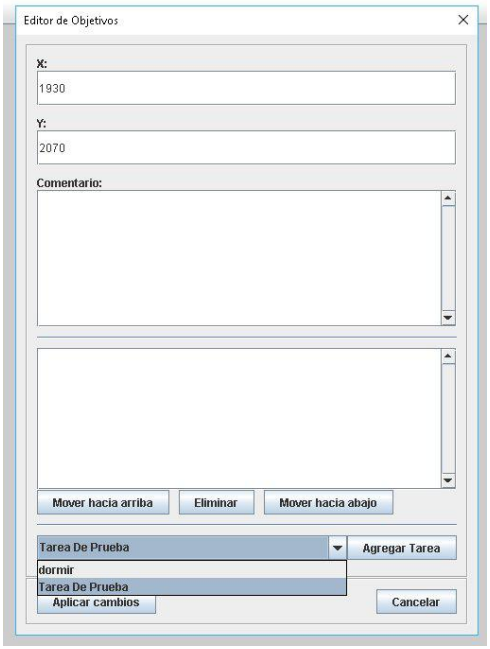

Figura B.21: Ejemplo de agregación de una tarea personalizada.

Al realizar el paso anterior tal como se muestra en la imagen, lo que ocurrirá será la creación de una instancia de la clase **TareaPersonalizada**, con el atributo *nombre* seteado con el valor *T̈area De Prueba¨* y el atributo *descripcion* seteado con el valor *D̈escripcion De Prueba¨*. Esa instancia creada, será agregada a la lista referenciada como *tareas* del objetivo actual. Para agregar una tarea *dormir*, se debe seleccionar dicha tarea y completar la duración en segundos que tendrá la misma, como se puede ver en la Figura B.22.

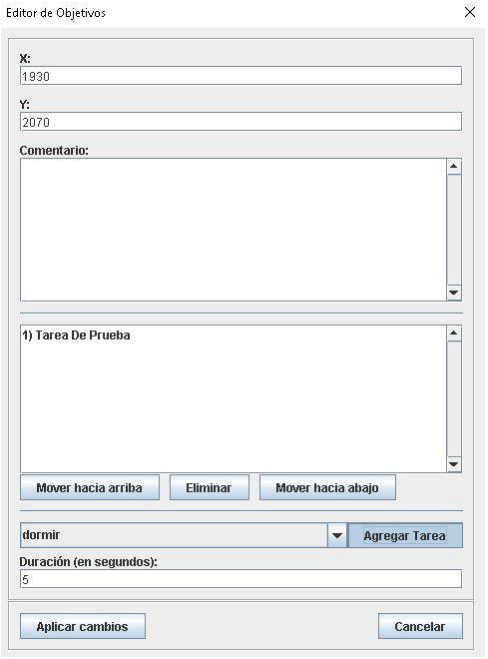

Figura B.22: Ejemplo de agregación de una TareaDormir de 5 segundos de duración.

Al realizar el paso anterior tal como se muestra en la imagen, lo que ocurrirá será la creación de una instancia de la clase **TareaDormir**, con el atributo *duracion* seteado con el valor *5*, el atributo *nombre* seteado con el valor *"dormir"* y el atributo *descripcion* seteado con el valor *."El robot no ejecuta ninguna operación durante una cierta cantidad de tiempo (en segundos).".* Esa instancia creada, será agregada a la lista referenciada como *tareas* del objetivo actual. Es de suma importancia tener en cuenta que para que los cambios se efectúen correctamente, al finalizar se debe seleccionar la opción *Aplicar Cambios*, ya que en caso de cerrarse la ventana con la cruz de la esquina superior derecha o con el botón *Cancelar*, los cambios realizados serán descartados.

Bibliografía

[1] Claudia Pons, Roxana Silvia Giandini, Gabriela Pérez. *Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica.* Editorial Edulp, ISBN: 9789503406304, 2010.

[2] Aaron Martinez, Enrique Fernández. *Learning ROS for Robotics Programming.* Editorial Packt Publishing, ISBN: 9781782161448, 2013.

[3] Claudia Pons, Carlos Neil, Roxana Silvia Giandini, Gabriela Pérez, Marcelo De Vincenzi. *Un enfoque dirigido por modelos para la creación de sistemas robóticos.* XVI Workshop de Investigadores en Ciencias de la Computación, 2014.

[4] Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, Massimo Tivoli. *Automatic generation of detailed flight plans from high-level mission descriptions.*. Conference: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. DOI: 10.1145/2976767.2976794, 2010.

[5] Davide Brugali, Patrizia Scandurra. *Component-Based Robotic Engineering (Part I) [Tutorial].* Robotics & Automation Magazine, IEEE. 16. 84 - 96. 10.1109/MRA.2009.934837, 2010.

[6] Davide Brugali, Azamat Shakhimardanov. *Component-Based Robotic Engineering (Part II).* Robotics & Automation Magazine, IEEE. 17. 100 - 112. 10.1109/MRA.2010.935798, 2010.

[7] Johannes Baumgartl, Thomas Buchmann, Dominik Henrich, Bernhard Westfechtel. *Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines.*. 8th International Conference on Software Paradigm Trends (ICSOFT-PT 2013), 2013.

[8] Dieter Fox, Wolfram Burgard, Frank Dellaert, Sebastian Thrun *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots.* Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence. Pages 343-349. 1999. `http://dl.acm.org/citation.cfm?id=315149.315322`

[9] Tae-bum Kwon, Ju-ho Yang, Jae-bok Song *Efficiency Improvement in Monte Carlo Localization through Topological Information.* 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems. Pages 424-429. 2006.

[10] Object Management Group (OMG)
`https://omg.org/`

[11] Unified Modeling Language (UML)
`http://www.uml.org/`

[12] MetaObject Facility (MOF)
https://www.omg.org/mof/

[13] Eclipse Foundation
https://www.eclipse.org/

[14] Eclipse Modeling Framework (EMF)
https://www.eclipse.org/emf/

[15] Acceleo
http://www.eclipse.org/acceleo/

[16] Robot Operating System
http://www.ros.org/

[17] ROS Wiki
http://wiki.ros.org/es

[18] Navigation Stack - ROS
http://wiki.ros.org/navigation

[19] FLYAQ - Enabling Non-Expert Users to Program Missions of Autonomous Multicopters
http://www.flyaq.it/

[20] Stanford Artificial Intelligence Laboratory
http://ai.stanford.edu/

[21] Willow Garage
http://www.willowgarage.com

[22] Gazebo - Robot simulation made easy
http://gazebosim.org/

[23] TurtleBot - Personal, low-cost robot kit with open-source software
https://www.turtlebot.com/

[24] 2D Robot Mapping Software for autonomous navigation
https://www.robotshop.com/
letsmakerobots/2d-robot-mapping-software-autonomous-navigation

[25] iRobot Corporation
https://www.irobot.com

[26] iRobot Create's Open Interface
https://www.irobot.com/filelibrary/pdfs/hrd/create/Create Open
Interface_v2.pdf

[27] Autonomy Lab. *Simon Fraser University, Canada.*
http://autonomy.cs.sfu.ca/

[28] create_autonomy: ROS driver for iRobot Create 1 and 2. *Jacob Perron (Autonomy Lab, Simon Fraser University).*
https://github.com/AutonomyLab/create_autonomy