UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

NAVEGACIÓN DE UN ROBOT MÓVIL EN PRESENCIA DE OBSTÁCULOS NO MAPEADOS

MEMORIA

TRABAJO DE FIN DE GRADO

Presentada por:

Adrián González Sieira

Dirigida por:

Dr. Manuel Mucientes Molina

Santiago de Compostela, Julio de 2010

Índice de contenido

Capítulo 1

Prefacio

1.1. Contextualización

El concepto de navegación se puede definir como la metodología que permite guiar el curso de un robot móvil desde un punto hasta un objetivo definido a través de un entorno con obstáculos, conocidos o no. La prioridad en este tipo de problemas está en realizar el desplazamiento siempre de forma lo más segura posible; para esto debe dotarse al robot de la capacidad de reaccionar ante situaciones inesperadas. De esta forma se consigue que tenga la suficiente autonomía para desenvolverse en entornos no estructurados, entendiendo como tales aquellos en los que existen obstáculos de los que no se conoce nada previamente, esto es, que no están *mapeados*.

Para llevar a cabo las tareas que involucra el proceso de navegación es necesario realizar una serie de pasos, entre los que se encuentran: percibir el entorno a través de una serie de sensores con los que el robot está equipado, de tal forma que se pueda recopilar información sobre el entorno que permita llevar a cabo un control de las acciones a llevar a cabo; y planificar una trayectoria libre de obstáculos que permita alcanzar el punto objetivo seleccionado. Si se desea dar una respuesta en tiempo real a esta última tarea, lo más adecuado es definir un controlador, también llamado comportamiento en el ámbito de la robótica móvil, y que las acciones se planifiquen de forma dinámica en base a él; esto supone una dificultad añadida, pues la definición del comportamiento manualmente es un proceso muy costoso, por lo que lo más adecuado sería realizar un aprendizaje de éste, de tal forma que el robot pueda hacer frente a todo tipo de situaciones en las que se pueda encontrar en base a este comportamiento.

Para el modelado del controlador que debe ejecutar el sistema de navegación se puede utilizar una base de reglas borrosas, que realice un proceso de inferencia partiendo de la información que se recoge sobre el entorno a través de los sensores, y que sirva para la obtención de una serie de salidas que definirán los comandos de control que el robot debe ejecutar para alcanzar el objetivo de forma segura. El proceso de aprendizaje debe ser previo a la ejecución de la navegación, y debe obtenerse como salida la base de conocimiento que permita llevarlo a cabo, esto puede realizarse a través de técnicas de aprendizaje automático, como por ejemplo un algoritmo evolutivo.

1.2. Objetivos del proyecto

El objetivo principal de este proyecto es la realización de una herramienta que realice el aprendizaje automático del navegador de un robot móvil, mediante la utilización de un algoritmo evolutivo. Esta herramienta permitirá la definición de una serie de parámetros para llevar a cabo el proceso de aprendizaje, permitiendo la reducción del tiempo de desarrollo del sistema de control del robot. En concreto, la definición de los objetivos del presente proyecto es la siguiente:

- Implementación de una herramienta que utilice la computación evolutiva para realizar el aprendizaje automático del navegador del robot. La salida de esta herramienta será un comportamiento definido en forma de una base de reglas borrosas, que será uno de los componentes esenciales del controlador del robot, y que contendrá la información necesaria para hacer frente al problema de la navegación autónoma.

- Desarrollar un controlador borroso que dé respuesta al problema de la navegación del robot, utilizando el conocimiento que modela la base de reglas que la herramienta anterior ofrece como salida.

- Realizar una serie de pruebas de simulación de las herramientas desarrolladas en los pasos anteriores, de forma que se pueda analizar su funcionamiento en un entorno controlado, estudiar los resultados, y realizar el ajuste de los parámetros del proceso de aprendizaje hasta obtener una solución de la calidad deseada.

1.3. Organización del documento

A lo largo del presente documento explicará con detalle el conjunto de procedimientos y tecnologías que se han utilizado para lograr el cumplimiento de los objetivos del proyecto. A continuación se detalla el contenido de cada uno de los capítulos de esta memoria:

- En el capítulo 2 se realiza una contextualización de la aproximación que se va a llevar a cabo para dar una solución al problema que se plantea en este proyecto. En él se hará una descripción general del ámbito de conocimiento que es necesario para abordarlo, esto es, robótica móvil, algoritmos evolutivos y lógica difusa.

- En el capítulo 3 se realiza una descripción pormenorizada del algoritmo que se va a implementar para la herramienta de aprendizaje, definiendo cada uno de los aspectos que resultan esenciales para la definición de un algoritmo evolutivo, esto es, el conjunto de operadores utilizados, la representación de los individuos de la población y la definición de la función de evaluación con la que se va a a trabajar.

- En el capítulo 4 se detalla el análisis que ha sido necesario como paso previo al diseño e implementación. Dentro de este capítulo se incluyen: el análisis de riesgos, de librerías, tecnologías y herramientas que se pretenden utilizar para llevar a cabo el desarrollo del proyecto.

- El capítulo 5 contiene una explicación pormenorizada de la metodología de desarrollo utilizada, el conjunto de riesgos que se han identificado y la planificación temporal y económica de cada una de las tareas que se han definido dentro de dicha metodología.
- El capítulo 6 recoge el conjunto de pasos que se han llevado a cabo para diseñar e implementar la herramienta, explicando también las consideraciones que se han tenido para realizarla, así como las posibles modificaciones respecto a la aproximación inicial a la solución del problema que se hayan tenido que realizar por diversas razones.
- En el capítulo 7 se detalla el conjunto de pruebas realizadas para llevar a cabo la validación de la aplicación. En este paso se realizará un estudio de la influencia que los diferentes parámetros tienen dentro del proceso de aprendizaje, y cuáles deberían ser los valores que permitan obtener los mejores resultados.
- En el capítulo 8 se exponen las conclusiones del proceso de desarrollo y validación, resumiendo en qué medida se han cumplido los objetivos del proyecto que se han detallado inicialmente, incluyendo posibles mejoras y ampliaciones que se podrían llevar a cabo sobre el mismo.
- Para finalizar, en el apéndice A se incluye una pequeña guía de utilización de instalación y de utilización de la aplicación, así como un manual de instalación y uso del programa de simulación Player/Stage para poder visualizar el resultado de la ejecución de la herramienta de aprendizaje.

Capítulo 2

Introducción

El presente proyecto abarca conceptos de diferentes áreas de conocimiento, como la robótica, la computación evolutiva y la lógica borrosa. Para poder realizar un estudio más detallado del mismo es necesario introducir previamente algunas nociones básicas a tener en cuenta durante todo el proceso de desarrollo.

A lo largo de este capítulo se realizará una presentación sobre las ideas básicas que rigen la robótica móvil, la lógica borrosa y los algoritmos evolutivos, para finalizar posteriormente con la descripción de los objetivos de este proyecto, aunque definidos en más detalle.

2.1. Robótica móvil

Un robot es una entidad, que puede ser tanto virtual como mecánica, generada artificialmente, que persigue un propósito determinado. En la actualidad los robots presentan numerosas aplicaciones en el ámbito comercial ya que están capacitados para la realización de diferentes tipos de tareas de un modo preciso, y a menudo más barato que las realizadas por seres humanos, y su utilización es cada vez más frecuente en todo tipo de ámbitos.

El ámbito de realización de este proyecto está enfocado hacia los robots móviles, uno de sus principales objetivos es el desplazamiento, que puede buscar diferentes fines, como el transporte de materiales, reconocimiento de una zona, exploración y búsqueda de elementos, etc. Su movimiento puede venir dado a través de elementos rodantes u otros mecanismos. Éstos responden a la necesidad de extender el campo de aplicación de la robótica mas allá del alcance de una estructura anclada a un punto, incrementando su autonomía y limitando en todo lo posible la intervención humana necesaria para llevar a a cabo sus funciones. En este sentido es importante destacar que, para lograr la autonomía que se desea, es necesario dotar al robot de la suficiente inteligencia para tomar decisiones en función de las observaciones que toma de su entorno, teniendo en cuenta además que puede no ser completamente conocido.

En la robótica móvil, la autonomía de un robot depende directamente de un sistema de navegación automática. Típicamente éste realiza tareas de planificación de rutas, percepción y control. De forma

general el problema de la planificación se puede descomponer en una serie de etapas:

1. *Definición del objetivo:* aquel al que se desea llegar desde la posición inicial.
2. *Cálculo de la ruta:* sin tener en cuenta nada más que la información del entorno conocida, traza una ruta desde el punto inicial hasta el punto objetivo, definiendo varios puntos de paso intermedios.
3. *Evasión de obstáculos no esperados:* son objetos que interrumpen la trayectoria del robot y de los que no se dispone información previa, esto es, no están *mapeados*, y por tanto no se pueden tener en cuenta en el paso anterior, de cálculo de la ruta inicial.

Para poder llevar a cabo estas operaciones es necesario disponer de una serie de datos sobre el estado del robot, como su posición y su orientación, en unos intervalos de tiempo lo suficientemente cortos como para permitir el control del robot de un modo fluido; se pueden obtener de modo directo a través de la odometría del robot. En cuanto al sistema de percepción, su misión es la de permitir una navegación segura, localizando obstáculos y modelando el entorno alrededor del robot. Aunque en robots móviles esto se utiliza fundamentalmente para poder recalcular la ruta cuando el robot se encuentra con algún objeto imprevisto que no le permite alcanzar el punto objetivo, también tiene aplicaciones en robots poliarticulados, donde el mecanismo manipulador del entorno debe conocerlo exactamente para poder operar con seguridad.

El sistema de percepción está compuesto por un conjunto de sensores que proporcionan información valiosa para el sistema de navegación. Respecto a este conjunto de sensores hay que prestar atención a sus características como la precisión, el rango de medición, y su robustez, entre otras cosas. Unos de los de los sensores más utilizados son láser, que permiten un barrido de la escena identificando las distancias a las que se encuentran los obstáculos.

Para poder coordinar las tareas necesarias para ejecutar el sistema que dota al robot de autonomía es necesario utilizar una arquitectura de control que proporcione una estructura en la que organizar el sistema. Las existentes se agrupan en tres categorías:

- *Arquitecturas jerárquicas:* definen un ciclo en el que el objetivo principal es la reducción del error, que está considerado como la diferencia existente entre el estado del robot y el estado objetivo. Durante este ciclo se llevan a cabo las siguientes operaciones de forma ordenada:

  1. *Modelado del entorno* a través de los sensores del robot.
  2. *Planificación de la acción* en base al sistema de control implementado, y la información recogida en el paso anterior.
  3. *Actuación*, es decir, el envío de la orden a los efectores del robot para pasar a un nuevo estado.

  La desventaja de este tipo de sistemas, cuya arquitectura se representa en la figura 2.1 es que la información que se recoge de los sensores es procesada según una representación del entorno que implementa el propio sistema, y requiere una planificación que es muy costosa computacionalmente, por lo que el sistema de control pierde capacidad de reacción. Es adecuado para entornos poco cambiantes y altamente estructurados, pero no para su utilización en robótica móvil.

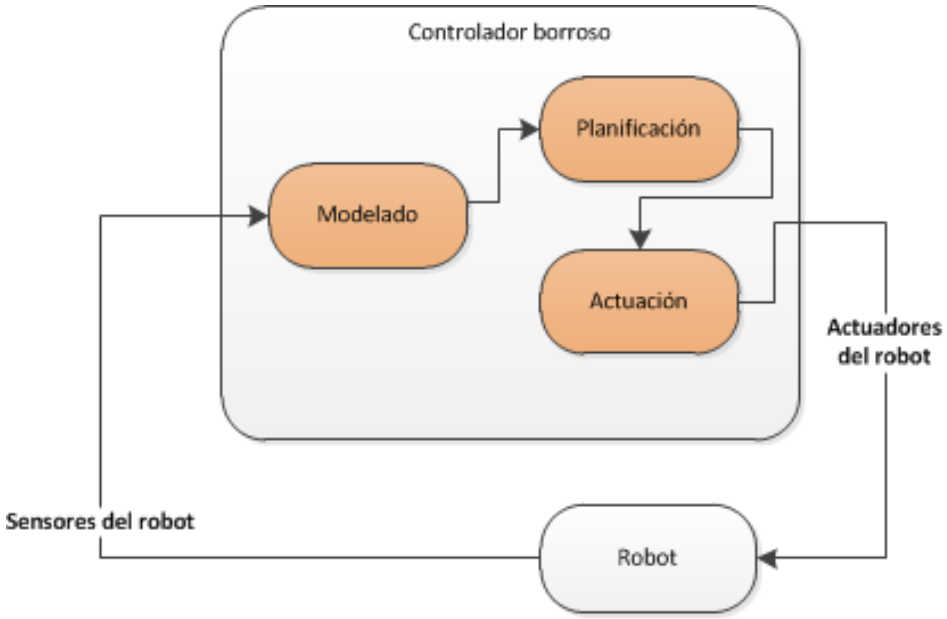

Figura 2.1: Diagrama que representa la arquitectura jerárquica.

- *Arquitecturas reactivas:* descomponen las funciones en módulos complementarios que son ejecutados en paralelo, intentando que cada una de ellas resuelva parte de las tareas. Cada uno de los módulos toma como entrada los datos de los sensores y genera una respuesta. Una vez integradas todas las respuestas se genera una orden que es transmitida a los efectores. A pesar de que el tiempo de reacción del robot disminuye de forma considerable respecto a las arquitecturas jerárquicas, el razonamiento de alto nivel es mucho mas limitado. Este tipo de arquitectura se representa en la figura 2.2.

- *Arquitecturas híbridas:* es la fusión de las dos aproximaciones anteriores que pretende obtener los beneficios de cada una de ellas. Se consigue utilizando un sistema de doble capa que se describe a continuación:

  - *Capa planificadora:* interviene a alto nivel, utilizando el conocimiento global del entorno para trazar rutas generales y obtener mapas.
  - *Capa reactiva:* interactúa con el entorno en tiempo real del modo descrito anteriormente. Permite una capacidad de respuesta rápida para las decisiones que se toman a más bajo nivel. No todos los componentes de la capa reactiva están activos siempre, sino que es la capa planificadora la que decide cuáles de ellos son utilizados en cada momento.

  En este caso, el ciclo de ejecución define una utilización de la etapa planificadora en primer lugar, donde se produce un análisis a alto nivel del entorno, y posteriormente se utiliza la capa reactiva para tomar las decisiones y ejecutar una orden de control sobre el robot. Sólo la capa reactiva funciona en tiempo real para tener un tiempo de reacción adecuado, la capa de planificación está desacoplada para evitar retardos, como se muestra en la figura 2.3.

En el ámbito de la robótica móvil hay numerosas fuentes de incertidumbre que pueden afectar al conocimiento que el robot tiene sobre el entorno en el que está situado. Esta incertidumbre proviene de

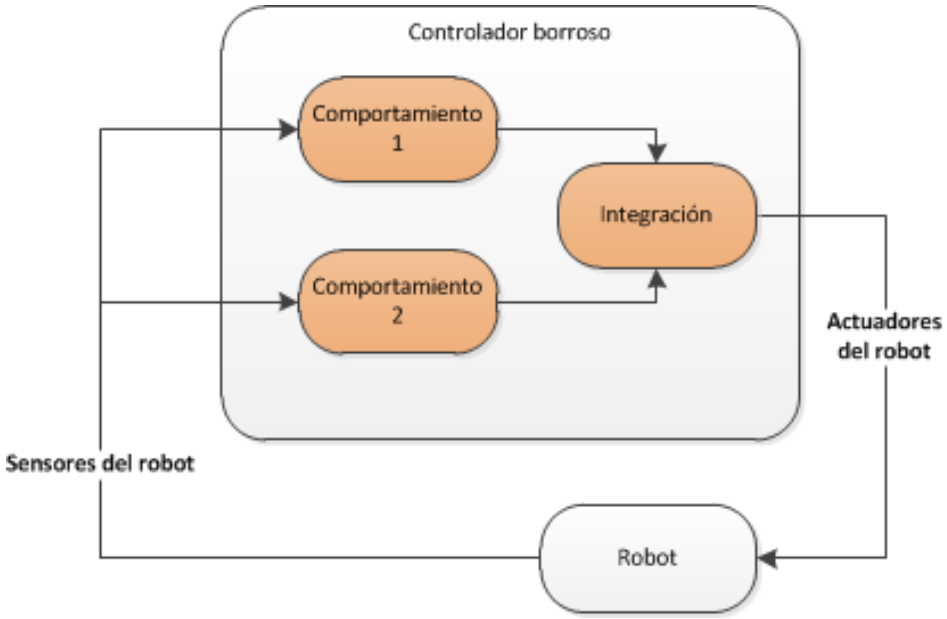

Figura 2.2: Diagrama que representa la arquitectura reactiva.

diferentes fuentes: la primera de ellas es el hecho de que se encuentra en un entorno no estructurado, esto es, no se tiene información completa y precisa de lo que sucede en el entorno del robot en tiempo real. En un entorno estructurado se conoce previamente todo el entorno, y no existen condiciones que lo alteren, por lo que las decisiones se pueden tomar sólo en base a esa información; sin embargo en un entorno no estructurado, aunque se tenga dicha información previa, esta puede no ser completamente precisa, ya que puede haber factores que la alteren, como la presencia de objetos no mapeados o elementos móviles dentro del entorno, que también hay que tener en cuenta para la toma de decisiones de control.

Otra de las fuentes de incertidumbre viene dada por la falta de fiabilidad de los sensores con los que está equipado el robot. Todos los sensores tienen un cierto margen de error en sus mediciones, es lo que se denomina *ruido de la medida*. Finalmente, al igual que los sensores de los robots, los efectores, o actuadores, también pueden introducir incertidumbre al no realizar con absoluta precisión la orden que se pretende ejecutar.

En los entornos con incertidumbre, como es el caso de la mayor parte de entornos reales en los que opera un robot móvil, la utilización de un control borroso resulta adecuada, pues permite gestionar la imprecisión y el desconocimiento asociado a dichos entornos. Además, su desarrollo no tiene que que tener en cuenta modelos matemáticos complejos del sistema a controlar, ya que la complejidad del proceso de inferencia inferencia de la información va incluido en la propia estructura de las reglas.

2.2. Lógica borrosa

En lógica clásica, las decisiones se toman en función de valores si/no que están representados mediante un valor *booleano*. Si se tiene en cuenta la incertidumbre relacionada con los entornos de

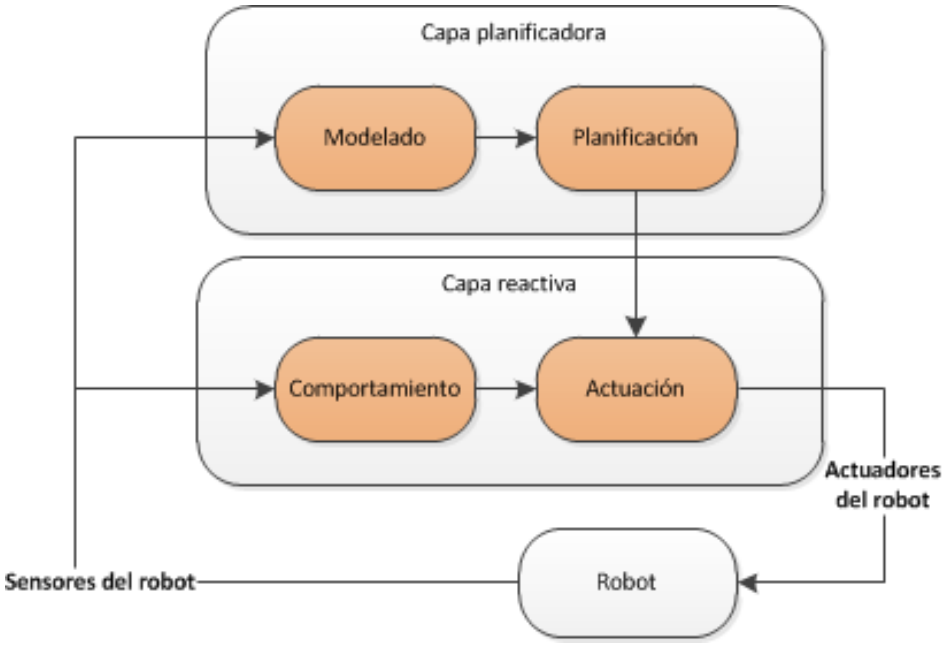

Figura 2.3: Diagrama que representa la arquitectura híbrida.

robótica móvil de la que se hablaba en la sección anterior, la lógica clásica no proporciona un mecanismo adecuado para la construcción de un controlador mediante el cual el robot pueda tomar las decisiones teniendo en cuenta la información de la que dispone.

La lógica borrosa, o difusa, proporciona una variación respecto a la lógica tradicional, donde se utiliza un modelo matemático que en lugar de representar la información mediante valores de cierto/falso, permita utilizar conceptos con una definición más flexible, como es el hecho de que un obstáculo se encuentre *cerca* o *lejos*, utilizando una serie de valores concretos que permitan definir los conceptos necesarios.

Una etiqueta borrosa es la definición formal, o matemática, de un concepto utilizado en una regla difusa, por ejemplo el concepto de que el objetivo se encuentre *ligeramente a la izquierda*. Por ejemplo, asociadas a la variable anterior se podrían tener las siguientes etiquetas borrosas: *a la izquierda*, *al frente* y *a la derecha* (figura 2.4).

Para cada conjunto difuso, existe asociada una función de pertenencia para sus elementos, que indican en qué medida el elemento forma parte de ese conjunto difuso. Las formas de las funciones de pertenencia más típicas son trapezoidal, lineal y curva. Cuando se evalúa si un valor corresponde o no a una determinada etiqueta borrosa, no se obtiene un valor si/no como en el caso de la lógica tradicional, sino un valor comprendido en el intervalo $[0, 1]$ que viene dado por la *función de pertenencia* de la etiqueta. Dentro de ese intervalo, los valores más altos corresponden a un mayor grado de pertenencia. Tan sólo se asocia un valor de 0 a aquellos valores que no tienen ninguna pertenencia a ella, y un valor de 1 a aquellos elementos cuya pertenencia coincida de modo absoluto con su definición. Esto se representa en la figura 2.5, donde se muestra un ejemplo de grado de pertenencia a una etiqueta borrosa. Si ésta se encuentra centrada en el valor 5 de la variable, éste corresponderá a un grado de pertenencia absoluta, es decir, 1. Para valores que se encuentran cercanos a éste, el grado de

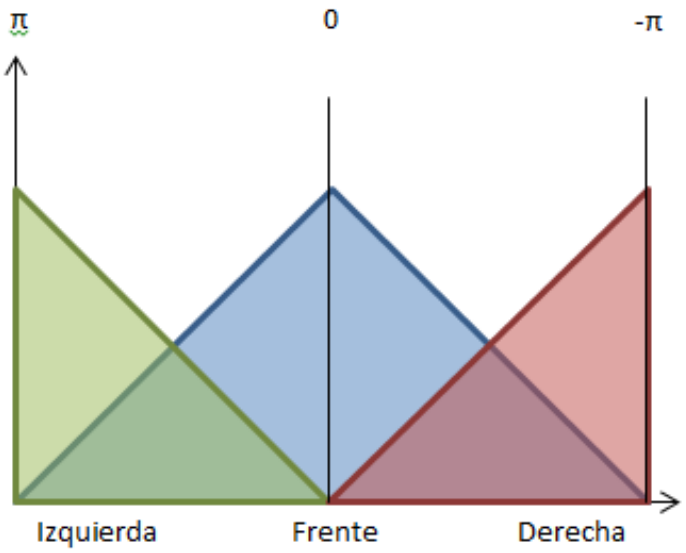

Figura 2.4: Universo de discurso para la variable *ángulo al objetivo*.

pertenencia es un número en el intervalo $[0, 1]$. En el caso de la figura, al valor de la variable 4 le corresponde una pertenencia a la regla borrosa de 0,5.

2.2.1. Controlador borroso

La navegación de un robot móvil se ha visto que es un problema que, además de su complejidad, introduce en varias de sus etapas incertidumbre que puede ser modelada con la lógica borrosa. El trabajo con un conjunto de reglas simple permite la realización de la operación de inferencia sobre los datos en un tiempo lo suficientemente corto como para garantizar su posible utilización en un problema que requiera de una respuesta en tiempo real.

Un controlador implementado mediante la utilización de lógica difusa, como el mostrado en la figura 2.6 tiene diferentes partes, que se detallan a continuación:

- *Borrosificador:* transforma un valor determinado en una etiqueta borrosa utilizando una función de borrosificación. No suele ser necesario en todos los problemas, si bien es imprescindible cuando se dan ciertas circunstancias, como tener varias fuentes de información para obtener el mismo valor, tener elementos que aporten incertidumbre al mismo, etc.

- *Base de reglas:* codifica el comportamiento del controlador mediante una serie de reglas borrosas. Existen dos tipos de reglas, las *Mamdani* y las *TSK* (*Takagi-Sugeno-Kang*). La diferencia entre las del primer tipo y las del segundo es que, en el caso de las *TSK*, el valor del consecuente es una función de las variables de entrada, mientras que las reglas de tipo *Mamdani* no disponen de esa restricción, por lo que el valor del consecuente puede ser especificado libremente.

  Las reglas contenidas en este elemento tienen la forma *if... else* que se puede observar en las

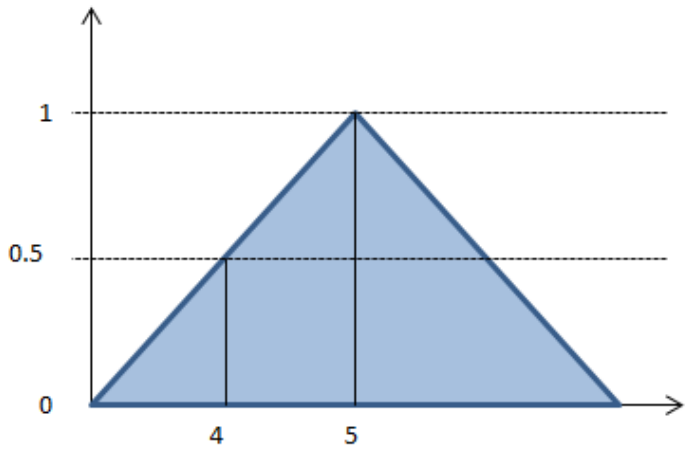

Figura 2.5: Grado de pertenencia de un valor a una etiqueta borrosa.

reglas de la lógica tradicional, expresada como:

$$X_1 \; es \; A_1 \; y \; X_2 \; es \; A_2 \; y \; ... \; y \; X_N \; es \; A_N \Longrightarrow Y_1 \; es \; B_1 \; y \; ... \; y \; X_M \; es \; Y_M \tag{2.1}$$

donde $X_1...X_N$ son las variables de entrada, $A_1...A_N$ son las etiquetas borrosas del antecedente, $Y_1...Y_M$ son las variables de salida, y $B_1...B_M$ son también etiquetas borrosas. Este es el elemento del controlador borroso que contiene el conocimiento aplicable para la resolución del problema de navegación de un robot móvil.

- *Motor de inferencia:* el mecanismo de inferencia es el encargado de mapear las entradas en salidas. El grado de disparo de una regla se calcula comparando la correspondencia que los valores de entrada tienen con las etiquetas borrosas correspondientes a los antecedentes de las reglas, con lo que se obtendrá un valor de correspondencia para cada una de las variables de entrada. La tasa de disparo de la regla será una función de todos esos valores, por ejemplo, el mínimo de todos ellos.

  Una vez conocidas las reglas que presentan un grado de disparo mayor que un cierto umbral, cada una de ellas determina un valor para las variables del consecuente. Todos ellos se tendrán en cuenta para el cálculo de la salida de la base de reglas.

- *Desborrosificación:* el proceso de desborrosificación consiste en obtener a partir de un conjunto borroso un valor numérico para cada una de las variables del consecuente.

El mecanismo de control borroso lo que permite es que, tomando como entradas una serie de valores, como el estado del robot o información sobre el entorno, ejecutar un comportamiento que se encuentra modelado en una base de conocimiento, para obtener como variables de salida una serie de valores que permitan al robot controlar sus dispositivos actuadores, como el motor de movimiento. En el caso del proyecto descrito en esta memoria, el comportamiento que debe estar modelado en la base de reglas debe permitir guiar el robot desde el punto actual donde se encuentra hasta el punto

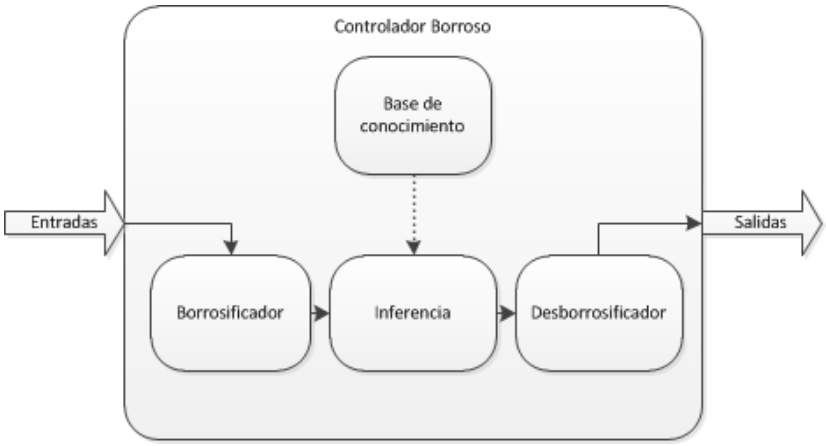

Figura 2.6: Estructura de un sistema de inferencia basado en lógica borrosa.

definido como objetivo, evitando todos los obstáculos intermedios, esto es, implementar la navegación del robot de forma autónoma.

Para que el controlador borroso funcione correctamente lo más importante es que el conocimiento modelado en la base de reglas sea adecuado a todas las posibles situaciones en las que pueda encontrarse el robot móvil. Si este conocimiento está representado de forma adecuada, el robot presentará una total autonomía para moverse entre dos puntos accesibles cualesquiera de un entorno no estructurado. El objetivo principal de este proyecto es que dicha base de conocimiento sea obtenida como resultado de un proceso automático de aprendizaje, que se encuentre implementado mediante un algoritmo evolutivo.

2.3. Algoritmos evolutivos

La idea subyacente en un algoritmo evolutivo se encuentra en la teoría darwiniana de la *Evolución de las Especies*, es decir, tener una población de individuos en un entorno donde los recursos son limitados, ponerlos en competición entre ellos para simular un mecanismo de selección natural, es decir, la supervivencia del mejor de ellos.

En términos computacionales, se define una función de *fitness* que asigna una puntuación a cada individuo de la población. El mejor individuo será aquel que consiga maximizar el valor de dicha función. Un algoritmo evolutivo está basado en iteraciones, de forma que en cada iteración se introduzcan cambios en la población, generando individuos nuevos a través de operaciones de cruce y mutación entre ellos. Esto produce un crecimiento en el número de individuos, que debe ser controlado eliminando algunos de ellos. Serán aquellos individuos que peor valor de puntuación, o *fitness*, presenten, los que no pasarán a la siguiente iteración del algoritmo. De esta forma se asegura la imitación del mecanismo de selección natural, donde sólo los individuos más fuertes sobreviven a los demás o, dicho de otra forma, los más débiles son eliminados progresivamente de la población.

Existen dos principales mecanismos que rigen el funcionamiento de los algoritmos evolutivos: por una parte los operadores de variación, que introducen diversidad suficiente en la población y facilitan

la creación de nuevos individuos; y por otra los operadores de selección, para asegurar que a lo largo de las distintas iteraciones del algoritmo la calidad de los individuos aumenta. Desde el punto de vista matemático, en realidad el proceso de evolución que se pretende modelar en este tipo de técnicas no es más que la resolución de un problema de optimización, donde lo que se busca es maximizar el valor de la función de puntuación que evalúa los diferentes individuos; si bien tal vez no se encuentra la mejor solución para dicha función, el objetivo es buscar una solución óptima. Desde el punto de vista evolutivo, lo que se pretende conseguir es una población de individuos que progresivamente se encuentre más adaptada al entorno que sus predecesores.

Todos los algoritmos evolutivos presentan una estructura general, detallada en el algoritmo 1. Si bien esto puede variar en función del problema particular a resolver, los principales componentes que los forman, así como sus procedimientos y operadores son los siguientes:

- Representación de los individuos
- Función de evaluación, o de *fitness*
- Población
- Mecanismo de inicialización
- Mecanismo de selección
- Operadores de mutación y de cruce para introducir diversidad en la población
- Mecanismo de reemplazamiento
- Condición de parada del algoritmo

Algoritmo 1 Estructura general de un algoritmo evolutivo

```
Inicialización de la población
Evaluación de todos los candidatos
while condición de terminación no satisfecha do
    Aplicar selección sobre la población
    Cruzar parejas de individuos padre
    Mutar los individuos
    Evaluación de todos los candidatos
    Selección de los individuos para la siguiente generación
end while
```

Para poder implementar un algoritmo evolutivo completamente funcional se deben definir todos los componentes anteriores. A continuación se detalla qué es cada uno de ellos y de qué manera participa en el proceso.

Representación de los individuos

Es el primer paso para la construcción de un algoritmo evolutivo. Consiste en elegir qué información contendrá cada individuo dentro de la población, así como la representación que se utilizará para trabajar con ella. Se definen los conceptos siguientes dentro de este ámbito:

- *Fenotipo:* es la representación de las soluciones en el contexto original del problema, esto es, sin realizar ninguna adaptación para trabajar con ellas dentro del algoritmo evolutivo.

- *Genotipo:* es la codificación, o también llamada representación, que el fenotipo tiene dentro del contexto del algoritmo. Esta es la forma que los diferentes operadores de manipulación de los individuos, como el proceso de cruce o de mutación, utilizarán durante la ejecución del algoritmo.

La correspondencia entre el genotipo y el fenotipo de un individuo no tiene por qué ser inmediata. Para ello se deberá disponer de un mecanismo que permita la decodificación de un genotipo, obteniendo su fenotipo equivalente. Dentro de este apartado cabe definir una serie de términos que se suelen utilizar en este contexto:

- *Cromosoma*: es equivalente a la representación codificada del individuo, esto es, su genotipo.

- *Solución candidata:* Se denomina de esta forma al individuo representado en el contexto del problema original que se pretende resolver, esto es, el fenotipo.

- *Gen:* cada una de las partes atómicas que componen un cromosoma.

Función de evaluación, o *fitness*

Su rol dentro del algoritmo es hacerse cargo de la representación de los requisitos a los que deben adaptarse los individuos de la población. Es la base del mecanismo de selección, y permite la introducción de mejoras a lo largo de las sucesivas iteraciones, definiendo además el término *mejora* para el contexto del problema. Técnicamente se encarga de asignar a los diferentes individuos un valor para definir su calidad como solución al problema que se pretende resolver.

Como el problema que los algoritmos evolutivos pretenden solventar de modo general es la búsqueda de un valor máximo para esta función, una vez encontrado el individuo que lo obtenga, será la mejor solución en el contexto del problema original.

Población

La población representa el conjunto de soluciones candidatas al problema que se pretende resolver. En general no es nada más que un conjunto de individuos, aunque está definida como la unidad evolutiva, en el sentido de que los individuos son unidades estáticas sobre los que no se producen cambios ni adaptaciones, todas estas operaciones se realizan sobre la población.

En este contexto es importante definir el concepto de *diversidad* de la población como el conjunto de diferentes soluciones que ésta presenta. Normalmente se puede conocer este valor de un modo sencillo, ya que se corresponde con el número de valores diferentes de *fitness* que presenta el conjunto de individuos que contiene. También se puede conocer realizando el recuento de los diferentes genotipos o fenotipos que existen en la población.

Mecanismo de selección

Se utiliza este mecanismo para seleccionar las parejas de individuos candidatas a cruzarse y mutarse, esto es, para convertirse en los padres de los individuos de la siguiente generación. En un algoritmo evolutivo esto es un proceso probabilístico, de modo que los individuos con mejor valor de *fitness* son los que tienen una mayor posibilidad de convertirse en padres, sin despreciar el hecho de que los individuos con peores puntuaciones también tienen probabilidad, aunque más reducida, de hacerlo.

Operador de mutación

Es un operador unario, esto es, donde sólo participa un individuo, que es aplicado sobre el genotipo y que devuelve una descendencia que no es más que el individuo del que proviene ligeramente modificado.

Normalmente se define como un operador en segundo plano que trabaja para añadir nuevos individuos a la población, y de este modo garantiza que no se produce una pérdida de diversidad, permite que el algoritmo evolutivo opere sobre un espacio de búsqueda más amplio, ya que con su aplicación pueden aparecer individuos en cualquier parte del espacio de búsqueda.

Operador de cruce

Es un operador binario que mezcla información genética de dos individuos para obtener dos nuevos como descendencia. Al igual que el operador de mutación, depende de decisiones tomadas en base a distribuciones de probabilidad, como qué parte de cada individuo participa en el proceso de recombinación de la información.

La teoría que existe detrás de la aplicación de este operador es que, mediante la combinación de dos individuos que presentan características deseables, es posible la obtención de descendencia que las combine a ambas; además, es posible que la combinación de características de los padres resulte en individuos de igual o peor calidad que ellos, si bien será el mecanismo de reemplazamiento de la población el que se encargue de decidir la supervivencia de unos individuos sobre otros para la siguiente iteración del algoritmo.

Operador de reemplazamiento

Este es un mecanismo similar al utilizado para la selección de las parejas candidatas para el cruce, puesto que también se basa en la puntuación de los individuos de la población para realizar operaciones sobre ellos.

En la mayor parte de los casos, la población de los algoritmos evolutivos mantiene un tamaño constante a través de las diferentes iteraciones del algoritmo, lo que significa que una vez que se genera la descendencia y los nuevos individuos son añadidos a la población actual, se debe realizar una selección para saber cuáles de ellos pasarán a formar parte de la población de la siguiente iteración.

Inicialización de la población

Para poder empezar a realizar el proceso que modela el algoritmo evolutivo es necesario contar con una población inicial sobre la que aplicar los diferentes operadores. Es muy frecuente que el método de generación de esta población inicial sea la creación de individuos aleatoriamente. Es posible utilizar heurísticas propias del problema a resolver para asegurar que la población inicial tiene, en promedio, un *fitness* alto.

Condición de terminación

La condición de terminación decide hasta qué momento el algoritmo evolutivo se sigue ejecutando para intentar mejorar la calidad de las soluciones que alberga la población. Existen numerosas condiciones típicas que son utilizadas en este tipo de algoritmos, entre las que destacan:

- Se ha alcanzado el máximo número de ciclos de CPU permitidos para la ejecución.
- El número total de evaluaciones realizadas ha superado el umbral máximo.
- La mejora del *fitness* de la población está por debajo de un umbral mínimo durante un número de iteraciones determinado.
- La diversidad de la población ha descendido por debajo de un valor crítico.

Normalmente se suele incluir como condición de terminación el hecho de alcanzar el valor máximo para la función de *fitness*, en el caso de que sea conocido. Esto se puede combinar con cualquier otra condición que sea adecuada al problema con el que se está tratando.

2.3.1. Modo de funcionamiento de un algoritmo evolutivo

Cada individuo de la población de un algoritmo evolutivo representa un punto determinado en el espacio de soluciones del problema con el que se está tratando. Durante el proceso de inicialización se generan individuos de modo aleatorio, de tal forma que se distribuyen por todo el espacio de búsqueda de soluciones. Después de una serie de iteraciones del algoritmo evolutivo esta distribución inicial cambia drásticamente y los individuos contenidos en la población abandonan las zonas de peor valor de *fitness* para escalar hacia los valores que se pueden determinar como máximos de la función de evaluación.

Es posible que dicho proceso de escalado alrededor de los máximos resulte en encontrar un punto de la función que, a pesar de ser un valor máximo para la misma, no es la mejor solución, es decir, que los individuos que contiene la población tiendan a concentrarse alrededor de un máximo local de la función de puntuación. Cuando esto sucede se produce una situación de *convergencia prematura* donde se pierde la diversidad de la población de una forma muy rápida. Este peligro se encuentra en todos los algoritmos evolutivos, aunque existen una serie de técnicas para prevenirlo.

Otro de los aspectos que hay que resaltar de los algoritmos evolutivos es que la curva característica que forma el valor de *fitness* de la población refleja un crecimiento muy rápido al principio del proceso,

aunque su crecimiento se va moderando a medida que se avanza en número de iteraciones. Estudiando la curva de evolución del *fitness* de la población se puede definir una condición de parada que tenga en cuenta esta situación. En el caso de utilizar heurísticas adecuadas al problema durante el proceso de inicialización de la población, se puede conseguir un valor de *fitness* inicial más alto, lo que hace que en k iteraciones se alcance un valor lo suficientemente bueno, que haga que el esfuerzo de computación adicional sea cuestionable en algunos problemas.

2.3.2. Resolución de problemas mediante algoritmos evolutivos

Desde una perspectiva global, la utilización de algoritmos evolutivos para la resolución de problemas de búsqueda de soluciones tiene como principal ventaja el poder utilizar esta aproximación para un rango de problemas muy elevado. Si bien es cierto que no se alcanza un rendimiento tan elevado como en el caso de utilizar algoritmos diseñados específicamente para el problema en cuestión, éstos tienen como principal desventaja el poder utilizarse en un abanico de problemas mucho más reducido. A pesar de que la búsqueda aleatoria se puede utilizar en la resolución de un gran número de problemas, al igual que los algoritmos evolutivos, es mucho más eficiente abordar el problema mediante estos últimos.

2.4. Sistemas borroso-evolutivos

En la sección 2.2.1 se ha explicado la conveniencia de utilizar un controlador borroso para la implementación del navegador de un robot móvil, introduciendo además la posibilidad de que la base de reglas utilizada por dicho controlador fuese el resultado de un proceso de aprendizaje llevado a cabo a través de un algoritmo evolutivo. Esta aproximación al problema resulta más viable que diseñar un controlador borroso únicamente partiendo de conocimiento experto. La utilización de un algoritmo evolutivo para llevar a cabo la tarea de aprendizaje de una base de conocimiento borroso se fundamenta en que la flexibilidad de la representación de las soluciones en este tipo de aproximación es muy alta. El aprendizaje de bases de conocimiento borroso a través de un algoritmo evolutivo presenta diferentes aproximaciones:

- *Pittsburgh:* un individuo representa una base de conocimiento completa, por lo que su tamaño dependerá del numero de reglas que contenga cada base de conocimiento. Es una metodología con un coste computacional muy elevado, ya que la evaluación de una base de conocimiento no es una tarea sencilla, y en cada iteración del algoritmo se producen un gran número de evaluaciones.

- *Michigan:* cada individuo representa una regla, por lo que su tamaño es siempre el mismo. La población entera del algoritmo representa la base de conocimiento. Las reglas evolucionan a través de las diferentes iteraciones del algoritmo debido a la interacción con el entorno y el aprendizaje por refuerzo. Normalmente tiene una enorme complejidad en la distribución de la recompensa.

- *IRL (Iterative Rule Learning):* es igual que la aproximación *Michigan*, pues cada individuo representa una regla. Sin embargo esta aproximación introduce la restricción de que en el conjunto

de iteraciones del algoritmo sólo se entrena una regla, no todas las que forman la base de conocimiento. Cuando finaliza cada conjunto de iteraciones, la mejor regla se añade a la base de conocimiento final.

- *GCCL (Genetic Cooperative-Competitive Learning):* al igual que las dos aproximaciones anteriores, cada individuo representa una única regla, con la diferencia de que éstas evolucionan en conjunto, mediante cooperación, compitiendo entre sí para obtener el mejor valor de *fitness*. Es importante en este mecanismo mantener la diversidad de la población.

2.5. Objetivos del proyecto

Durante el prefacio se han enumerado una serie de objetivos preliminares. Sin embargo, una vez que se ha descrito el ámbito del proyecto, es posible especificar con un poco más de precisión qué es lo que se busca conseguir con la realización de este proyecto:

- Implementación del algoritmo de aprendizaje que permita la generación de una base de reglas, que será utilizada por el controlador del robot para la toma de decisiones de control.
- Construcción de una herramienta que permita la generación del conjunto de ejemplos de entrenamiento y que pueda ser utilizado de forma transparente por el algoritmo de aprendizaje.
- Implementación de un controlador borroso que realice la inferencia desde la base de reglas que se ha obtenido en el paso de aprendizaje y que obtenga los comandos de control necesarios para resolver el problema de la navegación autónoma.
- Realizar pruebas de simulación mediante la herramienta Player/Stage para probar tanto el dispositivo controlador como la base de conocimiento aprendida, y comprobar que el comportamiento que se está modelando es el que se desea para realizar la navegación.

Capítulo 3

Algoritmo evolutivo

A lo largo del presente capítulo se realizará una descripción pormenorizada de la aproximación que se seguirá para la construcción del algoritmo de aprendizaje, explicando cada uno de los pasos que se llevarán a cabo durante la ejecución de una iteración. Para cada operador genético se realizará una breve introducción a sus funciones dentro del algoritmo, para posteriormente ofrecer una explicación detallada del mismo.

3.1. Contextualización

Un robot móvil situado en un cierto entorno debe disponer de cierta información a la hora de tomar decisiones sobre cuál va ser su próximo movimiento. Típicamente, para que no se produzca una colisión con algún obstáculo se debe conocer cuál es la distancia a ellos, para poder hacerlo, el robot debe llevar incorporado algún tipo de sensor que proporcione esa información. Existen diferentes tipos de sensores, aunque el utilizado en este caso será un dispositivo láser. Este dispositivo realiza un barrido de 180º, tomando una medida cada 0.5º, en total 361 medidas por dispositivo utilizado. En principio tan sólo se tomarán las medidas correspondientes a la parte delantera del robot, por lo que sólo será necesaria la información de un medidor láser. A pesar de que el robot está equipado con dos dispositivos de este tipo, el trasero no se utilizará.

Si se quisieran utilizar independientemente todas las mediciones del láser del robot se tendría un número muy elevado de antecedentes en las reglas de la base de conocimiento. En realidad no es relevante conservar la información de las mediciones a tan bajo nivel. Los obstáculos con los que se puede encontrar el robot, y que se pretenden evitar, serán de un tamaño lo suficientemente elevado como para hacer que una gran cantidad de mediciones sean, sino iguales, muy similares. Como se puede ver, tan sólo se especifican tres distancias: la frontal, y las dos laterales. Para poder integrar todas las mediciones que provienen del láser en tres variables se agrupan por sectores. De esta forma se definen:

- *Sector izquierdo:* es el que representa todas las mediciones comprendidas en el intervalo $[-\frac{\pi}{2}, -\frac{\pi}{6})$.

- *Sector frontal:* representa la parte frontal del láser y comprende el intervalo de medidas en el ángulo $[-\frac{\pi}{6}, \frac{\pi}{6})$.

- *Sector derecho:* representa las distancias del lado derecho, en el ángulo comprendido entre $[\frac{\pi}{6}, \frac{\pi}{2}]$.

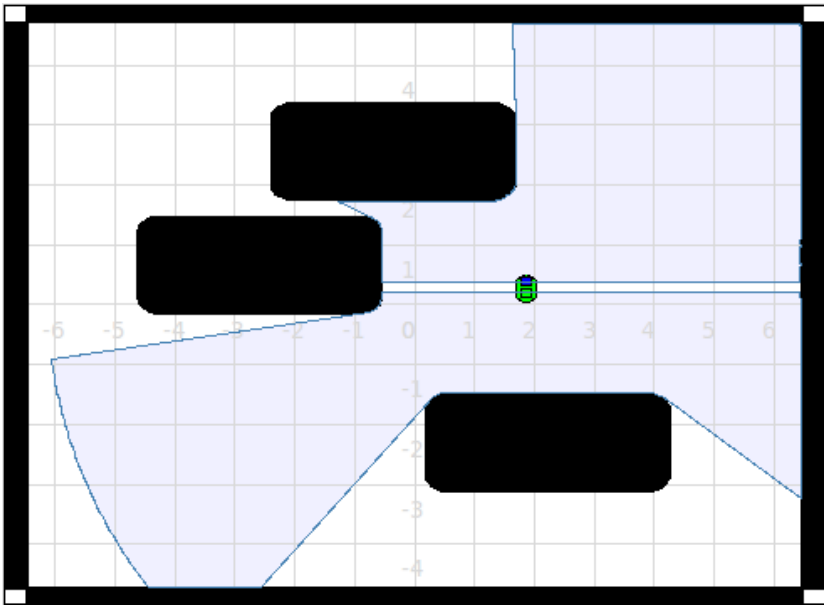

Figura 3.1: Ejemplo de entorno simulado en el que se mueve el robot.

Para mayor conveniencia del proceso de aprendizaje la distribución inicial de las distancias de los láser en sectores podría verse modificada. Es necesario tener en cuenta esta circunstancia a la hora de diseñar el algoritmo, para evitar posibles modificaciones posteriores de código.

Debe evitarse que el robot pueda colisionar con algún obstáculo. Por ello el controlador debe detectar cuándo se está demasiado cerca de alguno, y detener el robot para evitar un daño innecesario. Esto se producirá mediante la definición de una distancia de seguridad, una circunferencia en la que ningún objeto podrá situarse para que el robot prosiga su normal funcionamiento.

3.2. Aproximación

El objetivo final de este proyecto es la realización de un algoritmo evolutivo que permita el aprendizaje automático de un controlador basado en reglas borrosas utilizado para la navegación de un robot móvil. Cuando se ejecute el algoritmo de aprendizaje, éste deberá proporcionar como salida la mejor de ellas, que será la utilizada para el control del robot.

El proceso de aprendizaje necesita de un conjunto de ejemplos para guiar el proceso de aprendizaje. Este conjunto de ejemplos puede proceder de diferentes fuentes, en este caso se considerarán dos posibles opciones:

- Ejemplos que provienen de una ruta de ejemplo realizada previamente: hay que tener especial cuidado con este tipos de datos, ya que a la vez que introducen conocimiento experto en el proceso de aprendizaje, también pueden ser una posible fuente de error cuando la calidad de los ejemplos no es lo suficientemente elevada para aprender un comportamiento. En este sentido, es necesario comprobar que la ruta, o rutas, de ejemplo, reflejen de modo suficientemente preciso las posibles situaciones a las que el robot debe dar respuesta cuando se ejecute la base de conocimiento en su controlador.

  El hecho de que puedan existir situaciones no reflejadas en el conjunto de ejemplos implica que el comportamiento no las aprenderá, por lo que llegada esa situación el control fallará; por tanto, es necesario asegurar la calidad del conjunto de ejemplos de entrenamiento para garantizar que, tras la ejecución del algoritmo, el comportamiento aprendido responderá adecuadamente a todas las situaciones en las que el robot se encuentre.

- Ejemplos generados sintéticamente: con esta aproximación se pretende subsanar la posible falta de diversidad de ejemplos cuando se generan a partir de las rutas de ejemplo. De esta forma, se puede garantizar que todas las situaciones que se pueden dar según las variables de entrada definidas para la base de conocimiento son cubiertas por al menos un ejemplo que modele cuál debe ser el comportamiento del robot en ese caso. De igual forma que los antecedentes de los ejemplos se generan de modo sintético, también debe hacerse para los consecuentes. Todos los posibles ejemplos resultan de la aplicación del producto cartesiano de todas las combinaciones posibles entre los antecedentes existentes.

  La generación de los consecuentes es más compleja, pues hay que tener en cuenta la información que está contenida en el antecedente del ejemplo, y evaluar la acción resultante. Al igual que en el caso de los antecedentes, también hay una lista de posibles consecuentes, o acciones, que el robot puede tomar en una situación determinada. El mecanismo para asignar uno de ellos a un ejemplo pasa por evaluar cada uno de ellos a través de la misma función de evaluación que la utilizada para el proceso de aprendizaje, y asignarle el que mayor puntuación obtenga. Este método tiene como inconveniente que el número de ejemplos generados puede ser excesivamente alto, lo cual supondría un coste computacional excesivo para la ejecución del algoritmo, esto se puede calcular de modo sencillo mediante la siguiente expresión:

$$N = \prod_{i=1}^{A} (a_n + 1) \tag{3.1}$$

  es decir, se calcula como el producto de los intervalos de todas las variables de entrada incrementados en una unidad. La razón de este incremento es que si una variable está definida en N niveles, en realidad son necesarios $N + 1$ valores para poder definirlos, tal y como se refleja en la figura 3.2.

3.2.1. Generación de ejemplos a partir de los datos de entrada

Para poder realizar la transformación entre unos y otros se requieren una serie de operaciones matemáticas que se describen a continuación:

En cuanto a las distancias, el láser se divide en un número determinado de sectores, en este caso tres, cuyo principio y final se encuentra definido. Para cada medición se calcula el ángulo al que

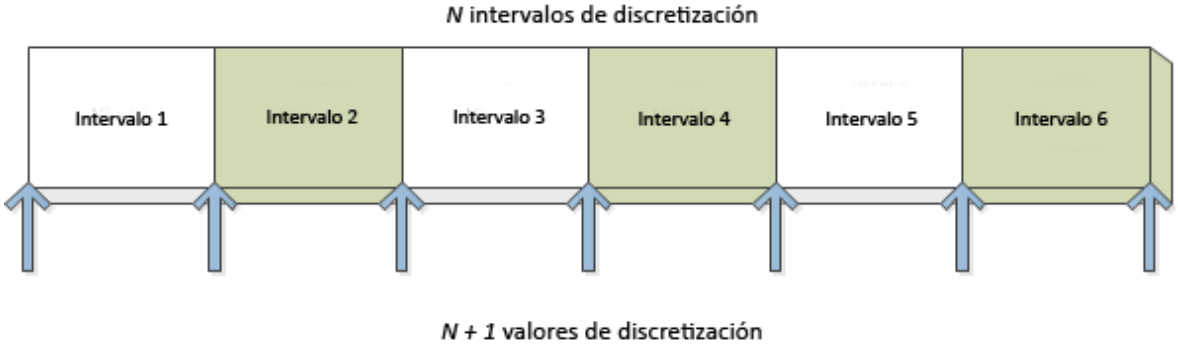

Figura 3.2: Intervalos y valores de discretización para una variable.

corresponde, $\phi = \phi_1 \cdot i$, donde ϕ_1 representa la resolución de las medidas en el láser, e i el número de medición que se está procesando. Una vez obtenido el ángulo al que corresponde una determinada medición, se le asigna el sector al que corresponde. La medición que se obtiene para cada sensor es el mínimo de las mediciones individuales que corresponden a él.

Para el cálculo de la distancia y el ángulo al objetivo se sigue la siguiente aproximación. Hay que tener en cuenta que el robot, sea cual sea su posición absoluta, siempre se va a encontrar en el instante inicial en el punto $P_0(x_0, y_0, \phi_0) = (0, 0, 0)$, de modo que la posición se conoce siempre de modo relativo a la posición inicial. El punto objetivo se conoce de antemano pues está definido por el usuario de la aplicación, y se define de igual manera, como $P_F(x_F, y_F, \phi_F)$. El cálculo del ángulo al objetivo, teniendo en cuenta la orientación del robot, se realiza como se detalla en el algoritmo 2.

Algoritmo 2 Cálculo del ángulo objetivo en el intérprete de Player/Stage

Require: $p_\theta \Leftarrow$ robot angle
Require: $p_x \Leftarrow$ position X
Require: $p_y \Leftarrow$ position Y
  if $p_y = 0$ **then**
    if $p_x > 0$ **then**
      $\alpha = \pi/2$
    else if $p_x < 0$ **then**
      $\alpha = -\pi/2$
    else if $p_x = 0$ **then**
      $\alpha = 0$
    end if
  else
    $\alpha = atan(p_y/p_x)$
  end if
  return $\theta - \alpha$

Finalmente, el cálculo de la distancia del robot al objetivo se lleva a cabo teniendo en cuenta que el simulador asigna al robot la posición del origen de coordenadas en el instante inicial, sea cual sea su posición absoluta dentro del mapa. Esto significa que todas las posiciones serán relativas a ese punto

de inicio del robot. Por tanto para el cálculo de la distancia de la posición actual del robot al punto objetivo se aplica la siguiente fórmula:

$$d_{obj} = \sqrt{(x_F - p_x)^2 + (y_F - p_y)^2} \tag{3.2}$$

donde x_F e y_F representan las coordenadas x e y del punto objetivo, y p_x y p_y son sus homólogos de la posición actual del robot.

El resto de los datos, como la velocidad lineal y angular, se toman de los datos leídos de modo directo, aunque con una peculiaridad. Los datos del antecedente se toman de la medición actual, en el instante t, y los datos del consecuente para ese mismo ejemplo se toman de la medición en el instante siguiente, $t + 1$.

3.2.2. Reducción del número de ejemplos

Para evitar trabajar con un número demasiado alto de ejemplos, y que esto alargue la ejecución del algoritmo evolutivo, se puede utilizar un algoritmo de reducción de ejemplos [3], cuya finalidad es detectar cuáles de ellos son redundantes, o demasiado cercanos a otros, y poder eliminarlos del conjunto. En este proyecto se ha pensado en la aplicación de dos algoritmos de reducción de ejemplos distintos, que se describen a continuación.

Algoritmo IB2

Algoritmo 3 Descripción del algoritmo de reducción de ejemplos IB2

```
for i in ejemplos do
  for j in ejemplosSeleccionados do
    s[j] = similaridad(i, j)
  end for
  jMax = max(s)
  if clase(i) != clase(jMax) then
    add(i, ejemplosSeleccionados)
  end if
end for
```

Se ha definido la función de *similaridad(i, j)* como:

$$similaridad(i, j) = -\sqrt{\sum_{n=1}^{n} f(i_n, j_n)} \tag{3.3}$$

y donde la función $f(i_n, j_n)$ se define de la siguiente forma, considerando *range* como el rango máximo que puede tomar la variable que se está procesando:

$$f(i_n, j_n) = \left(\frac{i_n - j_n}{range}\right)^2 \tag{3.4}$$

Una vez que finaliza la ejecución, el conjunto de ejemplos seleccionados son los que representan al conjunto total de los ejemplos, pues se han eliminado los más similares, quedándose con sólo los más representativos.

En el algoritmo se puede ver que se comparan las clases de los ejemplos para ver cuáles se añaden al conjunto de los seleccionados. Se dice que dos ejemplos son de la misma clase si tienen los mismos valores para sus consecuentes.

Agregación de intervalos de discretización

Cada ejemplo está formado por un número determinado de variables de entrada y de salida. El usuario puede especificar el nivel de discretización de cada variable, de modo que cada una de ellas está representada por unos valores, de la forma en la que se describe en la figura 3.3. Inicialmente las variables están definidas divididas en intervalos definidos por sus valores extremos.

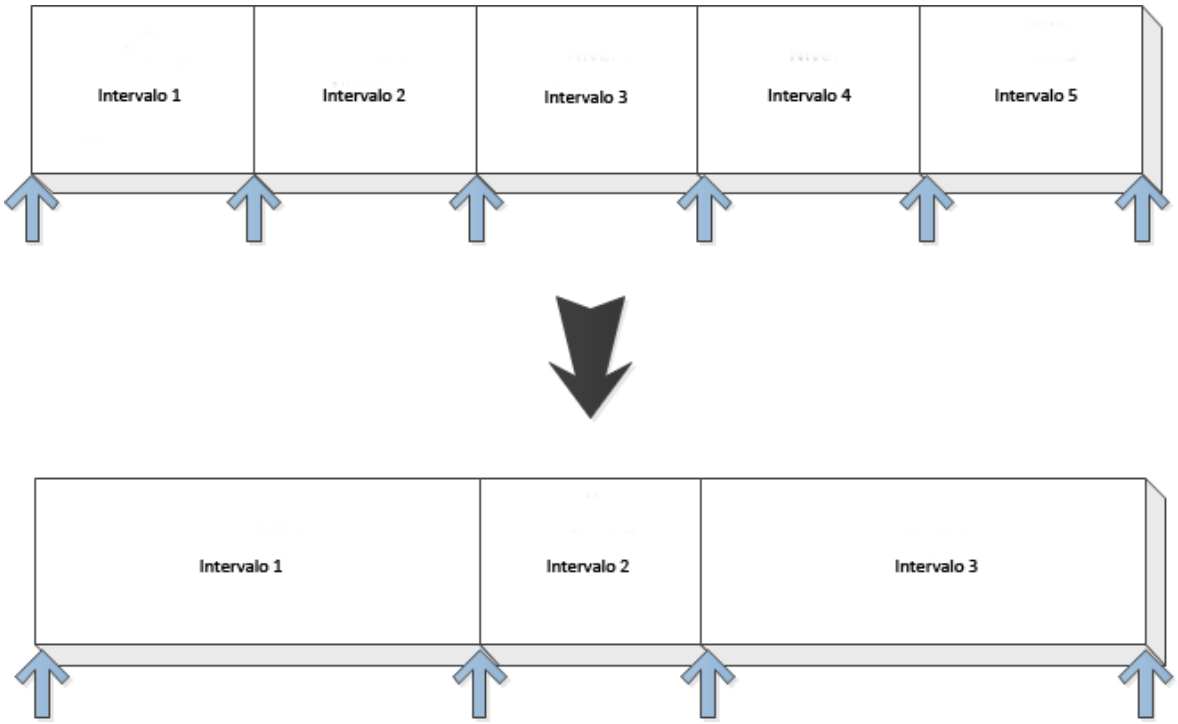

Figura 3.3: Discretización de las variables de los antecedentes de las reglas.

Para proceder a la reducción del número de ejemplos, se construye un vector de valores con el número de posiciones igual al número de posibles consecuentes que se generan con los valores de discretización introducidos. La construcción de este vector se realiza para poder analizar si dos intervalos de la misma variable de entrada son equivalentes. Su construcción se realiza como se detalla en el algoritmo 4.

Es necesaria la construcción de un vector de este tipo por cada intervalo que haya en cada variable de entrada. De esta forma, si N_0 es el numero de intervalos iniciales de la variable, se tendrá un vector con el mismo número de posiciones, donde el valor de cada una de ellas se obtiene directamente del vector de salida del algoritmo anterior, V, calculando:

$$intervalo[j] = \sum_{i=0}^{N_0} V_j[i] \tag{3.5}$$

Algoritmo 4 Construcción del vector para el análisis de intervalos de discretización de las variables

```
for i in ejemplos do
  while n ≤ N_conseq do
    if n.conseq = i.conseq then
      p = n
      break
    end if
    n++;
  end while
  V[p]++;
end for
```

Una vez llegados a este paso, se obtienen tantos vectores como variables de entrada definidas. El siguiente paso a realizar es el análisis de estos resultados para poder realizar modificaciones sobre ellos. De modo automático, se analizan las posiciones adyacentes del vector, marcando aquellas que tienen el mismo valor. Una vez finalizado, se crearán los nuevos niveles de las variables, donde se conservaran las posiciones de inicio y de fin de cada bloque, considerando que las posiciones iguales entre sí pertenecen al mismo bloque, tal y como aparece reflejado en la figura 3.3 . Cuando se hayan obtenido todos los nuevos niveles de las variables, el nuevo conjunto de ejemplos será un subconjunto del anterior, donde se eliminen los que tienen alguna correspondencia con algún nivel que se ha suprimido durante el proceso de reducción.

3.3. Representación de los individuos

Como se ha visto en la sección 2.4, un sistema borroso-evolutivo puede tener varias aproximaciones, en las cuales un individuo de la población puede representar tanto una regla como una base de conocimiento completa. En este proyecto se utilizará la aproximación *Pittsburgh* para construir este tipo de sistema, por lo que un individuo representará a una base de reglas completa.

El primer paso para el diseño del algoritmo evolutivo consiste en seleccionar la representación que tendrán los individuos, tanto su fenotipo como su genotipo.

La solución que debe aportar el algoritmo evolutivo es una base de reglas borrosas, por lo que cada individuo codificará una base de conocimiento entera y, por tanto, cada uno de los individuos será una posible solución.

Fenotipo

El fenotipo de un individuo representa una solución candidata expresada en términos del problema original. En este caso, el problema original se define como la obtención de una base de reglas borrosas para la implementación de un controlador que permita la navegación autónoma de un robot, evitando todos los obstáculos que se encuentren entre el punto actual y el objetivo, por lo que la solución al problema, y por tanto el fenotipo de un individuo, estará formado por una base de reglas expresada

de la siguiente forma:

$$\begin{aligned} &R_1 : antecedente_1 \Longrightarrow consecuente_1 \\ &R_2 : antecedente_2 \Longrightarrow consecuente_2 \\ &\vdots \\ &R_N : antecedente_N \Longrightarrow consecuente_N \end{aligned} \tag{3.6}$$

Cada una de las reglas de la base de conocimiento anterior está formada por una serie de valores que representan el antecedente, esto es, las entradas de la base de reglas; y otros valores que conforman el consecuente de la misma. Para abordar el problema se utilizarán reglas de tipo *Mamdani*, que se han detallado en la sección 2.4.

El antecedente de la base de reglas estará formado por un conjunto de variables que provengan del conjunto de sensores del robot, esto es: información relevante sobre el entorno en el que se encuentra, y sobre el propio estado del robot. Para este proyecto se tendrá en cuenta el siguiente conjunto de variables para el antecedente:

- Distancia frontal
- Distancia derecha
- Distancia izquierda
- Distancia al objetivo
- Ángulo al objetivo
- Velocidad del robot

La distancia trasera no se considera como una entrada para el controlador borroso, ya que el movimiento es siempre hacia adelante, con una velocidad lineal positiva, y aunque se produzca un giro, la velocidad angular máxima no puede ser, por las propias limitaciones de los efectores, tan alta como para obligar a tener en cuenta la distancia trasera del robot.

Como la funcionalidad que se pretende implementar mediante el control es de dotar al robot de la capacidad para llegar desde su posición actual al punto objetivo evitando los obstáculos intermedios, es importante saber la distancia a la que se encuentran los obstáculos, así como la posición del punto objetivo respecto a la posición actual del robot, y la velocidad a la que se está moviendo.

En un robot, los sensores ofrecen información sobre el entorno en el que se sitúa el robot con mucha precisión. Un sensor láser típico toma medidas de distancia a los obstáculos cada 0.5°, por lo que si el rango de medición abarca 180°, el número de mediciones totales será de 361, sin duda alguna un número demasiado elevado para ser tomado en bruto como entradas para un controlador borroso. Tener un número tan elevado de variables en el antecedente implicaría un procesamiento de las reglas mucho más complicado, además de la dificultad de obtener una solución óptima en un tiempo razonable. El formato de la base de reglas resultaría también más complicado de procesar en el controlador, y el nivel de detalle con el que se describiría la solución dificultaría su idoneidad para todas las situaciones, debido a la especificidad de la situación en la que se dispararía cada una de las reglas que la componen.

Por este motivo, se debe realizar una transformación de los datos que ofrecen los sensores del robot para concentrar dicha información en las variables de entrada que se han definido para el antecedente de las reglas. El hecho de integrar la información en un conjunto más reducido de variables de entrada permite su manejo con mayor comodidad, al tener que trabajar con una menor cantidad de datos de modo simultáneo. Por otra parte la definición de las reglas se simplifica, permitiendo su aplicación en un mayor número de situaciones, haciéndolas más generales. Finalmente, hay que tener en cuenta que las medidas de las distancias adyacentes que se obtienen en bruto de los sensores láser del robot serán muy similares, y no aportarán un mejor conocimiento del entorno, ya que muchas de ellas serán redundantes.

Los datos en bruto recogidos por los sensores del robot no pueden ser utilizados directamente por el controlador borroso, puesto que su formato no coincide con el del antecedente de la base de reglas que éste utiliza. Para poder llevar a cabo la transformación al formato adecuado es necesario realizar un conjunto de operaciones matemáticas que se detallarán en la sección 6.3.1.

El consecuente de las reglas debe estar especificado por lo que se desea que devuelva el controlador, es decir, por la información que está contenida en los comandos de control que se le envían al robot. En un robot móvil que trabaja sobre dos dimensiones, típicamente, se trabaja con:

- Velocidad lineal
- Velocidad angular

Este control es suficiente para determinar el siguiente punto a donde se desea que se desplace el robot, y se determina de modo automático en base a la correspondencia de la información de las entradas con las reglas que contiene la base de conocimiento. De esta forma, cada una de las reglas queda definida por 6 antecedentes y 2 consecuentes.

Genotipo

El genotipo es la expresión codificada de la solución que se expresa en el fenotipo. Para realizar la conversión hay que tener en cuenta que debe existir una correspondencia unívoca entre ambos, de modo que la codificación no deje lugar a ambigüedades.

El proceso de codificación de la base de conocimiento que se recoge en el fenotipo debe tener como salida un conjunto de reglas borrosas cuya representación sea lo más simple posible, para permitir un trabajo eficiente por parte de los operadores genéticos sobre estos datos.

En este proyecto, las etiquetas borrosas que se utilizarán para formar parte de las reglas que componen la base de conocimiento serán definidas *a priori*, en el proceso de inicialización del algoritmo. Cada variable de entrada que forme parte del antecedente tendrá una *granularidad* determinada por un número entero entre 1 y un límite máximo especificado previamente. Esto representa el número de intervalos de la variable de entrada, de la forma en la que se definió en la figura 3.2. Una variable con un determinado valor de granularidad tendrá un número de etiquetas exactamente igual a éste, por lo que cada una de ellas se puede identificar de modo unívoco especificando la variable a la que pertenece, su granularidad, y el número de etiqueta dentro de dicha granularidad.

Una regla de la base de conocimiento está determinada por la siguiente forma:

$$R:\ d_{frontal}\ d_{dcha}\ d_{izda}\ d_{obj}\ \theta_{obj}\ V_{lineal} \Longrightarrow V_{robot}\ \omega_{robot} \tag{3.7}$$

Cada una de las variables de entrada y de salida ocupa una posición dentro de la regla. Dado que su formato es constante y conocido de antemano, una variable queda identificada por su posición. De este modo, sabiendo que una etiqueta borrosa se puede definir de modo unívoco por el par de valores $< granularidad,\ etiqueta >$, ya se tienen todos los elementos necesarios para la codificación de una regla borrosa, que queda definida de la siguiente forma:

$$R_{codificada} : [G_1,\ E_1]\ ...\ [G_n,\ E_n] \Longrightarrow [G_1,\ E_1]\ ...\ [G_m, E_m] \tag{3.8}$$

donde G representa el valor de granularidad, y E el número de etiqueta dentro de éste.

3.4. Inicialización de la población

La inicialización de la población tiene como objetivo la generación de soluciones repartidas a lo largo de todo el espacio de búsqueda. Es interesante incluir en esta etapa conocimiento experto que permita tener un valor alto del *fitness* de la población inicial, y así reducir el tiempo que se tarda en encontrar la mejor solución al problema, esto es, el individuo que maximiza la función de evaluación.

Para garantizar el cumplimiento de estas dos circunstancias, se utilizará un método de inicialización mixto, que combine la generación de individuos de forma completamente aleatoria, y la introducción de conocimiento experto, que se realizará mediante el método propuesto por Wang y Mendel [10]. El esquema general que se seguirá para la inicialización de la población, combinando los dos métodos de inicialización de un individuo, se detalla en el algoritmo 5.

Algoritmo 5 Inicialización de la población del algoritmo evolutivo

```
while tamanoPoblacion < N do
    generar granularidades
    generar conjunto de antecedentes para esas granularidades
    inicialización aleatoria con los antecedentes generados
    inicialización Wang y Mendel con los antecedentes generados
    añadir individuos a la población
end while
```

Como se puede comprobar, por cada iteración del operador de inicialización se generan dos individuos que son añadidos a la población, uno que se ha creado mediante el método aleatorio, y otro que se ha creado mediante el de Wang y Mendel. Ambos se crean partiendo del mismo conjunto de antecedentes, que se crea previamente a la aplicación de estos métodos.

Dado un conjunto de granularidades para las variables de entrada, debe generarse el conjunto de todos los posibles antecedentes, de modo que contengan todas las posibles combinaciones de las etiquetas borrosas que pueden tener las variables de entrada de la base de reglas. Este proceso se puede realizar incrementalmente, de tal forma que en cada iteración se añada una nueva variable al conjunto de antecedentes, tomando el conjunto de la iteración anterior y generando todas las posibles

reglas que se puedan formar al añadir una nueva variable, de la forma en la que se detalla en algoritmo 6.

Algoritmo 6 Generación de todos los antecedentes posibles para un conjunto de granularidades

```
Require: antecedentes ⟸conjunto de antecedentes
  for i in tamanoAntecedente do
    if antecedentes está vacío then
      for j in granularidad_i do
        añadir valor [granularidad_i, etiqueta_j]
      end for
    else
      for k in antecedentes do
        for j in granularidad_i do
          añadir valor [granularidad_i, etiqueta_j]
        end for
      end for
    end if
  end for
```

En función de las granularidades de los antecedentes, el tamaño del conjunto de éstos puede variar. Si se busca limitar el tamaño de las bases de reglas generadas durante la inicialización, será necesario controlar de alguna forma que las granularidades que se inicializan de modo aleatorio no superen unos determinados valores.

El tamaño de una base de reglas, en función de los valores de granularidad para cada una de las variables del antecedente se puede calcular como el producto de los valores de todas ellas, esto es:

$$N_{RB} = \prod_{i=1}^{N} granularidad_i \tag{3.9}$$

Si se desea limitar el número de antecedentes posibles y, por ende, el tamaño de la base de reglas del individuo inicializado con estas variables, se podría disminuir en una unidad la granularidad en una variable aleatoria, hasta que el número de antecedentes se sitúe por debajo de un umbral determinado.

Una vez conocida la forma de obtención de todos los antecedentes posibles, se puede describir ahora cómo se obtienen los consecuentes para dichos antecedentes. La forma de realizarlo depende del método de inicialización utilizado, tal y como se describe a continuación:

- *Inicialización aleatoria:* para cada uno de los elementos en los antecedentes se elige de modo aleatorio una etiqueta que defina cada una de las variables del consecuente. La granularidad de éstas se conoce previamente a la ejecución del algoritmo, por lo que sólo hay que elegir aleatoriamente una de las posibilidades. Una vez se realice esta operación, sólo hay que añadir al antecedente tantas tuplas [*granularidad*, *etiqueta*] como variables en el consecuente haya, donde el valor *etiqueta* siempre es un valor entre 1 y *granularidad*.
- *Inicialización Wang y Mendel:* en este método de inicialización se introduce conocimiento experto que se encuentra contenido en el conjunto de ejemplos utilizado por el proceso de aprendizaje.

Para cada uno de los ejemplos en el conjunto se busca el antecedente que registra un mayor grado de disparo, y se selecciona para cada variable del consecuente la etiqueta borrosa más cercana al valor que contiene el ejemplo, tal y como se detalla en el algoritmo 7.

Algoritmo 7 Inicialización de los consecuentes mediante el método de Wang y Mendel

```
for i in ejemplos do
  buscar antecedente con mejor grado de disparo en el conjunto de antecedentes
  for j in tamanoConsecuente do
    buscar etiqueta más cercana al valor de consecuente[j]
    añadir tupla [granularidad, etiqueta] al antecedente seleccionado
  end for
end for
```

3.5. Operador de selección

Mediante este mecanismo se seleccionarán aquellos individuos que serán los que generen la descendencia. Este operador estará basado en la selección por torneo.

La selección por torneo consiste en seleccionar aleatoriamente n individuos de la población, y compararlos entre ellos para ver cuál es el mejor de todos. Aquel que cumpla dicha condición es añadido a la lista de padres. Existen dos maneras de llevar a cabo la operación de torneo:

- *Con reemplazamiento:* cuando un individuo resulta seleccionado, puede volver a participar en otro torneo posterior. De esta forma en la selección de individuos candidatos es posible que alguno de ellos aparezca más de una vez.
- *Sin reemplazamiento:* al ser seleccionado un individuo, éste se excluye definitivamente de los posibles participantes para el torneo, de modo que no existen repeticiones posibles en el conjunto de individuos seleccionados.

Algoritmo 8 Funcionamiento del algoritmo de torneo para la selección de individuos

```
while seleccionados < tamanoSeleccion do
  while grupo < tamanoTorneo do
    seleccionar aleatoriamente un individuo de la población
    si no está contenido en el grupo, añadirlo
  end while
  seleccionar mejor individuo del grupo
  if no hay reemplazamiento then
    eliminar el mejor del grupo de los participantes en la selección
  end if
  añadir a seleccionados el mejor del grupo
end while
```

El mecanismo de selección devuelve una colección de individuos que serán los padres de los nuevos

individuos. Cuando el tamaño del torneo es 2, se denomina a esta operación como *torneo binario*, que es el método de selección que se utilizará en este caso.

3.6. Operador de cruce

Durante esta operación se toman dos individuos del conjunto de padres, y se realizan sobre ellos una serie de operaciones que tienen como finalidad la mezcla de la información de sus genotipos.

Para la resolución del problema hay que considerar una regla como la unidad mínima de conocimiento, de modo que ésta sea indivisible, esto es, la operación de cruce debe alterar la composición de los individuos recombinando las reglas que contienen, sin alterar el contenido que éstas presentan como unidad de conocimiento. Este operador se implementará de tal forma que la información genética de los individuos padre se ponga en común al inicio del proceso del cruce, repartiéndola luego entre los dos individuos que se generan como descendencia.

Cada individuo de la población contiene la información necesaria para ser, de forma autónoma, una solución al problema de control del robot. Esto significa que cada uno de ellos es capaz de generar una respuesta para cada ejemplo que contiene el conjunto de entrenamiento. Si las reglas que forman los dos individuos padre son puestas en común, la información para hacer frente al problema estará duplicada, y será necesario repartirla entre los dos individuos que se forman como descendencia de tal modo que cada uno de ellos sea una nueva solución al problema. Hay que tener en cuenta que el operador de cruce no puede generar individuos con reglas inconsistentes o redundantes entre sí. Esto se puede realizar tal y como se detalla en el algoritmo 9.

Algoritmo 9 Esquema del proceso de cruce entre dos individuos

```
Require: genPadres ⇐ información genética de los padres puesta en común
  for i in ejemplos do
    if genPadres está vacío then
      break
    end if
    seleccionar reglas activadas por ejemplos[i]
    while reglasSeleccionadas no está vacío do
      j ⇐ regla aleatoria dentro de reglasSeleccionadas
      if hijo1 e hijo2 no cubren ejemplos[i] then
        añadir reglasSelecionadas[j] a hijo1 o hijo2 aleatoriamente
      else if hijo1 cubre ejemplos[i] e hijo2 no cubre ejemplos[i] then
        añadir reglasSelecionadas[j] a hijo2
      else if hijo2 cubre ejemplos[i] e hijo1 no cubre ejemplos[i] then
        añadir reglasSelecionadas[j] a hijo1
      end if
      eliminar reglasSeleccionadas[j]
    end while
  end for
```

El operador de cruce se ejecuta con una cierta probabilidad, que en este caso se ha especificado

inicialmente en 0.5 y, en el caso de que no se lleve a cabo, los dos individuos participantes son marcados para ejecutar obligatoriamente la operación de mutación sobre ellos.

Aunque la obtención de las parejas a cruzar podría hacerse directamente por orden en el conjunto de individuos candidatos para el cruce, esto presenta el inconveniente de que, si dos individuos adyacentes son iguales, la operación de cruce no dará el resultado deseado, además de que se podría perjudicar gravemente la diversidad de la población del algoritmo evolutivo. En lugar de seleccionar los individuos dos a dos, por orden, se comprueba si, al intentar obtener el elemento i del conjunto, el $i+1$ es igual a éste, en caso afirmativo, se saltan las posiciones siguientes hasta encontrar el individuo en la posición $i+k$ que no sea igual al de la posición i, y se produce el intercambio entre el $i+1$ y el $i+k$. Este proceso se detalla con claridad en el algoritmo 10.

Algoritmo 10 Formación de parejas partiendo del conjunto de individuos candidatos al cruce

Require: *candidatos* $\Leftarrow$ conjunto de individuos candidatos al cruce
Require: $i \Leftarrow$ posición en la que se está iterando en ese conjunto actualmente
  $salto = 1$
  while $candidatos[i] = candidatos[i + salto]$ **do**
    $salto = salto + 1$
  end while
  if $salto > 1$ **then**
    intercambiar individuos en las posiciones $i + 1$ e $i + salto$
  end if
  cruzar individuos en las posiciones i e $i + 1$
  $i = i + 2$

El hecho de que al final del algoritmo 10 la posición i se adelante dos unidades es porque, al organizarse los individuos en parejas, la siguiente de ellas empezará en esa posición.

Hay que hacer notar que, debido al diseño de este operador, se consiguen una serie de ventajas tras su aplicación:

- Se eliminan reglas que proporcionan información redundante dentro de los individuos. Si un determinado conjunto de ejemplos ya está cubierto por alguna de las reglas de los descendientes, no se produce la inserción de ninguna más que no proporcione información adicional.
- Las reglas que no son utilizadas por ningún ejemplo del conjunto de entrenamiento no son incluidas en la descendencia, ya que no aportan ninguna información relevante para la solución del problema.

3.7. Operador de mutación

La mutación es un operador unario que se aplica sobre el individuo para que éste sufra una serie de cambios sobre su contenido genético para introducir mayor diversidad en la población, forzar la aparición de nuevas soluciones candidatas dentro de la población y permitir que la búsqueda del mejor valor de la función de evaluación se lleve a cabo en cualquier punto del espacio de soluciones.

En este problema concreto, un individuo está formado por un conjunto de reglas que forman una base de conocimiento. Esto fuerza que sea posible aplicar distintos tipos de mutaciones sobre el individuo, y de esta forma generar diferentes resultados. Al igual que el cruce, este es un operador estocástico, eso es, existe una cierta probabilidad de que un individuo sea mutado, inicialmente este valor de probabilidad se ha establecido en 0.03, mucho más bajo que el operador de cruce, pues si no la búsqueda tendría un alto componente aleatorio. Todos los individuos que han sido marcados como de mutación obligada por diferentes razones ejecutan este operador con probabilidad 1.

En el caso de que un individuo sea mutado, se recorre su genotipo, y para cada una de las reglas se decide probabilísticamente si se va a realizar una mutación sobre ella o no. En caso afirmativo, existen dos alternativas para realizar el proceso, que se seleccionan cada una con una probabilidad de 0.5:

- *Mutación del antecedente:* para realizar la mutación del antecedente debe seleccionarse una de las variables de entrada, que será sobre la que se aplique la modificación. Si se tiene una etiqueta determinada en dicha variable, las opciones que hay son generalizarla o especializarla más. De entre estas dos opciones se puede realizar la selección también con un 0.5 de probabilidad cada una. En este caso, sólo hay que modificar el valor de *granularidad* en la tupla $[granularidad, etiqueta]$. Sin embargo, hay que tener cuidado pues realizar una operación de este tipo puede provocar que la regla entre en conflicto con otras que contenga la base de conocimiento, provocando inconsistencias o redundancias entre ellas. Es necesario realizar un análisis sobre el individuo para comprobar qué reglas son inconsistentes entre sí, y realizar las operaciones de reparación que sean necesarias. Si una etiqueta de una variable es especializada, debe garantizarse que la nueva que se ha seleccionado cubra el mismo número de ejemplos. Si es necesario se añadirán nuevas reglas para mantener dicha cobertura.

- *Mutación del consecuente:* en este caso, sólo existen etiquetas borrosas para un nivel de granularidad, por lo que la mutación debe realizarse de otra forma. Tras seleccionar una variable del consecuente, la modificación de su valor se realizará seleccionando la etiqueta borrosa que está a la derecha o a la izquierda de la que se está utilizando actualmente. En el caso de realizar este tipo de mutación no es necesario comprobar la integridad de la base de reglas, pues los valores de los antecedentes siguen siendo los mismos. En este caso habrá de realizarse la modificación del valor de *etiqueta* sobre la tupla $[granularidad, etiqueta]$ de la variable mutada.

Tras la aplicación del operador de mutación, se obtiene un individuo que es añadido a la población. Todo este proceso se describe en el algoritmo 11.

Aunque no aparezca reflejado en el algoritmo 11 por simplicidad, hay que destacar que el algoritmo debe controlar los casos especiales donde no se pueda realizar alguna operación, como seleccionar la etiqueta a la derecha del consecuente actual porque ya se encuentra en su valor máximo. Esto es muy importante para evitar introducir información errónea dentro del conocimiento representado en la base de reglas.

En el caso de tener que realizar reparaciones para mantener la integridad del individuo, se deben detectar qué reglas han sido modificadas durante el proceso de mutación, esto es, cuáles son las que se han creado más recientemente. Para realizar las reparaciones se debe tener en cuenta lo siguiente:

- Si se produce un solapamiento total entre dos reglas, esto es, se produce un solapamiento entre

Algoritmo 11 Proceso de mutación de un individuo

```
for i in tamanoIndividuo do
  if probabilidad > umbral then
    t ⇐ tipo de mutación seleccionada
    if t = mutación del antecedente then
      seleccionar una variable del antecedente aleatoriamente
      seleccionar generalización o especialización aleatoriamente
      analizar integridad del individuo
    else
      seleccionar variable del consecuente aleatoriamente
      modificar etiqueta borrosa a la derecha o a la izquierda aleatoriamente
    end if
  end if
end for
```

el rango que cubren las etiquetas borrosas de todas las variables del antecedente y, por tanto, éstas cubren los mismos ejemplos, se elimina la más antigua, para de esta manera forzar que se produzcan cambios en los individuos durante el proceso.

- Si la el solapamiento es parcial, esto es, sólo en alguna variable de todas las que forman el antecedente de las reglas se produce un solapamiento, se debe disminuir la granularidad de las etiquetas en conflicto en la regla más antigua, hasta que se subsane el conflicto. Este es un proceso complicado, ya que durante las reparaciones por esta vía se puede producir una pérdida de información en algunas de las reglas modificadas, ya que al disminuir la granularidad de las etiquetas, éstas pueden dejar de cubrir algunos ejemplos que antes sí cubrían. Hay que tener encuentra este tipo de situaciones y añadir reglas adicionales para mantener el grado de cobertura que se registraba inicialmente.

3.8. Operador de reemplazamiento

Debido a que los operadores de cruce y de mutación añaden nuevos individuos a la población, para mantener constante su tamaño es necesario aplicar este operador para decidir cuáles de ellos son los que sobreviven y pasan a la siguiente iteración del algoritmo, descartando el resto.

Cada individuo se encuentra puntuado mediante un valor que le asigna la función de *fitness*. El reemplazamiento realizará una ordenación de los individuos de mayor a menor valor de su *fitness*, y seleccionará los N primeros para pasar a la siguiente iteración del algoritmo, descartando el resto. De esta forma se asegura que el valor de puntuación de la población siempre tenderá a registrar un crecimiento a lo largo de las diferentes iteraciones de algoritmo.

3.9. Puntuación de los individuos, función de *fitness*

En esencia, la función de *fitness* asigna un valor de puntuación a cada individuo dentro de la población para definir de modo objetivo cuán idóneo es como solución al problema que se pretende resolver.

En este caso concreto, el problema al que se pretende hacer frente mediante el uso del algoritmo evolutivo es la búsqueda de una base de conocimiento que pueda utilizar el controlador borroso de un robot móvil para poder moverse de un punto a otro del entorno.

Además de la precisión del controlador representado en el individuo, la función de puntuación debe tener en cuenta el número de reglas que lo forman, obteniéndose menor puntuación cuanto mayor sea éste; debido a que, a mayor número de reglas, existe una mayor probabilidad de que se esté produciendo sobreaprendizaje.

La función de evaluación utilizada para puntuar los individuos en este caso se divide en dos partes, por un lado tenemos el *fitness básico*, que contiene una puntuación para definir la idoneidad de una solución al problema del control del robot sin tener en cuenta ningún factor más, y por otro lado el *fitness extendido*, que además de lo anterior tiene en cuenta el número de reglas de la base de conocimiento evaluada.

La fórmula para el cálculo del fitness básico en este problema se ha definido de la siguiente manera:

$$err = 1 - \frac{\sum_{i=1}^{N_{ej}} \sum_{j=1}^{T_{cons}} \left(\frac{V_{propuesto}[i,j] - V_{objetivo}[i,j]}{maxDiff[j]} \right)^2}{N_{ej} \cdot T_{cons}} \tag{3.10}$$

donde N_{ej} es el tamaño del conjunto de ejemplos, y T_{cons} es el número de variables que tiene el consecuente de las reglas. Por otra parte, $V_{propuesto}[i,j]$ es el valor del j-ésimo consecuente que la base de reglas ofrece como salida para el i-ésimo ejemplo del conjunto utilizado para la evaluación, y $V_{objetivo}[i,j]$ es el valor análogo especificado por el ejemplo. $maxDiff[j]$ representa la máxima diferencia que puede darse entre los dos valores anteriores, esto es, el rango de la variable j-ésima del consecuente.

Mediante esta expresión lo que se pretende conseguir es una medida objetiva del error que comete la base de reglas al calcular una salida que se conoce de antemano para un ejemplo. Si lo que se pretende es otorgar una puntuación al individuo que contiene dicha base de reglas, ésta debe ser tanto más alta cuanto menor sea el valor de err, y puede definirse como:

$$f' = 1 - err \tag{3.11}$$

ya que el valor del error se encuentra siempre dentro del intervalo $[0, 1]$.

Este valor de puntuación, conocido como *fitness básico* se aplica para todos los individuos que han sido obtenidos de la etapa de inicialización. Una vez realizada la evaluación sobre ellos debe almacenarse el mejor valor de f' obtenido para dicha población inicial, que se denominará f'_0, y su tamaño, esto es, el número de reglas de ese individuo, que se conocerá como S_0, pues serán utilizados para calcular el *fitness extendido*, que es la fórmula utilizada para hallar la puntuación de los individuos en las sucesivas iteraciones del algoritmo. Se define de la siguiente manera:

$$f = \frac{\left(f' + \beta \cdot f'_0 \cdot \frac{S_{max} - S_0}{S_{max}} \right)}{1 + \beta} \tag{3.12}$$

Por una parte, S_{max} representa el tamaño máximo que puede tener un individuo, el parámetro β representa el grado de importancia que se otorga al tamaño de la base de reglas. Cuanto mayor sea β, más se primará que el número de reglas que contiene la solución elegida por el algoritmo sea lo más bajo posible; su valor debe estar contenido en el intervalo $[0, 1]$.

Durante el proceso de evaluación se pueden detectar reglas que no son disparadas por ningún ejemplo y que, por lo tanto, no son relevantes para formar parte de ninguna solución al problema de navegación. Es necesario identificar, para todos los individuos evaluados, qué reglas son inútiles para el proceso de aprendizaje, y eliminarlas de las bases de conocimiento, de esta manera se consigue un doble objetivo, por una parte la minimización del número de reglas que contienen las soluciones candidatas, y por otra parte acelerar el proceso de aprendizaje, ya que cuantas más reglas haya que procesar, mayor será el tiempo necesario para llevar a cabo todas las operaciones.

Finalmente, hay que tener en cuenta que se van a realizar cálculos para dos valores de *fitness* diferentes:

- *Fitness de entrenamiento:* representa la calidad de la solución en función de los ejemplos que se están utilizando para el proceso de aprendizaje, esto es, realiza la evaluación de la calidad del individuo con el propio conjunto de entrenamiento utilizado para realizar la inyección de conocimiento experto en el algoritmo evolutivo.
- *Fitness de validación:* dentro del conjunto total de ejemplos, se reserva un pequeño porcentaje para realizar el cálculo de este valor, que se concibe como una medición de la calidad del individuo como solución al problema, con un conjunto de ejemplos que no es el utilizado para realizar el aprendizaje. Este valor dará una idea más precisa de la calidad del individuo ante entradas distintas a las de los ejemplos.

A pesar de que la aproximación anterior es una buena idea, la función de evaluación debe recoger el error cometido en cada uno de los objetivos que es necesario analizar para el controlador desarrollado. En este proyecto, la información que es interesante utilizar para evaluar es: el tiempo de colisión con un obstáculo, el tiempo de llegada al objetivo desde el punto actual, y el ángulo de error que forma el robot con el objetivo. Es por esta razón por la que se propone un método de evaluación alternativo.

Para la construcción de esta función de evaluación se ha tomado como referencia el trabajo publicado en [11], que se debe adaptar al problema concreto que se pretende abordar en este proyecto. Dicha idea pasar por utilizar una serie de valores de error, α_1, α_2 y α_3, que tengan relación directa con los valores de salida de la base de conocimiento. La redefinición de la función de evaluación afecta a la parte que se ha denominado como *fitness básico* en las ecuaciones 3.10 y 3.11. Se sigue manteniendo la fórmula para el cálculo del *fitness extendido* definida previamente en la ecuación 3.12. Se define ahora un valor global de error para una base de reglas que viene dado por:

$$SF(RB(e)) = w_1 \cdot \alpha_1 + w_2 \cdot \alpha_2 + w_3 \cdot \alpha_3 \tag{3.13}$$

donde RB es la base de reglas que se evalúa, e es el conjunto de ejemplos, y los valores w_1, w_2 y w_3 son los pesos que a cada componente del error total se le asignan. Cada uno de estos valores de error parcial se define, en este problema, de la siguiente forma:

$$\alpha_1' = \frac{1}{t_{col} + 0{,}01} \tag{3.14}$$

$$\alpha'_2 = t_{llegada} \tag{3.15}$$

$$\alpha'_3 = |\theta_{error}| \tag{3.16}$$

Estos valores no necesariamente tienen que estar dentro del rango $[0, 1]$, así que se deben normalizar antes de utilizarlos para realizar alguna operación con ellos. La normalización se lleva a cabo mediante la siguiente fórmula general:

$$\alpha = \frac{\alpha' - \alpha'_{min}}{\alpha'_{max} - \alpha'_{min}} \tag{3.17}$$

obteniendo de esta forma los valores α_1, α_2 y α_3.

Los valores mínimos y máximos para cada uno de los errores parciales dependen directamente del rango de las variables tanto de entrada como de salida. Para este problema, los valores de los rangos se muestran en la tabla 3.1, donde se hallan divididos entre antecedente y consecuente.

Variable	**Valor mínimo**	**Valor máximo**
Distancia frontal	0	3
Distancia izquierda	0	1
Distancia derecha	0	1
Distancia al objetivo	0	3
Ángulo al objetivo	$-\pi/4$	$\pi/4$
Velocidad lineal	0	1
Velocidad lineal	0	1
Velocidad angular	$-\pi/6$	$\pi/6$

Tabla 3.1: Rango de valores para las variables involucradas en las reglas.

Conocidos los valores entre los que se encuentran todas las variables que intervienen en el proceso de aprendizaje es posible realizar un análisis sobre los rangos entre los que están contenidos los valores para los errores parciales, que son los que se muestran en la tabla 3.2. Estos son los valores que se utilizarán para la normalización que se describe en la fórmula 3.17, dando lugar a los valores normalizados: α_1, α_2 y α_3.

Error parcial	α_{max}	α_{min}
α'_1	100	0.003322
α'_2	300	0
α'_3	$\pi/4$	0

Tabla 3.2: Rango de valores para los errores parciales de la función de evaluación

Una vez que se ha calculado el error total, $SF(RB(e))$ para la base de reglas, (ecuación 3.13), se calcula el valor de la función objetivo, que es el índice de la calidad global de la base de reglas. Esta parte es independiente del comportamiento que se pretende aprender, y sólo utiliza el error global

que se define específicamente para cada problema. La función objetivo viene dada por la siguiente expresión:

$$f(RB) = \frac{1}{2N_e} \sum_{i=1}^{N_e} (g(e_i))^2 \tag{3.18}$$

donde N_e es el número de ejemplos que se utilizan en el proceso de aprendizaje, y la función $g(e)$ está definida como:

$$g(e) = \begin{cases} (1 - h(e)) \cdot \zeta + 1 \ si \ h(e) \leq 1 \\ exp(1 - h(e)) \ si \ h(e) > 1 \end{cases} \tag{3.19}$$

y $h(e)$ se define como:

$$h(e) = \frac{min(SF(e)) + 1}{SF(RB(e) + 1} \tag{3.20}$$

esto es, el mínimo error total que se puede conseguir con la definición de α_1, α_2 y α_3 utilizadas, dividido entre el error total conseguido con los datos de evaluación utilizados.

Para aplicar estas fórmulas necesitamos una serie de información de la que no disponemos *a priori*, como t_{col}, $t_{llegada}$ y θ_{error}. Para obtenerla es necesario aplicar el **modelo de movimiento** del robot. Este modelo permite calcular el estado del robot cuando se aplica sobre él la salida que se obtiene de la base de conocimiento. Se supone como instante actual t_0, y como instante posterior a la acción de control, t_1. Para hacerlo correctamente hay que llevar a cabo una serie de pasos:

1. Calcular el punto en el que se encontrará el robot en el instante t_1.
2. Calcular el punto objetivo en t_0. Hay que tener en cuenta que el punto objetivo no es conocido durante el proceso de aprendizaje, sino que viene dado por la información que proporciona el antecedente, esto es, el ángulo al objetivo y la distancia en t_0.
3. Calcular la nueva distancia al objetivo, distancia frontal, trasera e izquierda en t_1, con toda la información que se ha obtenido en los pasos anteriores. Para determinar los puntos de colisión se realiza la suposición de que el robot no puede ir más allá del área que delimitan sus distancias laterales y frontal. Para calcular el tiempo de colisión se tiene en cuenta el ángulo del robot en t_1.

En la figura 3.4 se puede ver de qué manera se aplica el modelo de movimiento del robot para estimar los datos necesarios para proceder a la evaluación. El modelo de movimiento del robot está definido con las siguiente ecuaciones:

$$\begin{aligned} x' &= x - \tfrac{v}{\omega} sin\theta + \tfrac{v}{\omega} sin(\theta + \omega \cdot \Delta t) \\ y' &= y + \tfrac{v}{\omega} cos\theta - \tfrac{v}{\omega} cos(\theta + \omega \cdot \Delta t) \\ \theta' &= \theta + \omega \cdot \Delta t \end{aligned} \tag{3.21}$$

Cuando $\omega = 0$, se aplican las siguientes ecuaciones:

$$\begin{aligned} x' &= x + v \cdot \Delta t \cdot cos\theta \\ y' &= y + v \cdot \Delta t \cdot sin\theta \\ \theta' &= \theta \end{aligned} \tag{3.22}$$

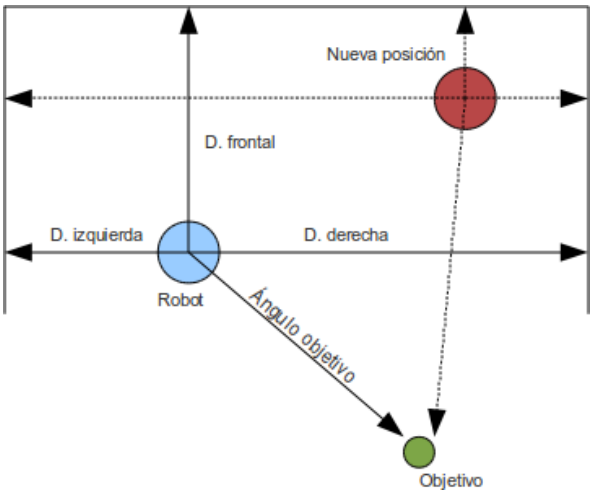

Figura 3.4: Diagrama que muestra la aplicación del modelo de movimiento del robot.

Hay que tener en cuenta que, tal como se ha planteado el problema de la evaluación, no tiene sentido la especificación de las coordenadas del robot, del objetivo y del punto de colisión en coordenadas absolutas. Es suficiente para abordar todos los cálculos que las coordenadas utilizadas sean relativas a la posición inicial del robot, considerando ésta como $P_0(x_0, y_0, \theta_0) = (0, 0, 0)$.

Como durante el proceso de aprendizaje no se conoce la posición del punto objetivo, es necesario utilizar la información de las variables de entrada para calcularla. Se conoce la posición en la que se encuentra el robot, que es P_0, así como la distancia y el ángulo al objetivo. Para situar el objetivo en el sistema de coordenadas del robot, se sitúa primero en $P'_F(x'_F, y'_F) = (0, d_{obj})$, de modo que se encuentre a la distancia del robot marcada por la variable de entrada d_{obj}, y posteriormente se aplica una transformación de rotación sobre el punto, para situarlo en ángulo θ_{obj} con el robot. Estas transformaciones se aplican de la siguiente forma:

$$\begin{aligned} x_F\prime &= x'_F \cdot cos\theta_{obj} - y'_F \cdot sin\theta_{obj} \\ y\prime &= x'_F \cdot sin\theta_{obj} + y'_F \cdot cos\theta_{obj} \end{aligned} \tag{3.23}$$

obteniendo de esta forma en punto objetivo respecto al robot, $P_F(x_F, y_F)$.

El cálculo del tiempo de colisión requiere de una técnica algo más compleja que la utilizada para el cálculo de la posición del objetivo respecto al robot. En este caso hay que tener en cuenta principalmente cuál es la orientación del robot una vez aplicado el comando de control, así como la nueva posición y las distancias recogidas por los sensores láser. En la figura 3.5 se puede ver la aproximación que se va a seguir para el cálculo del punto de colisión del robot y, por ende, del tiempo de colisión.

Para empezar, hay que comprobar a qué sector, de entre los que el láser está dividido, corresponde la orientación que el robot tiene en el nuevo punto donde se encuentra. Una vez identificado, disponemos del punto donde se encontraba inicialmente el robot y del punto donde se encuentra en el instante siguiente, así que podemos identificar una recta dada por la expresión $b = m \cdot a + n \rightarrow m \cdot a$, donde m está definido como:

$$m = \left(\frac{b_2 - b_1}{a_2 - a_1}\right) = \left(\frac{b}{a}\right) \tag{3.24}$$

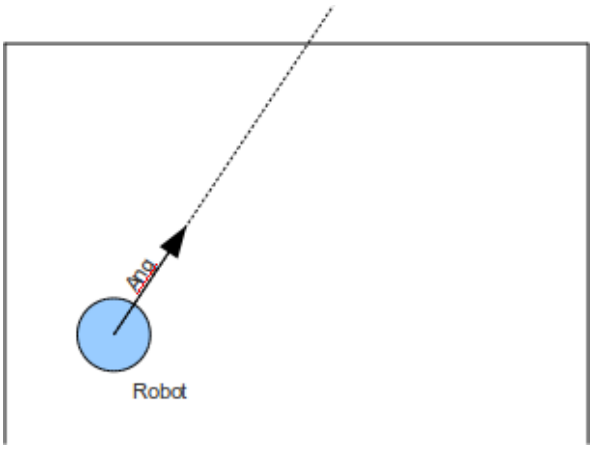

Figura 3.5: Cálculo del punto de colisión del robot.

La simplificación de las ecuaciones anteriores se produce porque uno de los puntos es el de partida del robot, que en todo caso será $P(a, b) = (0, 0)$. Una vez que se conoce la ecuación de la recta, se puede calcular el punto de colisión. En caso de que intervenga la distancia frontal, tendremos un b conocido, y en caso contrario, un a, que despejando en la ecuación de la recta nos permite obtener el otro componente del punto. Una vez obtenido, el cálculo del tiempo de colisión sólo depende de la distancia del nuevo punto al de colisión, y de la velocidad de movimiento especificada por el control, esto es:

$$t_{col} = \frac{d_{col}}{v} \tag{3.25}$$

donde la distancia de colisión, d_{col}, viene dada por la siguiente expresión:

$$d_{col} = \sqrt{(col_x - pos_x)^2 + (col_y - pos_y)^2} \tag{3.26}$$

siendo col_x y col_y las coordenadas del punto de colisión, y pos_x y pos_y las coordenadas de la posición del robot en el instante t_1

Finalmente, falta por calcular el nuevo ángulo que forma el robot, teniendo en cuenta la orientación del mismo, con el punto objetivo. Como el punto donde se encuentra el robot ($P(x, y)$), la orientación del robot (θ) y el punto donde está situado el objetivo, $P_F(x_F, y_F)$, se conocen, el ángulo se puede obtener de modo inmediato:

$$\alpha_{err} = arctan\left(\frac{x_F - x}{y_F - y}\right) + \theta \tag{3.27}$$

La utilización de la fórmula 3.13 para la evaluación permite la asignación de un grado de importancia, determinado por su peso, a cada uno de los factores que son fuente de error, que es lo que se desea cuantificar con la evaluación de una base de conocimiento.

3.10. Evasión de mínimos locales

Para evitar que el controlador borroso caiga en mínimos locales y no se pueda realizar el movimiento del robot desde su posición actual hasta el objetivo determinado, se puede realizar un estudio para

determinar un punto objetivo intermedio, cuando llegar hasta el original desde la posición actual es imposible mediante el trazado de una recta, tal y como se detalla en la figura 3.6.

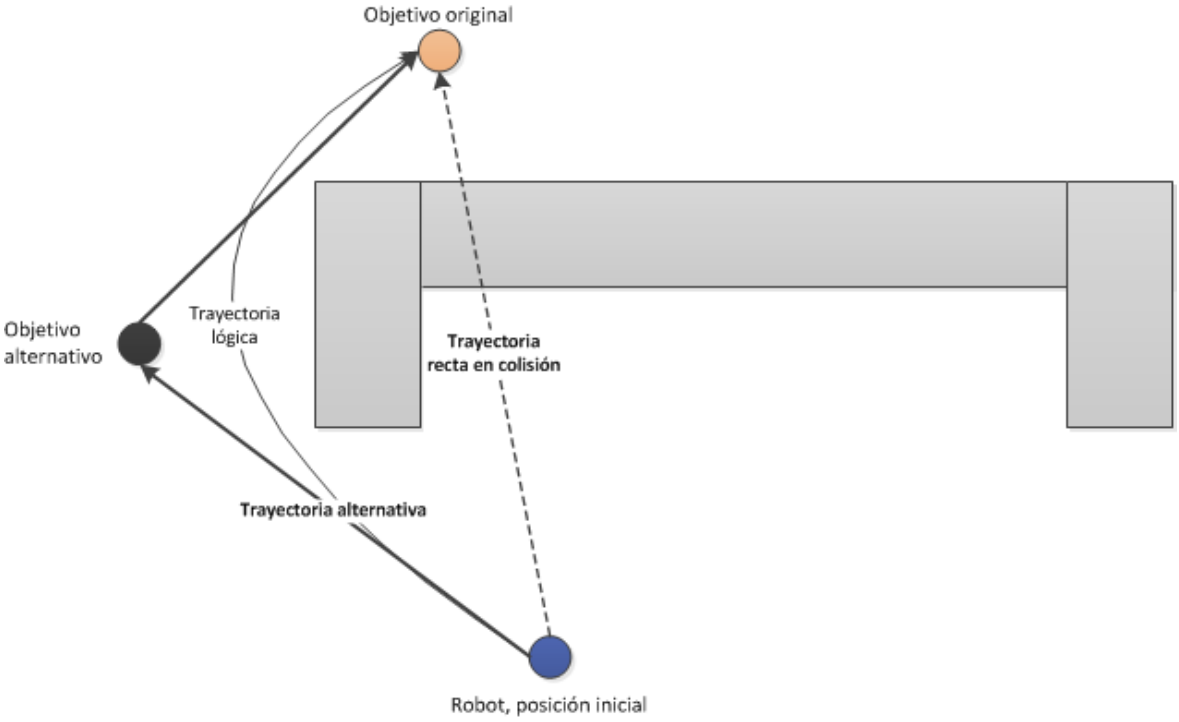

Figura 3.6: Definición de un punto objetivo alternativo cuando alcanzar el original en línea recta es imposible.

Cuando se produce esta situación, lo más adecuado es analizar las mediciones de los láser buscando huecos, seleccionando todos aquellos en los que la profundidad media, esto es, la distancia promedio de las mediciones de los láser en el rango identificado como hueco, cumpla que $d_{media} \geq d_{objetivo}$. En el caso de que no se cumpla esta condición, situación que puede darse cuando el objetivo está fuera del rango de medición máximo que alcanza el sensor, se deben buscar los huecos tal que cumplan $d_{media} \geq d_{maxSensor}$, esto es, que la distancia promedio sea al menos igual o superior al rango máximo permitido por el sensor, teniendo en cuenta que la distancia promedio de un rango de mediciones se lleva a cabo de la siguiente manera:

$$d_{media} = \frac{\sum_{i=1}^{N_{med}} d_{med[i]}}{N_{med}} \tag{3.28}$$

donde d_{med} representa la distancia de una medición, y N_{med} es el número de mediciones dentro del hueco identificado.

De entre todos los huecos que cumplan la anterior condición, hay que seleccionar como preferente aquel cuyo ángulo proporcione una desviación lo menor posible al objetivo, esto es, que el ángulo objetivo y el ángulo en el que se encuentra el hueco sea lo más similar posible. Encontrar huecos en toda la colección de mediciones no es una tarea trivial, un hueco puede serlo por diferentes circunstancias, y el análisis de las mediciones del sensor láser debe distinguirlas todas ellas. Este proceso se realiza del modo detallado en el algoritmo 12.

Una vez que se han encontrado todos los huecos dentro del rango total del láser, es necesario identificar aquel que menos alejado se encuentra de la ruta que permite un acceso más directo al objetivo original, esto es, cuyo ángulo difiera lo menos posible del ángulo entre el robot y el objetivo. En el caso de no estén conectados por una línea recta libre de obstáculos, se debe definir un punto objetivo intermedio situado en el hueco seleccionado como el mejor de todos los existentes. El ángulo seleccionado dentro del rango que cubre el hueco debe ser el que menor desviación tenga para alcanzar el objetivo original.

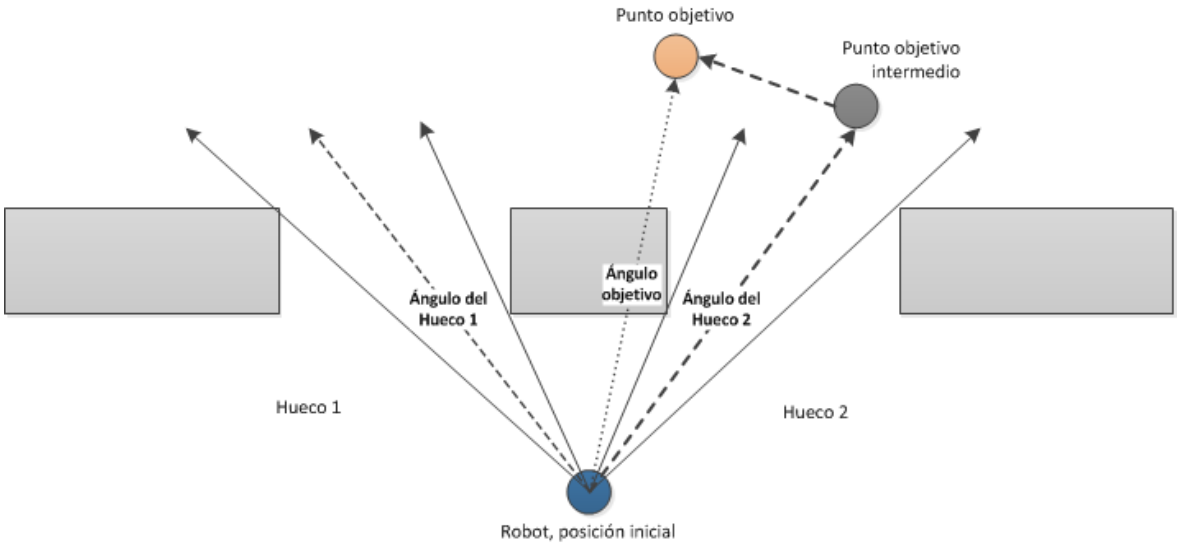

Figura 3.7: Selección del mejor hueco de entre todos los disponibles.

Como se puede ver en la figura 3.7, de entre todo el conjunto de huecos identificados, hay que seleccionar el mejor de ellos, esta selección se realiza en base a la cercanía que dicho hueco tiene con el objetivo. Si el objetivo original se encuentra fuera de ese hueco, entonces se define un punto objetivo intermedio que se sitúe dentro de éste. Sin embargo para poder realizar esta acción es necesario verificar en primer lugar que el hueco es lo suficientemente ancho para permitir el paso del robot, como se ve en la figura 3.8. Para esto hay que calcular la distancia entre los dos puntos de corte de los haces de inicio y de fin del hueco con algún obstáculo.

Para que un hueco sea considerado lo suficientemente ancho, debe cumplir la condición de que la distancia entre los dos puntos que cortan con los obstáculos desde los haces de inicio y de fin superen

Algoritmo 12 Proceso de identificación de huecos en las mediciones de las distancias

for i in distanciasLaser **do**
  if $i = 0$ o $distanciasLaser[i] >= d_{max}$ o $distanciasLaser[i] \geq distanciasLaser[i-1] + d_{minHueco}$ **then**
    $hueco_{inicio} = i$
  else if $i = i_{max}$ o $distanciasLaser[i] \geq distanciasLaser[i+1] + d_{minHueco}$ **then**
    $hueco_{fin} = i$
  end if
end for

la distancia de seguridad especificada, esto es, que $d_{ancho} \geq d_{seguridad}$.

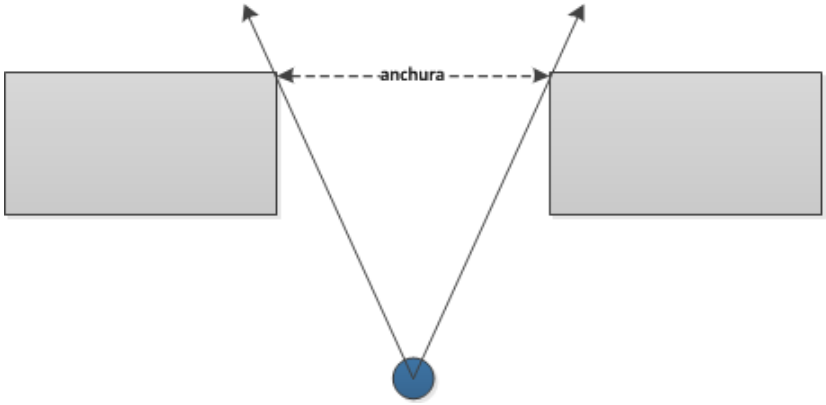

Figura 3.8: Comprobación de la anchura del hueco para permitir el paso del robot.

Capítulo 4

Análisis del proyecto

A lo largo de la presente sección se abordará un análisis detallado, que tiene como objetivo la especificación formal del problema que plantea el proyecto que se pretende desarrollar. Dicha especificación formal vendrá determinada por un análisis de requisitos que establezca cuáles son los aspectos que debe implementar la aplicación, y conocer restricciones en cuanto al diseño o el funcionamiento del sistema.

Posteriormente se analizarán tanto las librerías que son útiles para la implementación del proyecto, justificando su necesidad y qué aspectos positivos pueden introducir; finalmente se darán a conocer las herramientas que se utilizarán durante cada una de las fases del proyecto, justificando la idoneidad de la elección en función de las alternativas disponibles para cada uno de los casos.

4.1. Análisis de requisitos

La realización de un nuevo proyecto de software tiene como una de sus fases más importantes la especificación de los requisitos que deberá cumplir. Se ha seguido el modelo de especificación de requisitos propuesto en el estándar IEEE 830-1998 [7].

Con el análisis de requisitos se pretende reflejar de una forma clara y estandarizada las necesidades que debe cubrir obligatoriamente la aplicación, así como características de las que debe disponer. Para agrupar los requisitos, se pueden categorizar respecto a su función descriptiva, de la siguiente manera:

- *Requisitos funcionales:* son aquellos que describen el comportamiento interno de la aplicación, describiendo qué operaciones deben realizarse, y de qué manera, así como otros detalles técnicos y funcionalidades específicas.
- *Requisitos no funcionales:* al contrario que los requisitos funcionales, están enfocados a otros aspectos del software que no tienen tanto que ver con las operaciones que realizan, sino que se orientan más hacia características que pueden influenciar en el diseño y, por tanto, de un modo más indirecto, en la implementación.

En las siguientes secciones se detallan tanto los requisitos funcionales como los no funcionales, cada

requisito está etiquetado por un identificador único y diferenciado según el tipo, una descripción que procura ser completa, detallada y libre de ambigüedades, y un valor de prioridad, que puede ser uno de los siguientes:

- *Esencial:* el no cumplimiento de un requisito de este tipo invalida completamente la aplicación. Es el tipo de requisito más importante y de prioridad más elevada.
- *Deseable:* aunque su cumplimiento no es indispensable para la validez del proyecto, es recomendable si el plazo de entrega lo permite.
- *Opcional:* corresponde al nivel de prioridad más bajo. Corresponde a requisitos cuyo cumplimiento es optativo,

4.2. Requisitos funcionales

Se identificarán de forma unívoca con un identificador del tipo RF-X, siendo X un entero incremental con inicio en 1:

Requisito RF-1

Título: Aprendizaje de una base de reglas borrosas

Descripción: La finalidad de la aplicación desarrollada será la obtención de una base de reglas borrosas, que controlará el movimiento del robot. Dicha base de reglas deberá centrarse en el comportamiento que permita alcanzar un punto objetivo partiendo de una posición inicial, evitando los obstáculos que pudieran encontrarse entre ambos.

Prioridad: Esencial

Requisito RF-2

Título: Aproximación *Pittsburgh*

Descripción: El modelo de población que utiliza el algoritmo evolutivo sigue la aproximación Pittsburgh. Ésta define que cada individuo de la población estará formado por una serie de reglas borrosas, tal que en sí mismo cada individuo forma una base de conocimiento independiente de las demás.

Prioridad: Esencial

Requisito RF-3

Título: Salida del algoritmo

Descripción: Cuando la ejecución del algoritmo finalice, se debe poder volcar el contenido del mejor individuo seleccionado a un archivo, cuya información debe estar escrita en un formato que admita su

integración en el entorno de simulación Player/Stage. Este fichero contendrá: número de antecedentes y de consecuentes en cada regla, número de reglas, y una descripción del contenido de cada una a bajo nivel, donde cada línea contiene un antecedente o consecuente.

Prioridad: Esencial

Requisito RF-4

Título: Flexibilidad en la definición de la población y el funcionamiento general del algoritmo

Descripción: Se deben considerar parámetros de ejecución, y no elementos del diseño de la aplicación, ciertos valores como: el número de iteraciones del algoritmo, tamaño de la población, número máximo de reglas por individuo, etc. Debe hacerse un esfuerzo por parametrizar el mayor número de elementos posible, para flexibilizar las ejecuciones y evitar la dependencia del funcionamiento de decisiones de diseño.

Prioridad: Deseable

Requisito RF-5

Título: Seguimiento de las operaciones

Descripción: El programa permitirá el estudio y seguimiento de las operaciones realizadas por el algoritmo mediante una serie de *logs* que describan la población y las operaciones realizadas sobre los individuos. Para cada uno de ellos se detallará, al menos, el conjunto de operaciones que lo llevaron al estado actual, así como el contenido de las reglas que lo forman.

Prioridad: Esencial

Requisito RF-6

Título: Visualización de la evolución de la población

Descripción: Se debe generar una gráfica que muestre al menos la evolución del fitness de entrenamiento y de validación de la población a lo largo de las sucesivas iteraciones del algoritmo evolutivo, para detectar anomalías o un mal funcionamiento del mismo.

Prioridad: Deseable

Requisito RF-7

Título: Uso de ejemplos para el proceso de aprendizaje

Descripción: El aprendizaje debe estar basado en ejemplos, de modo que los diferentes operadores genéticos deben tenerlos en cuenta, bien de modo directo o indirecto, al menos para evaluar los individuos y poder establecer cuán buenos o malos son respecto al conocimiento que proporciona el conjunto de ejemplos. A este efecto, los ejemplos se dividen en dos grupos: de entrenamiento y de validación. Los primeros influyen de modo directo en la ejecución del algoritmo, participando en las

operaciones con los individuos, mientras que los segundos sólo sirven como evaluación para estimar la calidad del comportamiento de un individuo con información no aprendida.

Prioridad: Esencial

Requisito RF-8

Título: Generación de ejemplos a partir Player/Stage

Descripción: Una de las características de la herramienta Player/Stage es que genera unos archivos donde se escribe toda la información relativa al robot, las mediciones de los láser y su posición en cada uno de los instantes de tiempo que se han registrado. Como esta información es muy valiosa para generar ficheros de ejemplos a partir de rutas que el usuario realice de forma manual en dicha herramienta, se debe proveer un sistema de conversión de los ficheros en el formato de salida de Player/Stage al formato de fichero de ejemplos de la herramienta de aprendizaje.

Los ficheros de salida de la herramienta de simulación contienen información necesaria y sin procesar sobre el estado del robot. Para poder adaptarlos será necesaria la realización de una serie de cálculos que permitan generar los valores de entrada y de salida en el formato de las reglas utilizadas.

Prioridad: Esencial

Requisito RF-9

Título: Generación de ejemplos sintéticamente

Descripción: Aunque se pueden generar ejemplos manualmente a partir de rutas llevadas a cabo por el usuario, es conveniente disponer también de una herramienta que genere ejemplos sin ningún archivo de estos. Esta herramienta debe explorar el dominio de todas las variables de entrada y generar todos los posibles ejemplos dentro de ese espacio de información. Debido a que tanto las entradas como las salidas son valores en el espacio de los números reales, para hacer posible el trabajo de la herramienta se discretizará el rango de los datos en un número de niveles que será introducido por el usuario para cada caso.

Para la selección del consecuente más adecuado a cada uno de los antecedentes de los ejemplos habrá que evaluar para cada uno de ellos todos los posibles consecuentes que pueden tomar, y seleccionar el que mejor valor de evaluación reciba de la función de *fitness*.

Prioridad: Esencial

Requisito RF-10

Título: Operador de selección

Descripción: La selección de los individuos para el cruce se realizará mediante la técnica de torneo binario y con reemplazamiento. Esto quiere decir que de toda la población se extraen individuos por parejas, y de cada pareja se selecciona al mejor como candidato para el cruce. Al ser una selección con reemplazamiento, después de cada iteración del torneo los individuos vuelven a ser introducidos

en la población, por lo que existe la posibilidad de que vuelvan a ser seleccionados posteriormente.

Prioridad: Esencial

Requisito RF-11

Título: Operador de cruce

Descripción: El cruce se llevará a cabo entre una pareja de individuos devueltos por el operador de selección. Se generará una probabilidad de cruce antes de hacer la operación; si supera un determinado umbral, entonces se aplica el operador entre los individuos; en caso contrario, son marcados para la aplicación obligatoria del operador de mutación. Durante el cruce de individuos se realizarán las siguientes operaciones:

1. Se ponen en común las reglas de los individuos padre.
2. Para cada uno de los ejemplos, se comprueba qué reglas de las puestas en común son activadas.
3. Se comprueba si, para cada regla activada por ese ejemplo, se cubre por alguno de los hijos. Si ninguno de los dos cubre ese ejemplo, la regla se añade a uno de los dos al azar; si alguno de los dos la cubre, se añade al otro, y si lo hacen ambos, se descarta esa regla.

Tras recorrer todos los ejemplos en el conjunto de entrenamiento, se obtiene una pareja de individuos, que son el resultado de la aplicación de la función de cruce sobre los dos individuos padres. Esta pareja de individuos debe añadirse a la población una vez creados.

Prioridad: Esencial

Requisito RF-12

Título: Operador de mutación

Descripción: La mutación es un operador que, al igual que el cruce, se aplica sobre un individuo si, generada una probabilidad aleatoria, supera un umbral determinado, o bien el individuo está marcado como de mutación obligada, situación que puede darse circunstancialmente por no haber sido cruzado con otro en la iteración actual del algoritmo. En caso de aplicarse, la mutación ejecutará las siguientes operaciones:

1. Para cada regla del individuo, se genera una probabilidad aleatoria de mutación.
2. Si la probabilidad de mutación de una regla supera un umbral determinado, se selecciona aleatoriamente entre la mutación del antecedente o del consecuente de la regla.
3. En caso de aplicar el operador sobre el antecedente, se selecciona aleatoriamente una de las variables de entrada, y sobre ella o bien se aumenta su granularidad o bien se disminuye.
4. En caso de aplicarlo sobre el consecuente de la regla, se selecciona aleatoriamente uno de ellos, y sobre él, o bien se selecciona el valor inmediatamente inferior, o bien el inmediatamente superior.

Prioridad: Esencial

Requisito RF-13

Título: Operador de reemplazamiento

Descripción: Para determinar qué individuos de la población pasan a la iteración siguiente, se ordenan todos los individuos que la forman de mayor a menor, en función del valor de fitness de entrenamiento asignado por la función de evaluación. Los individuos seleccionados serán los N primeros, donde N es el tamaño de la población.

Prioridad: Esencial

Requisito RF-14

Título: Función de evaluación

Descripción: Con la finalidad de saber cuán bueno o malo es el comportamiento de los individuos respecto a un conjunto de ejemplos, se debe desarrollar una función de evaluación que tenga en cuenta lo siguiente:

1. El modelo de movimiento propuesto para el robot.
2. La distancia y el ángulo al objetivo, la velocidad y la distancia de colisión tras aplicar a cada ejemplo del conjunto la acción propuesta por la base de reglas que contiene el individuo evaluado.
3. El número de ejemplos con el que se ha realizado la evaluación.
4. El número de reglas del individuo evaluado.

Prioridad: Esencial

Requisito RF-15

Título: Reducción del número de ejemplos

Descripción: Cuanto mayor sea el número de ejemplos, mayor será el coste temporal de llevar a cabo el aprendizaje de una base de conocimiento. Debido a esto, y en previsión de que el número de ejemplos será muy elevado, en especial durante la generación sintética de ejemplos, se aplicará un algoritmo de reducción del número de ejemplos, que garantice una reducción significativa de su número, sin descartar aquellos realmente representativos dentro del conjunto. Esta reducción de ejemplos se llevará a cabo mediante el algoritmo IB2.

Prioridad: Deseable

Requisito RF-16

Título: Inicialización de la población

Descripción: Para esta primera etapa del algoritmo evolutivo se llevará a cabo una inicialización de dos tipos diferentes, para añadir una mayor diversidad al conjunto inicial. Los antecedentes se

inicializarán siempre del mismo modo, eligiendo una granularidad aleatoria para cada una de las variables de entrada y generando todas las posibles combinaciones de etiquetas para ellas con las granularidades obtenidas. La inicialización se realizará a nivel de individuo y se llevará a cabo al 50 % por cada uno de los siguientes métodos: aleatorio y Wang & Mendel.

Prioridad: Esencial

Requisito RF-17

Título: Flexibilidad en la definición de las reglas

Descripción: Para añadir una mayor flexibilidad al proceso de aprendizaje, y poder establecer un número variable de elementos en el antecedente y el consecuente de las reglas, se programará el código encargado de procesarlas de un modo lo más genérico posible, de modo que un cambio en la estructura de las reglas no implique una modificación del mismo.

Prioridad: Esencial

Requisito RF-18

Título: Formato e información contenida en las reglas

Descripción: Las reglas estarán definidas en el siguiente formato: Su antecedente estará formado por un triángulo por cada una de las variables de entrada. En el caso de que la granularidad sea la mínima absoluta, y por tanto sólo haya una posible etiqueta para definir esa variable, se hará mediante un rectángulo que abarque todo el rango posible de esa variable de entrada. Los consecuentes se especificarán mediante un único valor para cada variable de salida que contengan.

En cuanto a la información que debe contener cada regla, se especifica el siguiente formato para el antecedente:

1. Distancia Frontal. Rango: [0, 3]
2. Distancia derecha. Rango: [0, 1]
3. Distancia izquierda. Rango: [0, 1]
4. Distancia al objetivo. Rango: [0, 3]
5. Ángulo al objetivo. Rango: $[-\frac{\pi}{4}, \frac{\pi}{4}]$
6. Velocidad actual del robot. Rango: [0, 1]

Y para el consecuente:

1. Velocidad lineal. Rango: [0, 1]
2. Velocidad angular. Rango: $[-\frac{\pi}{6}, \frac{\pi}{6}]$

Prioridad: Esencial

Requisito RF-19

Título: Compatibilidad con un robot tipo Pioneer 3-DX

Descripción: Ante la previsión de poder realizar pruebas en un entorno real, para lo cual se dispone de un robot móvil de este tipo, la aplicación debe tener en cuenta la compatibilidad con los dispositivos con los que viene equipado este modelo de robot.

Prioridad: Esencial

4.3. Requisitos no funcionales

Se identificarán de forma unívoca con un identificador del tipo RNF-X, siendo X un entero incremental con inicio en 1:

Requisito RNF-1

Título: Posibilidad de modificación de los operadores genéticos

Descripción: El diseño de la aplicación tendrá en cuenta la posibilidad de modificar los diferentes operadores genéticos por otras implementaciones distintas, de modo que el tiempo empleado en realizar el cambio sea mínimo.

Prioridad: Deseable

Requisito RNF-2

Título: Tiempo de ejecución lo más bajo posible

Descripción: Se optimizará el código para que el tiempo de realización de las operaciones en una iteración del algoritmo sea lo más corto posible, de modo que la validación del algoritmo sea un proceso plausible en cuanto a tiempo de realización.

Prioridad: Deseable

Requisito RNF-3

Título: Utilización de la herramienta Player/Stage

Descripción: Para realizar las pruebas sobre la salida que ofrece el algoritmo implementado, se utilizará la herramienta Player/Stage, con la que se simulará el comportamiento de la base de reglas en un entorno similar al del funcionamiento del robot en una situación real.

Prioridad: Esencial

Requisito RNF-4

Título: Paralelización de código

Descripción: Se intentará en la medida de lo posible la paralelización de las secciones de código en las que el algoritmo tenga un mayor coste temporal, con la intención de agilizar la ejecución. Ya que el mayor problema radica en las partes que iteran directamente sobre el conjunto de ejemplos de entrenamiento, cuyo tamaño puede ser muy variable, se priorizará la paralelización en este tipo de bucles.

Prioridad: Opcional

Requisito RNF-5

Título: Independencia de los parámetros de ejecución

Descripción: Siempre que sea posible, se debe intentar que los parámetros que intervengan en la ejecución sean independientes del código y puedan ser definidos externamente, por ejemplo a través de un fichero de propiedades, con el fin de tener un algoritmo lo más general posible, y que los cambios potenciales en la ejecución no supongan un cambio en la codificación del mismo.

Prioridad: Deseable

Requisito RNF-6

Título: Uso de librerías

Descripción: Algunas funcionalidades del programa pueden tener una codificación más sencilla utilizando librerías que resuelvan al menos parte del problema que se pretende abordar, o que la faciliten; por esta razón se utilizarán siempre que sea posible.

Prioridad: Deseable

Requisito RNF-7

Título: Comentarios y código en inglés

Descripción: Para que pueda ser utilizado por otros investigadores, tanto los comentarios como los nombres de variables, funciones, etc, serán en inglés y razonablemente descriptivos.

Prioridad: Esencial

4.4. Análisis de librerías utilizadas

Uno de los requisitos no funcionales del proyecto, el RNF-7 (ver sec. 4.3), establece que es deseable la utilización de librerías para realizar alguna de las funciones que exige el proyecto; esto permite contar con un código externo que permita simplificar el diseño y la implementación, y que cuenta con

la ventaja de estar probado. A continuación, para cada una de las librerías que se utilizan en el proyecto, se describe su función y su actuación dentro de las funcionalidades que se requiere implementar.

JFR

Su nombre son las siglas de *Java Fuzzy Rules*, contiene diferentes implementaciones para una base de reglas borrosas como las que forman cada uno de los individuos de la población del algoritmo genético. Su inclusión en el proyecto permitirá ahorrar tiempo en la implementación de la estructura de los individuos, así como tener una implementación, válida para este proyecto, de los tipos principales de etiquetas borrosas que se van a utilizar.

El propósito de la utilización de esta librería es básicamente estructural, ya que provee de una serie de clases muy necesarias para abordar el problema que no será necesario implementar, al menos totalmente. En el caso en el que sea necesario añadir código a alguna de las clases que se van a utilizar, se utilizará la herencia. Las clases cuya utilización es interesante para el proyecto son las siguientes:

- *Triangle*, *Rectangle* y *Singleton*: son los tres tipos de etiquetas borrosas que se se van a utilizar en un principio. Los antecedentes de las reglas van a ser definidos mediante la primera y la segunda, mientras que los consecuentes lo harán mediante la última de ellas. La librería provee una implementación válida de las tres para este problema, que puede ser utilizada de modo directo.
- *RuleMamdani*, *Antecedent*, *ConsequentMamdani* y *FuzzySet*: son las clases necesarias para la definición de una regla borrosa, de modo que una instancia de la primera contiene una instancia de la segunda y la tercera; ambas están compuestas por instancias de la última de ellas, que representa una etiqueta borrosa de cualquier tipo.
- *RuleBaseMamdani*: es la implementación de una base de reglas que contiene instancias de *RuleMamdani*, se utiliza para tener una representación básica de una unidad de conocimiento, esto es, un individuo de la población en este caso.

JEVA

Es el acrónimo de *Java Evolutionary Algorithms*, contiene una serie de utilidades para algoritmos evolutivos escritos en el lenguaje Java, aunque en este caso concreto sólo se utilizará un pequeño conjunto de interfaces que permitan un modelado básico de la estructura general del algoritmo, sus operadores y los individuos. Define la estructura básica de las clases principales con las que trabajará el algoritmo, por lo que su utilización condiciona el diseño a alto nivel del algoritmo, añadiendo ventajas como la modularidad entre diferentes componentes del código.

Al igual que en el caso anterior, el propósito de la utilización de esta librería es meramente estructural, puesto que no añade ninguna funcionalidad al proyecto actual, sólo se utiliza como un conjunto de interfaces que las clases del programa implementan para añadir una mayor modularidad al diseño. En este caso las clases e interfaces a utilizar son las siguientes:

- *Genotype*, *Phenotype*, *Individual* y *Population*: son las interfaces que describen la población del

algoritmo, así como los individuos que la componen. Estas interfaces no especifican nada sobre la forma de implementación que deben tener las clases que implementan, tan sólo pretenden introducir una mayor modularidad en el diseño del algoritmo, haciendo que éste pueda trabajar con clases de implementaciones distintas, aunque tratándolas siempre como una instancia de una genérica. Estas, en concreto, permiten la definición de una población de individuos, definidos en base a un fenotipo y un genotipo determinados.

- *Pair*: esta es una clase de utilidad, un contenedor de una pareja de objetos de cualquier tipo. Es interesante su utilización para operaciones como el cruce, que debe devolver una pareja de individuos, como se detalla en el RF-11 (ver sec. 4.2).
- *Crosser*, *Mutator*, *Creator*, *Decoder*, *Evaluator*, *Replacer* y *Selector*: permiten una definición general de los operadores que todo algoritmo evolutivo utiliza. En la definición se contempla su utilización con los elementos de la población genéricos que se incluyen en esa misma librería, por lo que es un elemento de valor añadido al diseño por la misma razón por la que se han utilizado las interfaces que definen la población.
- Excepciones: algunos métodos incluidos en las interfaces utilizadas lanzan excepciones que define la propia librería. Es necesario utilizarlas para poder llevar a cabo las implementaciones de las interfaces utilizadas.

JFreeChart y JCommon

JFreeChart es una librería de código abierto escrita en Java para el dibujo de gráficas [6]. Su utilización se requiere para cumplir alguno de los requisitos que se han definido para este proyecto, como el RF-6 (ver sec. 4.2). A pesar de que hay más alternativas a esta librería, la elección se debe a que el desarrollador ya cuenta con experiencia previa en la utilización de esta. Su utilización permitirá la creación de una gráfica que se actualizará en tiempo real conforme el proceso de aprendizaje tenga lugar. Se utilizarán en este caso las siguientes clases:

- *ChartPanel*, *JFreeChart* y *ChartFactory*: la primera de ellas es un panel que se debe pegar en una ventana de la interfaz, es el contenedor que contiene la gráfica que se pretende dibujar; la segunda es el objeto que representa la gráfica propiamente dicha, y sobre quien hay que realizar todas las operaciones para el manejo de datos y el aspecto visual. La tercera es una clase que inicializa con unos valores por defecto las gráficas típicas que se desean representar, y se utilizará para una mayor comodidad en la inicialización de los objetos que permiten la representación gráfica de los datos.
- *XYSeriesCollection* y *XYSeries*: son los objetos que contienen los datos que se desean representar, esto es, las series de datos de la gráfica. Cada objeto *XYSeries* contiene una serie de puntos que forman una línea de evolución, cada uno de ellos define el fitness en un determinado instante.

JCommon es una librería utilizada por la anterior. se utiliza como soporte a utilidades de texto, gestión de código de configuración, gestión de dependencias, clases de interfaz gráfica, personalización del diseño de las gráficas y serialización de objetos [5].

QFTR, general, robot y mapping

Todas estas librerías no son independientes entre sí, unas necesitan a las otras para poder llevar a cabo la función para la que han sido diseñadas. No se necesitan para las funciones que lleva a cabo el algoritmo evolutivo, aunque sí para la ejecución de su salida en un controlador que recoge las entradas del robot y les aplica la base de reglas para poder obtener una salida. Todas ellas están escritas en C++.

General es, como su nombre indica, una librería de propósito no específico que puede ser de utilidad en diversas situaciones. Es utilizada por el resto de las librerías: QFTR, robot y mapping. Su estructura se detalla en la figura 4.1.

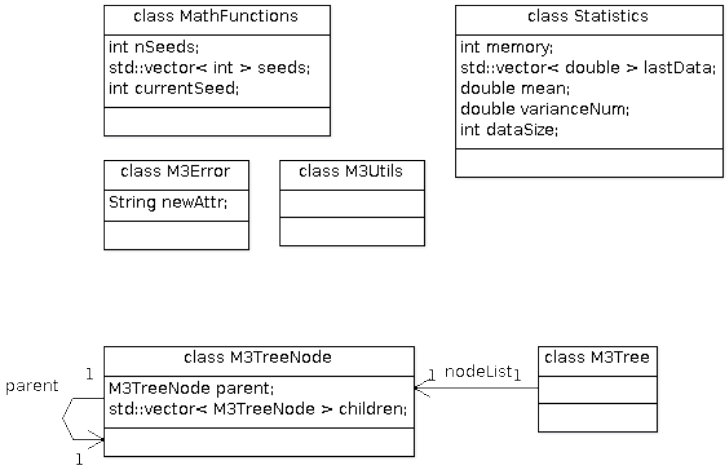

Figura 4.1: Diagrama de clases de la librería general.

QFTR es una librería que contiene todo lo necesario para la implementación de un controlador borroso basado tanto en reglas Mamdani como en reglas TSK. Mediante él es posible la representación de una base de reglas, y realizar la evaluación de las entradas utilizándolas. Su estructura es algo más complicada que la librería general, aunque la parte que nos interesa es la que concierne a la representación de una base de reglas, tal y como se detalla en la figura 4.2.

La librería Robot, por su parte, es necesaria para la integración con la herramienta Player/Stage [4, 18]. Contiene todas las clases y métodos necesarios para la interacción con el control del robot, es decir, el manejo de su velocidad y orientación, así como para la recopilación de información de los sensores que lleva incorporados. Juntas, la librería QFTR y Robot permitirán situar al robot en un entorno simulado, creado por Stage, recoger información a través de sus sensores y evaluarla mediante una base de reglas para obtener las salidas que deben aplicarse sobre el control del robot.

Las librerías descritas en este apartado dependen unas de otras para poder llevar a cabo la función que se requiere de ellas, tal y como se describe, en forma de diagrama de paquetes, en la figura 4.3.

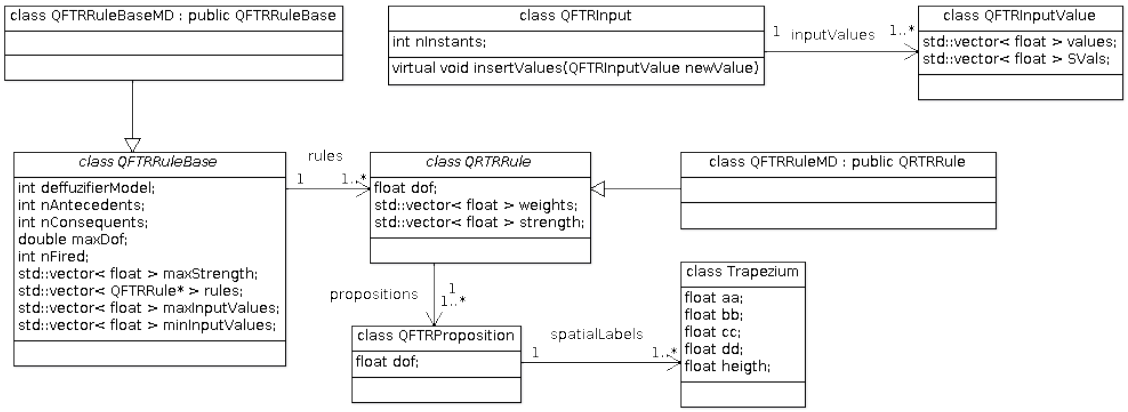

Figura 4.2: Diagrama de clases de la librería QFTR.

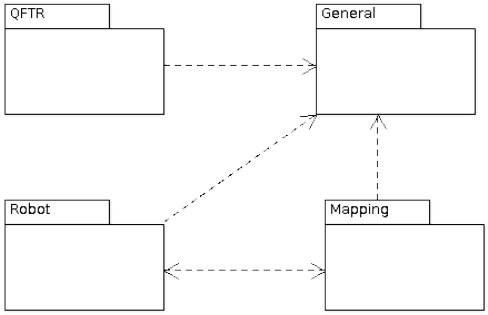

Figura 4.3: Diagrama que muestra las dependencias entre las librerías.

4.5. Análisis de herramientas

A lo largo de la siguiente sección se detallará el conjunto de herramientas utilizadas en función de las etapas de desarrollo del proyecto: diseño, implementación y pruebas.

4.5.1. Fase de diseño

Durante esta etapa la principal herramienta para la especificación del diseño han sido diferentes tipos de diagramas escritos en el lenguaje UML (*Unified Modelling Language*) [13], que permite la visualización, especificación, documentación y construcción de un sistema. UML es un lenguaje de modelado, que tiene una serie de diagramas que sirven a diferentes propósitos:

- *Diagramas de estructura:* están centrados en los elementos que existen en el sistema que se está modelando y sus relaciones. Son diagramas de este tipo los de clases, componentes, objetos, estructura compuesta, despliegue y de paquetes.

- *Diagramas de comportamiento:* enfocados a lo que debe suceder dentro del sistema, como los diagramas de casos de uso, de actividades, de estados y de secuencia.

- *Diagramas de interacción:* son un caso especial de los diagramas de comportamiento anteriores, aunque más centrados en el flujo de control y de datos entre los distintos elementos del sistema. Son diagramas de este tipo los de secuencia, comunicación, tiempos o los de vista de interacción.

A pesar de la cantidad de diagramas que permite realizar el lenguaje UML, en este proyecto sólo serán necesarios para realizar la documentación del diseño los diagramas de paquetes y de clases, ambos centrados en la estructura del sistema que se va a crear. Es posible que para poder explicar algún aspecto concreto de la construcción del sistema se recurra a otros diagramas que no se encuentran incluidos dentro del lenguaje UML.

Para la construcción de los diagramas se utilizará la herramienta de código abierto *ArgoUML* para Linux [17], que es la plataforma donde se va a desarrollar el proyecto. Es un editor de diagramas que soporta todos los diagramas de UML 1.4, incluye por lo tanto todos los diagramas que se van a utilizar. Sus diagramas soportan la notación estándar de UML, la notación Java y la notación C++. Es por tanto útil para el trabajo en diferentes lenguajes orientados a objetos.

Para la construcción de diagramas que no están incluidos dentro del lenguaje UML se han utilizado otras herramientas, como el *Openoffice.org Draw*.

4.5.2. Fase de implementación

En esta sección se engloban las aplicaciones utilizadas para el desarrollo del proyecto, el lenguaje seleccionado como base para la construcción y el entorno en el el que se llevará a cabo el trabajo.

En primer lugar se debe tomar la decisión del lenguaje utilizado para la construcción del programa. Es necesario que sea un lenguaje orientado a objetos, debido a la naturaleza del problema es el paradigma de programación que mejor se ajusta a las necesidades del proyecto. Los lenguajes candidatos para realizar la implementación son Java y C++. En este caso no se debe tomar una decisión absoluta, pues hacerlo aumentaría la complejidad técnica del proyecto. Debido a que no se compone de una única herramienta, hay que seleccionar el lenguaje que mejor se adapte a las necesidades de cada una.

En el caso del algoritmo evolutivo, la necesidad de una interfaz gráfica para mostrar la evolución del proceso de aprendizaje, hace que la mejor opción en este caso sea la utilización del lenguaje Java. para la herramienta generadora de ejemplos para la aplicación anterior también se utilizará este mismo lenguaje. Además de la compatibilidad con la librería JFreeChart, JCommon y JEVA, la utilización de Java para la implementación del algoritmo evolutivo también se ha fundamentado en una mayor experiencia por parte del desarrollador con el uso de este lenguaje.

En el caso de la implementación del controlador que utiliza la base de reglas obtenida de las herramientas anteriores, se utilizará el lenguaje C++ para tener compatibilidad asegurada con las librerías que utiliza esta parte del proyecto: QFTR, general, robot y mapping. Conlleva el inconveniente de la falta de experiencia en el desarrollo con este lenguaje, aunque en este caso la parte para la que se utilizará este lenguaje será mínima: la construcción de un controlador que aplique la base de reglas

en función de las entradas en un entorno simulado por la herramienta Player/Stage.

En ambos casos tanto el entorno de desarrollo como la aplicación utilizada para trabajar con el código fuente será la misma, se utilizará el *NetBeans* IDE (*Integrated Developing Environment*), que incorpora extensiones para trabajar tanto con Java como con C++. La elección de esta herramienta proporciona una serie de ventajas:

- Trabajo simultáneo con código escrito en Java y C++, gracias a su sistema de extensiones que permite ampliar las funcionalidades con las que viene por defecto.
- Autocompletado de código para ambos lenguajes. Proporciona un soporte más ampliado para Java que para C++, aunque en ambos lenguajes la ayuda que presta es bastante significativa.
- Incorporación del trabajo con un sistema de control de versiones tipo Subversion, que es el que se ha elegido en este proyecto para poder llevar a cabo la gestión de la configuración del código fuente (ver sec. 5.5). La integración con este sistema permite ver los cambios realizados en los ficheros de diferentes versiones, deshacer cualquiera de ellos, y establecer una conexión directa con el repositorio de Subversion desde el propio programa.
- Compilación desde dentro del propio IDE, tanto en Java como en C++.
- Permite hacer depuración de código línea a línea, mostrando los valores de cada variable en el contexto de ejecución para poder hacer una análisis detallado de lo que sucede en cada segmento de código.

4.5.3. Fase de validación

Las pruebas realizadas sobre el código fuente se realizará con el propio *NetBeans*, donde se puede utilizar la herramienta de depuración, tanto sobre código Java como sobre código C++. Para la fase de validación del proyecto propiamente dicha, se utilizará la herramienta Player/Stage para comprobar el comportamiento que una base de reglas tiene en un entorno simulado. Esta herramienta tiene tres partes diferenciadas [1, 4, 18]:

- *Player:* proporciona una interfaz en red para un tipo determinado de robot y de sensores hardware. Está implementado mediante un modelo cliente/servidor que permite controlar el robot mediante un programa escrito en cualquier lenguaje que tenga una conexión en red con el robot, soportando además conexiones concurrentes con distintos dispositivos.
- *Stage:* simula un conjunto de robots móviles en un espacio bidimensional representado mediante mapas de bits. Se incluyen varios modelos de sensores, como sonar, láser, etc. Presenta una interfaz estándar con Player de modo que la diferencia entre implementar un control para hardware partiendo de uno para simulación sea casi inmediato.
- *Gazebo:* es una herramienta de simulación en cierto modo similar a Stage, solo que la simulación que es capaz de realizar es en 3D. Presenta una interfaz con Player y es compatible con las interfaces generadas con Stage.

Capítulo 5

Gestión del proyecto

El desarrollo de cualquier proyecto de ingeniería requiere de una metodología de trabajo que permita la obtención de un producto final de calidad, y que se extienda más allá del proceso de trabajo que permite obtener el producto que se pretende construir.

Para poder decidir la metodología de desarrollo a utilizar, se debe antes realizar el análisis de riesgos del proyecto, con el fin de hacer que sean mínimos, ya que su impacto dependerá de la metodología elegida. Una vez seleccionada, se deberá realizar en base a ella una planificación temporal y una estimación de costes para cada una de las etapas a las que involucra, así como la gestión de la configuración de los diferentes procesos que forman parte de él. En este capítulo se detallará, en primer lugar, el análisis de riesgos necesario para realizar el conjunto de decisiones tomadas respecto a la gestión del proyecto, que se detallarán a continuación.

5.1. Análisis de riesgos

Todo proyecto de software conlleva una serie de riesgos que pueden poner en peligro el éxito de su resultado, o bien afectar a la calidad del producto final; entendiendo un riesgo como la probabilidad de que alguna circunstancia adversa pueda ocurrir durante el ciclo de vida del proyecto. La gestión de riesgos pretende anticiparse a ellos, analizar el impacto que pueden tener individualmente en el desarrollo y estudiar una serie de acciones para evitarlos; en el caso de que finalmente un riesgo se ponga de manifiesto, también se deben planificar acciones de recuperación destinadas a paliar sus efectos.

Para definir los riesgos presentes en el proyecto se ha seguido el estándar IEEE 1540-2001 [8], que detalla una serie de actividades que se deben realizar en este análisis:

1. *Identificar los riesgos:* todos ellos derivan de una situación de incertidumbre que se manifiesta durante el desarrollo, y pueden ser riesgos de diferentes tipos; pueden aparecer riesgos relacionados con la tecnología que se está usando, con los recursos involucrados en el proyecto, con las herramientas utilizadas y con los requisitos, entre otros.
2. *Analizar los riesgos identificados:* para cada uno de ellos debe estudiarse el posible impacto que

puede tener en el proyecto, la probabilidad de que ese riesgo se convierta en una realidad, y las consecuencias que tendría en caso de producirse.

3. *Planificación de riesgos:* hay que tener en cuenta antes del desarrollo de un proyecto que si los riesgos del proyecto son muy elevados, muy problemáticos o si, durante el desarrollo, la aparición de uno o más de esos riesgos hace superar unos umbrales de sobrecostes generados, retraso en los plazos de entrega o degradación de la calidad del producto final, el desarrollo del mismo debería interrumpirse.

4. *Gestión de riesgos:* debe llevarse a cabo una supervisión de los riesgos durante el desarrollo de forma que los riesgos puedan ser detectados en el momento de su aparición, y pueda de este modo reducirse su impacto sobre el proyecto y aplicar acciones de contingencia.

Por lo tanto, cada uno de los riesgos detallados en esta sección tendrá la siguiente información: descripción, probabilidad de ocurrencia, impacto en el proyecto y acciones preventivas para evitar su aparición.

5.1.1. Riesgos de gestión

Se corresponden con las incertidumbres generadas por la propia organización del proyecto, y tienen que ver con el cumplimiento de los plazos previstos, la disponibilidad de los recursos necesarios para el desarrollo, estimaciones y asunciones realizadas. Se detallan a continuación los riesgos de este tipo identificados.

Cumplimiento de los plazos de entrega

Descripción: el plazo de entrega del producto final de este proyecto se ha fijado en una fecha inamovible. Debido a la complejidad del proyecto y a la inexperiencia en el ámbito que abarca, los plazos reales para ejecutar cada una de las etapas del proyecto podrían variar respecto a los que se han estimado inicialmente.

Probabilidad: alta

Impacto: crítico

Prevención: la planificación del proyecto incluirá plazos de entrega más exigentes que los reales, para garantizar que el proyecto se finaliza dentro del plazo estipulado. Además, se planificarán las tareas de modo que algunas de ellas se puedan realizar en paralelo y de esta forma optimizar el tiempo invertido en cada una de ellas.

5.1.2. Riesgos del proyecto

Se corresponden con las incertidumbres que genera el propio proceso de desarrollo, teniendo como causas aspectos técnicos relacionados con las diferentes etapas por las que el proyecto va a pasar: análisis, diseño, implementación y validación. En este proyecto se han identificado los siguientes:

Cambios en los requisitos

Descripción: a pesar de que todos los requisitos deben estar bien definidos al inicio del proyecto, es posible que alguno de ellos pueda variar si los resultados ofrecidos por la herramienta no son los esperados. Así, por ejemplo, se pueden introducir variaciones en los requisitos que afectan a los diferentes operadores genéticos de la herramienta para aumentar la calidad del producto final antes de obtener una versión estable del producto.

Probabilidad: media

Impacto: alto

Prevención: se tendrá en cuenta esta circunstancia para la elección del tipo de metodología de desarrollo. Además, se intentará que los posibles cambios en los requisitos tengan lugar en fases iniciales del proyecto, analizando con los usuarios los avances llevados a cabo sobre el mismo para poder discutir la idoneidad de los posibles cambios que puedan surgir.

Falta de conocimiento en el ámbito del proyecto

Descripción: existe una falta de conocimiento por parte del desarrollador del proyecto sobre el ámbito que este abarca: algoritmos evolutivos y uso de reglas borrosas para la implementación de un controlador para robótica móvil.

Probabilidad: alta

Impacto: bajo

Prevención: previamente a las etapas de análisis, diseño e implementación de la herramienta, se dedicará un tiempo a la formación del desarrollador sobre aquellos temas que es necesario dominar para abordar el proyecto. Dichos recursos formativos estarán disponibles durante todo el desarrollo del mismo para atender posibles consultas posteriores.

Falta de conocimiento sobre las librerías utilizadas

Descripción: falta de conocimiento acerca de la estructura y funcionalidades de algunas librerías utilizadas en este proyecto, en especial de las que guardan relación con los algoritmos evolutivos y la lógica borrosa. La falta de experiencia en su utilización podría causar algún retraso en la implementación en una primera toma de contacto con ellas.

Probabilidad: baja

Impacto: bajo

Prevención: Se tendrá acceso a de toda la documentación disponible que guarde relación con las librerías que más conflicto puedan causar, para poder consultarla ante las dudas que puedan surgir a la hora de su utilización.

Falta de conocimiento sobre las herramientas utilizadas

Descripción: alguna de las herramientas utilizadas para el desarrollo del proyecto, como es el caso de Player/Stage para la simulación en robótica móvil, es también una novedad para el desarrollador del proyecto, por lo que la falta de experiencia en su utilización podría reflejarse en algún retraso, al necesitarse una etapa de toma de contacto para familiarizarse con ella.

Probabilidad: media

Impacto: bajo

Prevención: además de la documentación de Player [4] y Stage [18], se dispondrá de un tutorial de uso [14] para ayudar en la configuración de la herramienta para las tareas de simulación hasta estar familiarizado con ella.

Falta de experiencia con el lenguaje de desarrollo

Descripción: algunos de los componentes del sistema están escritos en C++, un lenguaje con el que el desarrollador no tiene experiencia previa. Este hecho podría causar alguna dificultad en el desarrollo en el que se involucre este lenguaje, o la no utilización de toda la potencia que C++ permite.

Probabilidad: media

Impacto: bajo

Prevención: además de recurrir a la documentación existente sobre el código ya implementado en este lenguaje, se recurrirá al personal que ha realizado la implementación anterior para solventar las dificultades iniciales. La familiaridad con el desarrollo en el lenguaje C puede ser un buen punto de partida para el desarrollo con C++, con el que guarda mucho parecido.

5.2. Metodología de desarrollo

La construcción de un proyecto de software conlleva una serie de actividades, o procesos. Detectar cuáles son necesarias para llevarlo a cabo no es una tarea sencilla, y para facilitar la tarea hay algunas normas que se pueden consultar al respecto, como la ISO 12207-1 y la IEEE 1074-2006. En este caso particular, se utilizará esta última como referencia, ya que el primero está más orientado a procesos de la organización, introduciendo muchos elementos que no son relevantes para este caso, como los procesos relacionados con la gestión de la calidad.

La norma IEEE 1074-2006 [9] proporciona una relación de procesos obligatorios para el desarrollo y mantenimiento de un proyecto de software. Los procesos se dividen en cuatro secciones, como se puede ver en la figura 5.1:

- El proceso de *modelo de ciclo de vida del software* requiere que se seleccione y utilice un modelo de ciclo de vida para el desarrollo; el propósito del estándar, sin embargo, no es definir ciclos de vida ni sus metodologías, sólo requerir que se utilice uno.

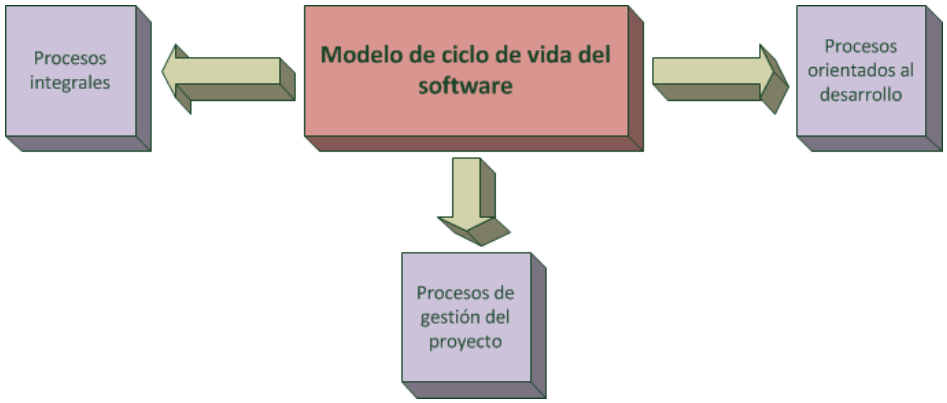

Figura 5.1: Procesos en el estándar IEEE 1074-2006.

- Los procesos de *gestión de proyecto* son los que inician, supervisan y controlan los proyectos a lo largo de todo su ciclo de vida, asegurando el cumplimiento de todas las actividades obligatorias.
- Los *orientados al desarrollo* comprenden todos aquellos realizados antes, durante y después del desarrollo. Incluyen, entre otras cosas, los procesos de análisis de requisitos, diseño e implementación, así como el de instalación y mantenimiento.
- Los procesos *integrales* son los que se necesitan para completar exitosamente las actividades de un proyecto, y se utilizan para asegurar la terminación y la calidad del mismo. Son de este tipo, por ejemplo, el proceso de desarrollo de la documentación, y el de verificación y validación.

El ciclo de vida del proyecto comprende una sucesión de etapas que comprenden desde que se inicia el proyecto de software hasta que se deja de utilizar, por tanto, no se da por finalizado el ciclo de vida cuando termina la codificación del mismo. Cada etapa lleva asociada una serie de tareas y generará unos resultados que serán utilizados en las siguientes.

Buena parte del éxito de un proyecto de software depende de la metodología de desarrollo utilizada para el mismo. Hay diversas alternativas, y ha de elegirse la más adecuada en función del tipo de producto a desarrollar, las necesidades de los clientes y los riesgos que puedan manifestarse. La elección es una decisión de gran importancia, que condicionará todo el ciclo de vida del proyecto, y del que depende el éxito del mismo. Trabajar sin una metodología de desarrollo clara o con una equivocada puede traducirse en un coste mayor que el deseado, o un producto final de calidad insuficiente. Para los proyectos de menor envergadura se puede optar por una implementación directa, sin un estudio intermedio de cuál es la mejor forma de abordar el desarrollo, pero es claramente necesario cuando su tamaño o complejidad aumentan.

En un entorno donde los requisitos pueden variar, modelos de ciclo de vida como el de cascada deben ser descartados. Se necesita uno que proporcione algo de flexibilidad y que permita responder ante circunstancias cambiantes, como el basado en prototipos, el incremental, el modelo en espiral o la programación extrema. En el caso de este proyecto en particular, el modelo basado en prototipos no es procedente: el esfuerzo de programación que se dedicaría para crear cada prototipo es inasumible dada la inflexibilidad de la fecha de entrega del producto final, que requiere que se realice el proyecto en un espacio de tiempo lo más corto posible; el modelo en espiral habría que descartarlo por razones

similares.

Figura 5.2: Ciclo de la Programación Extrema.

En este proyecto se utilizará un ciclo de vida basado en la programación extrema, que se muestra en la figura 5.2. Algunas de sus características no son aplicables a este proyecto, por ejemplo la programación en parejas, ya que este es un proyecto individual, sin embargo, se toma como base para la fase de planificación de tareas. Cada una de las etapas del ciclo de programación extrema son las siguientes:

1. *Planificación:* el contacto con el cliente permite obtener una serie de tareas a realizar. Las tareas no tienen por qué ser definidas al inicio del proyecto, sino que pueden añadirse, eliminarse o modificarse en cualquier momento del ciclo de vida. Cada tarea tiene una prioridad respecto a las demás, y se mide en número de semanas de trabajo que se estima que se tardará en completarla. Las tareas se deben comenzar siempre de modo inmediato, en pocas semanas, y siempre tendrán preferencia en la implementación aquellas con una duración más elevada, o que tengan un mayor riesgo.

2. *Diseño:* el diseño de las tareas que se van a implementar en un ciclo de programación extrema siempre debe buscar la máxima simplicidad, evitando generar una solución con funcionalidades extraordinarias a lo requerido por el cliente, que puede redefinir o eliminar requisitos en etapas siguientes del ciclo de vida. La refabricación implica diseñar de modo continuo a medida que el sistema se construye.

3. *Codificación:* en lugar de centrarse en pasar directamente del diseño a la codificación de la solución, en primer lugar se diseñarán una serie de pruebas unitarias. La codificación en cada etapa estará enfocada a pasar dichas pruebas unitarias, de tal modo que se pueda proporcionar retroalimentación instantánea al desarrollador, por ejemplo, si algo no funciona como se esperaba.

4. *Pruebas:* en este caso se refiere a pruebas de aceptación del producto por parte del cliente. Están

definidas en base a las tareas que se implementan en una etapa determinada del ciclo de vida, y se enfocan en las características generales del proyecto y sus funcionalidades.

5.3. Planificación

En el documento de anteproyecto ya se adelantaban algunas etapas de las que iba a constar el desarrollo, sin embargo, una vez que se analiza más profundamente el alcance, hay que revisar la previsión inicial para incluir las modificaciones oportunas. A continuación se detalla el conjunto de tareas en las que queda dividido; para cada una de ellas se especifica un número de horas de trabajo, y una duración estimada, tanto en horas como en semanas de desarrollo, asumiendo cuatro horas de dedicación diaria a las tareas relacionadas con el proyecto:

Formación sobre el ámbito del proyecto

Trabajo: 50 horas

La primera etapa del proyecto es necesario que sea un período de formación que permita abordarlo con conocimientos suficientes sobre el ámbito en el que va a ser necesario trabajar. Durante este tiempo el desarrollador se dedicará a la lectura de libros y artículos que le proporcionen una base para estar en condiciones de realizar las sucesivas tareas con un mínimo de conocimiento sobre las áreas en las que se apoya el proyecto: los algoritmos evolutivos y la lógica difusa.

Análisis de requisitos

Trabajo: 10 horas

La finalidad de esta etapa es la recopilación de una serie de requisitos que servirán como marco inicial en el que basar las siguientes, tanto el diseño como la implementación de la aplicación. Durante este período las reuniones con el cliente serán uno de los instrumentos para la obtención de la información que permita definir los requisitos del proyecto de un modo lo más claro e inequívoco posible. No se contempla la recogida de requisitos a través de casos de uso, puesto que para este tipo de proyecto no es procedente contemplarlos, debido a que la tarea principal es la del desarrollo de un algoritmo.

Diseño de la aplicación

Trabajo: 30 horas

Una vez establecidos los requisitos hay que proceder a realizar el diseño inicial de la aplicación. Hay que elaborar una estructura de clases que sea capaz de satisfacer todos los requisitos que se han obtenido de la tarea anterior, y a la vez que satisfaga los criterios de un buen diseño; que sin complicar demasiado la estructura de clases sea lo más modular posible, y por tanto extensible y que soporte posibles cambios en los componentes que lo forman. En esta etapa se incluye también la posibilidad de incluir diferentes librerías en el sistema para realizar algunas de las funciones del mismo, y por tanto

el diseño se verá condicionado por las funciones que de cada una de ellas se utilicen, lo que requiere un estudio previo de su potencial y aplicación en este problema concreto.

Implementación

Trabajo: 180 horas

En la etapa de implementación se recogen las iteraciones propias del ciclo de vida de la Programación Extrema. Partiendo de la etapa anterior con un diseño inicial y unos requisitos definidos, se entra en el ciclo propio de la metodología de desarrollo: se codifican algunas funciones, se realizan las pruebas unitarias para los módulos desarrollados, y posteriormente se revisan con el cliente. Es posible que en las siguientes iteraciones el cliente requiera que se realicen cambios o se añadan nuevas funciones, por lo que el diseño y la implementación se modificará en cada una de ellas hasta que se obtenga un producto final que cumpla los requisitos establecidos por el cliente.

Pruebas del sistema

Trabajo: 30 horas

Una vez finalizada la implementación es necesario someter al programa a una serie de pruebas con todos los módulos integrados, para comprobar que el funcionamiento de todos ellos una vez integrados no presenta ninguna anomalía respecto del comportamiento previsto. En esta etapa se engloba también la validación por parte del cliente para evaluar si la calidad final de la herramienta es suficiente.

Validación en entorno simulado

Trabajo: 20 horas

En esta etapa se tiene la herramienta ya probada por el cliente, a la que hay que realizar ahora una serie de pruebas de validación para verificar que las salidas que ofrece son adecuadas en distintos entornos, y que por tanto el comportamiento del robot es adecuado para situaciones diferentes, todo ello en un entorno simulado mediante la herramienta Player/Stage.

Validación en entorno real

Trabajo: 20 horas

Si la validación en un entorno simulado ha sido satisfactoria, es conveniente probar la salida del algoritmo evolutivo en un entorno con un robot real, para realizar una segunda validación en el entorno de funcionamiento que tendrá en última instancia la herramienta que se construyó a lo largo del proyecto.

Documentación

Trabajo: 65 horas

La tarea de elaboración de la documentación se extiende desde la finalización de la formación sobre el ámbito del proyecto hasta la validación en el entorno real del mismo. Todas y cada una de las tareas que se desarrollan entre la primera y la entrega del proyecto son objeto de realización de algún tipo de documentación, por eso esta es una tarea que se realiza en paralelo a todas las demás, aunque con una ocupación mucho menor que las otras.

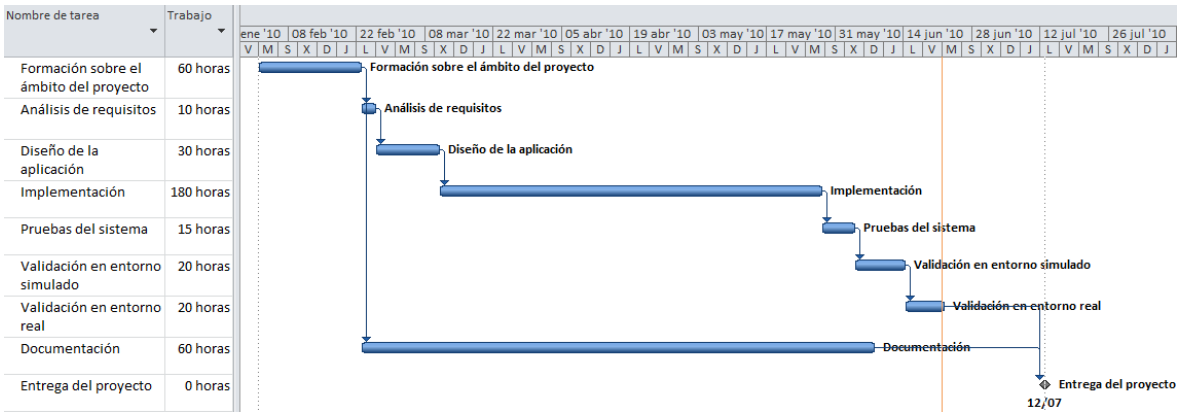

Figura 5.3: Diagrama de Gantt del proyecto.

En la figura 5.3 se puede ver gráficamente la planificación de las tareas que se detallaron anteriormente. Como se puede comprobar, la correspondiente a la realización de la documentación se extiende desde que comienza la fase de diseño del proyecto hasta que termina la fase de validación del mismo y se da por finalizado el desarrollo. El número de horas de trabajo que se estiman necesarias para completarlo es de 405.

5.4. Análisis de costes

Es una parte importante que hay que tener en cuenta previamente a la realización de cualquier proyecto, ya sea de software o no. Los costes de un proyecto dependen directamente tanto de las herramientas necesarias para poder llevarlo a cabo, como del tiempo que se ha tenido que invertir en su desarrollo hasta obtener el producto final. También se tendría que tener en cuenta el tiempo que, una vez finalizado el desarrollo, se tiene ha invertir en soporte y mantenimiento del mismo.

El coste total del proyecto se ve incrementado al tener en cuenta los gastos indirectos de su realización, bajo este concepto se agrupan elementos como la electricidad consumida por los equipos, gastos de agua, mantenimiento de instalaciones, etc. En un proyecto de este tipo, se omiten los gastos relativos a mantenimiento y soporte, ya que no es una tarea implícita en él. El coste desglosado del proyecto se detalla en la tabla 5.1.

5.5. Gestión de la configuración

Se denomina de esta forma al conjunto de procesos destinados a asegurar la validez de cualquier producto obtenido durante las etapas de desarrollo de un proyecto de software. Esto se consigue

Descripción	Unidades	Coste Unitario	Coste total
Equipo de sobremesa, Intel Core 2 Quad, 2.83 GHz, 8GB RAM	1	800.00 €	800.00 €
ArgoUML 0.30.1	1	0.00 €	0.00 €
Kile for Latex Developement	1	0.00 €	0.00 €
NetBeans IDE 6.8 for Java SE	1	0.00 €	0.00 €
Planner gestor de proyectos	1	0.00 €	0.00 €
Gasto de personal	405	9.50 €	3847.50 €
Subtotal			4647.50 €
Gastos indirectos (+20 %)			929.50 €
Total			5577.00 €

Tabla 5.1: Costes desglosados del proyecto, se incluyen los referentes a las herramientas utilizadas, al coste hardware, y al coste de personal.

mediante un estricto control de cambios y la disponibilidad permanente de una versión estable de cada elemento que participa en el desarrollo. En el caso de este proyecto la gestión de la configuración se aplicará al código fuente, la especificación de requisitos, la documentación, los productos entregables y el resultado de las pruebas.

El proceso de gestión de la configuración se realizará desde la puesta en marcha del desarrollo del proyecto hasta la retirada del mismo, esto incluye el período de mantenimiento una vez se ha puesto en el entorno de producción. En este proyecto en particular, se realizará a partir de la finalización de la tarea de formación sobre el ámbito del proyecto, y hasta la finalización de la fase de validación. El conjunto de decisiones tomadas en este ámbito se detallan a continuación.

Código fuente

Para alojar las sucesivas versiones estables del proyecto se empleará un software de gestión de versiones Subversion, alojado en un servidor del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela. Esto permitirá acceder a las distintas versiones estables del código que se suban a dicho servidor. En el repositorio, además de las clases que conforman el programa (archivos *.java*) también se alojará cualquier archivo auxiliar que utilice el programa, como ficheros de definición de parámetros, entradas y salidas, etc.

En paralelo a este proceso, el equipo en el que se trabaje estará equipado con *Dropbox*, un software que permite hacer una copia de respaldo en tiempo real de todos los archivos contenidos en un directorio, y que además guarda diferentes versiones de cada uno de los archivos, lo que permite estar protegido contra modificaciones indeseadas o la pérdida de información de archivos que no habían sido subidos al Subversion.

Al igual que el nombre de las variables y los comentarios dentro del código, el nombre de las clases estará en inglés, y procurará dar una descripción clara de cuál es la función que desempeña cada una de ellas.

Productos entregables

Como productos entregables de este proyecto de software se tiene:

- La herramienta generadora de ejemplos, bien a partir de un *log* de Player/Stage o de una descripción que permita obtener un conjunto de ejemplos sintéticos.
- La herramienta que ejecuta el algoritmo evolutivo para el aprendizaje del controlador borroso. Los programas compilados se guardarán junto con la versión del código fuente que los ha generado en el sistema de control de versiones Subversion descrito anteriormente.

Documentación

La documentación generada a lo largo de un proyecto software es de naturaleza muy variada. Además de los comentarios incluidos en el código fuente del programa, el lenguaje Java permite la generación de documentación asociada al código, mediante una herramienta llamada *Javadoc* [16]. El contenido de dicha documentación, que es una descripción detallada de todas las clases y métodos asociados a cada una de ellas, está insertado en el propio código fuente, por lo que cada versión estable contiene la documentación del código guardada en Subversion.

Además de la documentación relativa al código fuente, durante el desarrollo de un proyecto de software se genera más documentación, no a tan bajo nivel como la anterior. Por ejemplo la relativa a la estructura de la aplicación desarrollada, que se puede describir mediante diagramas en el lenguaje UML [13], así como a través de diagramas de arquitectura software. Esta documentación y la especificación de los requisitos del proyecto, así como el presente documento, que también forma parte de la documentación a alto nivel, se almacenará en un repositorio diferente al del código fuente.

Capítulo 6

Diseño e implementación

La etapa de diseño de software pretende analizar qué componentes son necesarios para afrontar el problema que se pretende solucionar. El proceso de diseño de un software es un proceso complicado que debe buscar una solución lo más escalable y modular posible para cumplir todos los requisitos que se han obtenido en la fase de análisis.

La fase de implementación busca, partiendo de la anterior, realizar una codificación lo más fiel posible a lo acordado. Es posible que durante esta etapa se produzcan errores que obliguen a modificar el diseño inicial, o que algunas de las ideas codificadas inicialmente resulten erróneas o inconvenientes para el proyecto, por lo que de esta fase pueden surgir modificaciones que afecten a las decisiones tomadas anteriormente. Es importante señalar que dichas modificaciones suelen ser pequeños cambios, de ningún modo a gran escala, y siempre en beneficio del cumplimiento de los requisitos del proyecto.

A lo largo del presente capítulo se presentará un diseño a alto nivel, correspondiente a la arquitectura del sistema, que permitirá contextualizar el entorno en el que se va a situar la herramienta de aprendizaje que es objetivo de este proyecto. Posteriormente se detallará un análisis a más bajo nivel de la aplicación que contenga la estructura del programa, detallando las clases que se implementarán y su relación entre ellas, utilizando los diagramas del lenguaje UML descrito en la sección 4.5.1.

6.1. Arquitectura del sistema

El objetivo de este proyecto es la construcción de un sistema de aprendizaje de un controlador borroso para un robot móvil que debe tener como prioridad llegar a un objetivo evitando todos los obstáculos intermedios. El proyecto no es, por lo tanto, una entidad de software independiente, sino que se coloca dentro de un entorno donde debe interactuar con diferentes componentes.

Para poder describir las interacciones y dependencias que tienen las diferentes partes del sistema entre sí, se utiliza un diagrama de arquitectura. Para poder utilizar el lenguaje UML seleccionado como herramienta de diseño se ha adaptado el diagrama de despliegue para explicar la arquitectura del sistema completo.

Como se puede ver en la figura 6.1, el sistema tiene dos niveles principales, los cuales se describen

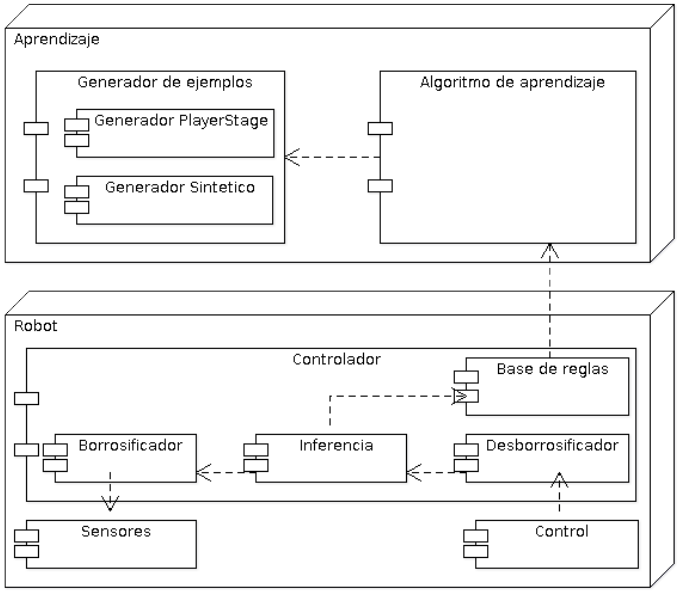

Figura 6.1: Diagrama de arquitectura del sistema.

detalladamente a continuación, identificando la función que tiene cada componente dentro del sistema:

- *Aprendizaje:* es el nivel donde se realiza el proceso de aprendizaje de la base de reglas. Como se puede comprobar se distinguen dos componentes con funciones bien diferenciadas:
  - *Generador de ejemplos:* es un componente que se encarga de generar los ficheros de ejemplos que serán utilizados como base para la ejecución del algoritmo de aprendizaje. Este componente se divide en dos internamente, uno que realiza una conversión de formato desde los logs de Player/Stage al formato de fichero de ejemplos del programa, y otro que genera un fichero de ejemplos sintéticamente.
  - *Algoritmo de aprendizaje:* en base al fichero de ejemplos obtenido del componente descrito anteriormente, este componente es el algoritmo evolutivo que permitirá obtener una base de conocimiento que modele el comportamiento que debe tener el controlador del robot.
- *Robot:* este es el nivel donde se aplica el conocimiento contenido en la base de reglas obtenida del anterior. Se divide en tres componentes fundamentales para permitir llevar a cabo el control del robot:
  - *Sensores:* es necesario que uno de los componentes lleve a cabo la recogida de datos del entorno del robot. Esto se puede realizar de modo inmediato mediante un conjunto de sensores, que pueden ser de diferentes tipos, láser, sonar, etc. En el caso de este proyecto se utilizará un sensor láser que abarque toda la parte delantera del robot.

- *Controlador:* es el elemento más complejo de esta parte de la arquitectura, pues contiene varios componentes que son los que deciden qué control se le va a enviar al robot con los datos de entrada recogidos mediante los sensores. Contiene varios subcomponentes que realizan trabajos bien diferenciados:
  - *Borrosificador:* dados los datos obtenidos por los sensores del robot, aplica un proceso sobre ellos para obtener las entradas sobre las que aplicar el proceso de inferencia.
  - *Base de reglas:* modela el comportamiento aprendido por el robot, como salida del algoritmo evolutivo que se ejecuta en el primer nivel. Es aplicado por el motor de inferencia para obtener las salidas que corresponden a unas determinadas entradas.
  - *Inferencia:* es un componente que aplica el conocimiento que modela la base de reglas sobre las entradas que proporciona el componente desborrosificador. La base de reglas responde a los datos introducidos activando una serie de reglas, que tienen como respuesta unos datos de control especificados por sus consecuentes. El motor de inferencia busca en qué medida cada regla de la base de conocimiento se activa ante unos determinados datos de entrada.
  - *Desborrosificador:* La salida de este componente es el control que se debe enviar al robot para los datos que han recogido los sensores.
- *Control:* así como es necesario un componente que trate los datos del entorno del robot para producir unas entradas al controlador, también es necesario otro que aplique sobre el robot los comandos de salida que se obtienen como respuesta para poder efectuar la orden de movimiento obtenida. Este componente efectúa la comunicación con los efectores del robot para aplicar los comandos de control que se han obtenido.

6.2. Patrones de diseño

Un patrón de diseño proporciona una base para solucionar problemas comunes que pueden darse en proyectos de software [15]. Es una solución a un problema de diseño que tiene ciertas características que hacen muy beneficiosa su utilización siempre que sea posible:

- Proporcionan una solución *eficaz* que ha sido probada en numerosas situaciones.
- Son *reutilizables*, de tal forma que tienen cabida en la resolución de problemas de diseño de diferente índole y en distintas circunstancias.

La utilización de patrones de diseño dentro de un proyecto software significa además un ahorro de tiempo al poder recurrir a ellos como una fuente de soluciones ya probadas en diferentes ámbitos y que son adaptables a la naturaleza de cualquier problema concreto. Su uso tiene además la ventaja de estandarizar las soluciones que se proponen como diseño de un proyecto de software.

Abusar de ellos o imponerlos aún cuando su uso no está debidamente justificado para una situación concreta es un error, por lo que su aplicación debe realizarse siempre con cuidado. En este proyecto se han utilizado una serie de patrones, los cuales se describen con detalle a continuación.

6.2.1. *Singleton*

Es un patrón de diseño creacional, que establece una limitación en cuanto al método en el que se puede instanciar una clase. Su objetivo consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

La clase que sobre la que se desea implementar este patrón debe facilitar un método de creación de la instancia, si todavía no se ha creado. El constructor debe ser accesible sólo desde la propia clase para evitar la instanciación en más de una ocasión. El patrón *Singleton* provee una única instancia global gracias a que, por una parte, la propia clase es responsable de crear la única instancia; se permite el acceso global a dicha instancia mediante un método de clase, o haciéndola accesible mediante un atributo estático, y se declara el constructor de la clase como privado para que no sea instanciable directamente.

La utilización de este patrón de diseño permite además ahorrar tiempo en instanciación de diferentes clases, cuando es un paso que puede ahorrarse, ya que por diferentes motivos, la realización de dicha operación siempre arroja el mismo resultado, por ejemplo porque la clase carece de atributos modificables. Esta es una situación que se repetirá en algunas ocasiones a lo largo de este proyecto, como se puede comprobar en los diagramas estructurales detallados más adelante. Todas las clases que implementen este patrón estarán marcadas con el estereotipo *Singleton* para identificarlas fácilmente. La implementación de este patrón se detalla en la figura 6.2.

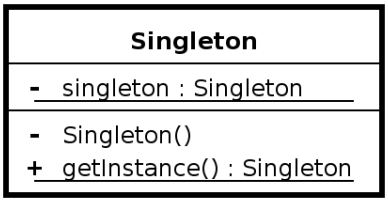

Figura 6.2: Implementación del patrón de diseño *Singleton*.

6.2.2. *Observer*

Es un patrón de comportamiento que define una dependencia entre varias clases de objetos, de manera que cuando uno de ellos, el *observado*, modifica su estado, se encarga de notificar este cambio a todos los otros dependientes, que están suscritos al primero y a la espera de que se produzcan. El objetivo de este patrón es desacoplar la clase de los objetos que están suscritos a la variación del estado del observado, aumentando la modularidad del lenguaje, así como evitar bucles de actualización que lo único que hacen es comprobar el estado del objeto buscando cambios.

De esta manera, en lugar de ser los observadores los que buscan las modificaciones en los objetos a los que observan, se invierte el paradigma utilizado, que hace que sean estos últimos los que notifiquen de los cambios a los objetos suscritos a ellos.

En este problema, esto es especialmente útil en el caso de las clases de la interfaz gráfica que

desean reflejar el estado en el que se encuentran los objetos de la aplicación, por ejemplo la variación del *fitness* de la población. En lugar de tener que realizar manualmente las comprobaciones sobre el estado de los objetos para realizar los cambios sobre las clases de la interfaz, son los objetos los que notifican a la interfaz que debe actualizarse para reflejarlos.

La utilización de este patrón se lleva a cabo tanto en la herramienta de generación de ejemplos, como en la herramienta de aprendizaje, y se lleva a cabo mediante las clases *Observer* y *Observable* de Java, que están diseñadas explícitamente para la implementación de este patrón, cuya estructura se muestra en la figura 6.3.

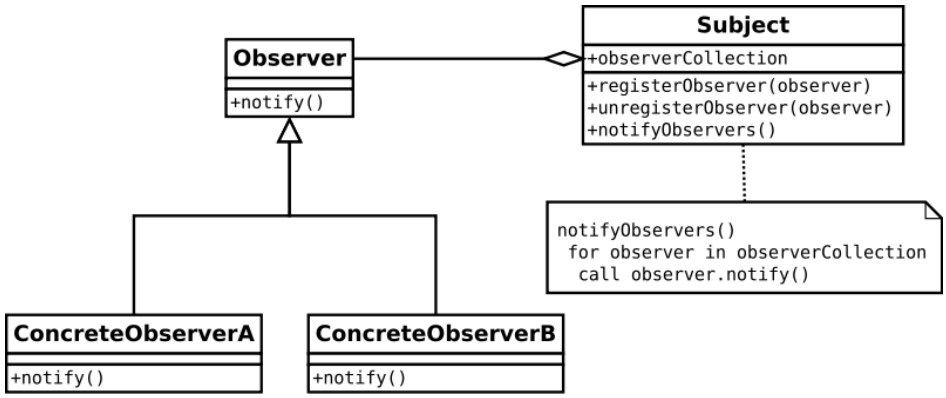

Figura 6.3: Estructura de la implementación del patrón de diseño *Observer*.

6.2.3. *Strategy*

Al igual que el anterior, es un patrón de comportamiento, que en este caso permite mantener un conjunto de algoritmos de los que el objeto cliente puede elegir aquel que le conviene e intercambiarlo según sus necesidades. Lo interesante de esta acción es que puede realizarse en tiempo de ejecución, sin necesidad de realizar modificación alguna sobre el código.

La implementación de este patrón de diseño puede llevarse a cabo mediante la definición de una interfaz o una clase abstracta que defina el tipo de objeto que se a a utilizar en la aplicación, y posteriormente la definición de varias clases que utilicen a ésta, realizando diferentes implementaciones de los métodos que contiene.

En tiempo de ejecución puede decidirse cuál de las subclases que implementan la interfaz se va a utilizar para realizar las operaciones, y cuando se acceda ellas, se hará siempre a través de la interfaz, por lo que cualquier objeto que la implemente puede ser utilizado para realizar la misma función. Esta es una funcionalidad muy interesante, tanto para el generador de ejemplos sintéticos, donde hay varios algoritmos de reducción de ejemplos que se pueden utilizar, como para el algoritmo evolutivo, donde sería muy interesante que tanto los operadores como, por ejemplo, la función de evaluación, fuesen seleccionados entre varias alternativas *en caliente*, sin necesidad de realizar modificaciones sobre el código.

En la figura 6.4 se observa la estructura que tiene este patrón de diseño. En ella se puede ver que las distintas estrategias utilizadas dependen de un contexto, que puede contener una o varias de ellas,

este puede ser por ejemplo, en nuestro caso, el algoritmo evolutivo, que contiene diferentes estrategias, u operadores genéticos, que a su vez pueden tener diferentes implementaciones.

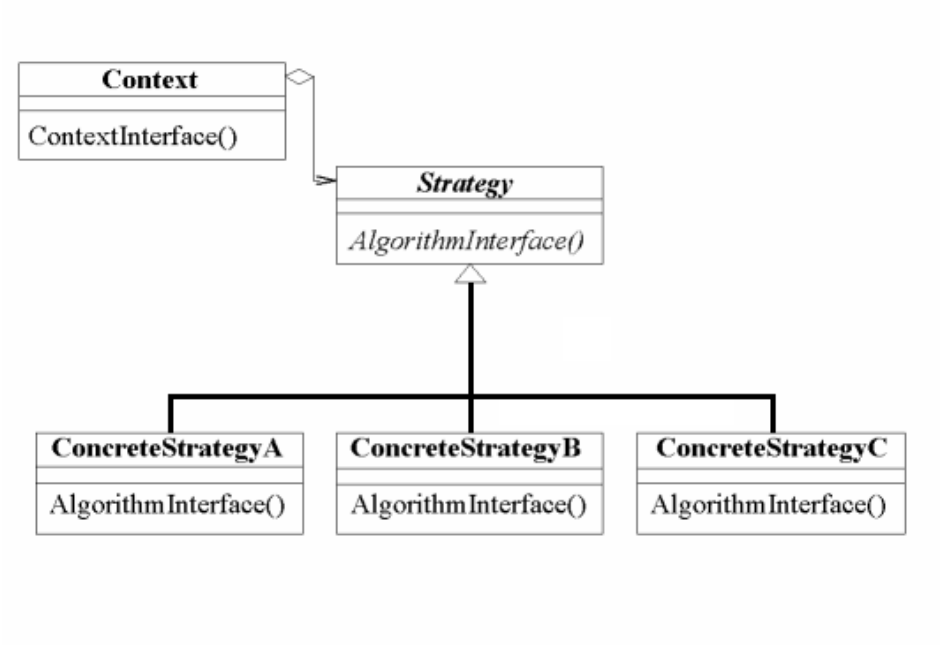

Figura 6.4: Estructura de la implementación del patrón de diseño *Strategy*.

6.2.4. *Builder*

Este es un patrón de diseño creacional que abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes. Su estructura se refleja en la figura 6.5. Está directamente relacionado con el patrón *Composite*, descrito en la sección 6.2.5.

La utilización de este patrón dentro de este proyecto es interesante para generalizar la creación del conjunto del algoritmo evolutivo que se utilizará para el aprendizaje. Cada una de las partes que lo componen implementa un patrón estrategia, definido en la sección 6.2.3, por lo que para la creación del algoritmo evolutivo en su totalidad habrá que realizar la instanciación de las diferentes implementaciones de cada uno de los operadores según las necesidades definidas por el usuario en cada ejecución del programa.

En este patrón, el objeto que implementa el *Builder* va llamando a los constructores necesarios para crear el producto final, de una forma completamente transparente al usuario, quien lo que recibe a la salida de la ejecución de éste es un producto utilizable en su totalidad. Al generalizarse la creación del objeto en un único punto, se produce un mayor control sobre las operaciones realizadas, además de abstraer el proceso de construcción del algoritmo evolutivo a una clase independiente de éste. Como el propio algoritmo evolutivo implementa en sí mismo el patrón estrategia, la abstracción del proceso de creación de este objeto permite su utilización con otras implementaciones que utilicen la interfaz que implementa el algoritmo.

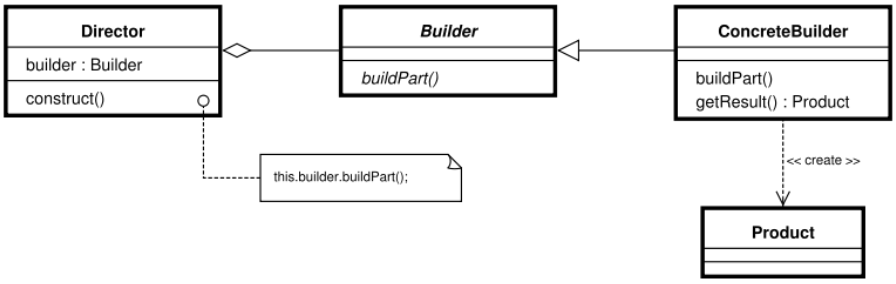

Figura 6.5: Estructura de la implementación del patrón de diseño *Builder*.

6.2.5. *Composite*

El patrón *Composite* permite la creación de objetos complejos a partir de otros más simples, y similares entre sí, gracias a la composición recursiva y a la estructura en forma de árbol. Como todos los objetos poseen una interfaz común, pueden ser tratados de la misma manera, tal y como se detalla en la figura 6.6.

Cada uno de los nodos que forma el árbol puede ser un objeto compuesto que, a su vez, contenga más nodos, o bien representar un nodo hoja, que es una implementación distinta, y que representa cada uno de los elementos que están formando parte del objeto compuesto.

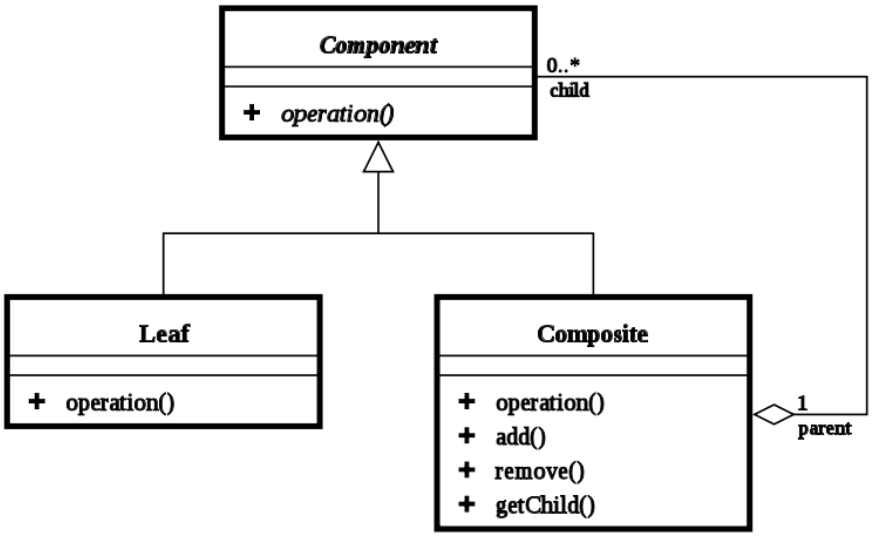

Figura 6.6: Estructura de la implementación del patrón de diseño *Composite*.

En este problema concreto, se ha utilizado el patrón *Composite* para realizar el almacenamiento de los conjuntos de etiquetas borrosas, que se detalla más adelante, en la sección 6.3.2.

6.3. Diseño detallado

Una vez que se ha definido la arquitectura del sistema y se conoce la relación de los módulos a implementar, es posible abordar un diseño a más bajo nivel que permita abordar todos los requisitos del proyecto que se han definido en la sección 4.1. Además de atenerse a las restricciones especificadas por los requisitos, el diseño deberá buscar la mayor independencia entre las partes del programa, lo que permitirá afrontar con una mayor facilidad posibles cambios en los requisitos, como ya se ha detallado en la sección 5.1; hacer más extensible el sistema y proporcionar una solución lo más óptima posible en términos de calidad.

A lo largo de la siguiente sección se detallará y justificará el diseño de cada uno de los módulos de los que se debe realizar la implementación, explicando el diseño de su estructura mediante un diagrama de clases donde se justifiquen las decisiones de diseño que se han tomado.

6.3.1. Generador de ejemplos

Este componente tiene la función de generar ficheros de ejemplos para el proceso de aprendizaje. Este componente puede implementar diferentes formas de generar los ficheros de ejemplos, como de hecho aparece reflejado en los requisitos del proyecto. Cada implementación diferente de la generación de este fichero será un subcomponente de este. A priori en los requisitos se detallan dos formas de abordar la implementación de esta funcionalidad, por lo que serán necesarios dos subcomponentes del generador de ejemplos, como aparece detallado en el diseño de la arquitectura en la sección 6.1.

Intérprete de logs de Player/Stage

Este componente se encarga de traducir los datos que contiene un fichero de log de la herramienta de simulación Player/Stage a un fichero de ejemplos utilizable por la herramienta de aprendizaje. Sobre los datos de entrada en bruto hay que realizar un procesamiento de la información para llegar a obtener los datos necesarios para formar un ejemplo. No es una transformación de información inmediata, puesto que el procesamiento requiere la integración de la información de mediciones de los láser y realizar cálculos sobre la posición del robot y el punto objetivo para poder calcular la distancia y el ángulo al que se encuentra del robot en un momento determinado.

El diseño de este componente se ha enfocado al cumplimiento de todas las funcionalidades que debe cumplir obligatoriamente. El diagrama de clases con la estructura de este componente descrita de modo detallado se puede ver en la figura 6.7.

Se ha producido un reparto de responsabilidades entre las diferentes clases que lo conforman, de modo que la modularidad entre las distintas partes sea máxima y esté bien definida. A continuación se define el rol que dentro del diseño tiene cada una de ellas:

- *DataProcesser:* esta es la clase encargada de realizar las transformaciones necesarias sobre los datos de entrada para convertirlos al formato de ejemplo adecuado. Internamente realiza las operaciones que transforman las mediciones de los láser en bruto y los datos de posición del robot para obtener las variables del antecedente de la base de reglas definida en los requisitos.

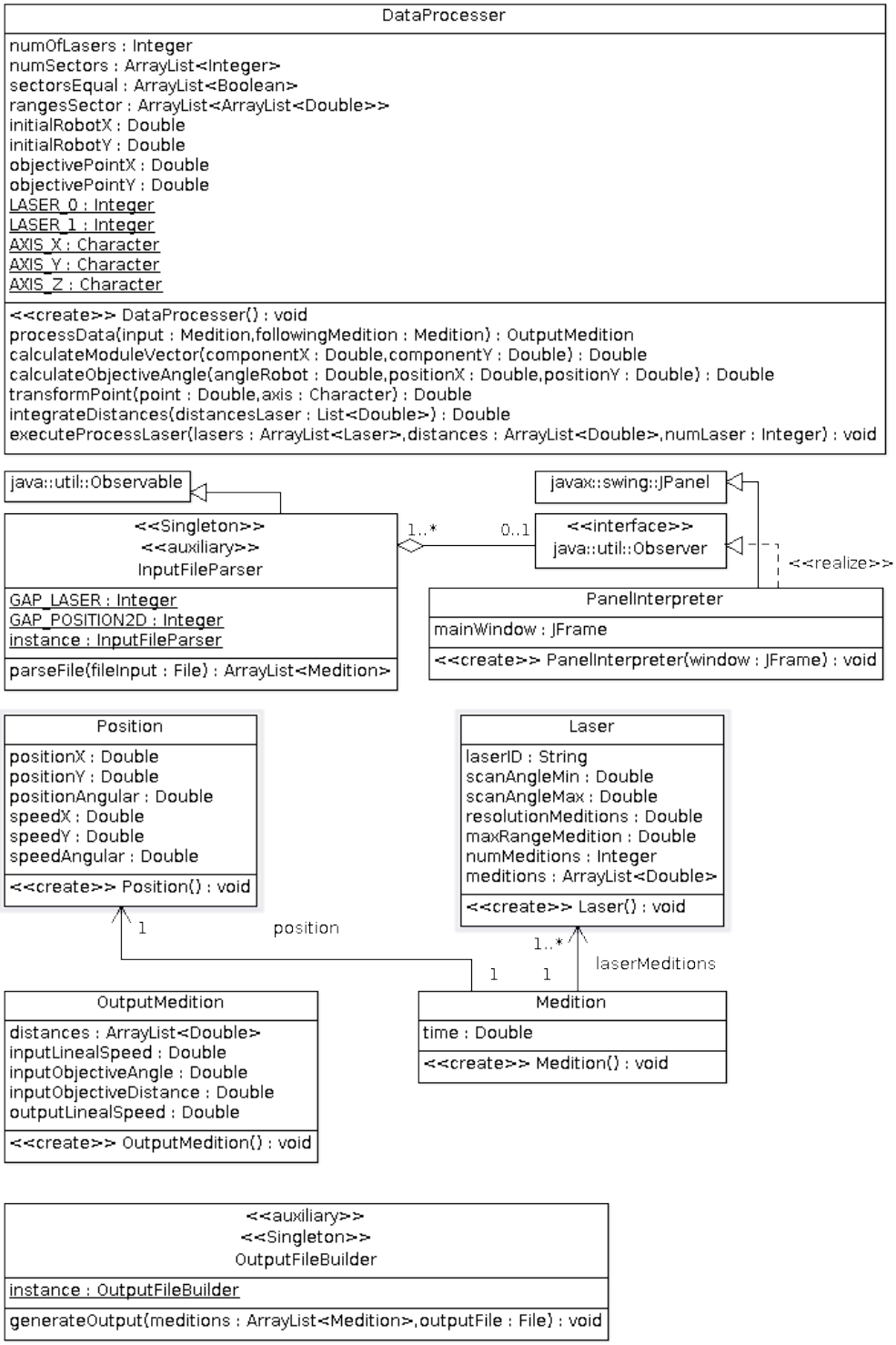

Figura 6.7: Diagrama de clases para el intérprete de logs de Player/Stage.

- *InputFileParser:* es una clase auxiliar que toma los datos del archivo de log de Player/Stage y los divide en estructuras conocidas por la aplicación y con las que es capaz de trabajar. Esta clase está involucrada dentro del patrón de diseño *Observer* y es una clase que hereda de *java.util.Observable*. También implementa el patrón *Singleton*, por lo que sólo habrá una posible instancia de esta clase en el programa.

- *PanelInterpreter:* es una de las clases de la interfaz gráfica, que se muestra en el diagrama para mostrar de qué modo se ha modelado la utilización del patrón *Observer*. Es la clase que observa los cambios que se han hecho sobre la anterior para reflejarlos gráficamente al usuario. Para cumplir con sus funciones implementa la interfaz *java.util.Observer* y extiende a *javax.swing.JPanel*.

- *Position:* es una de las clases que se utilizan para modelado de datos. Representa el estado en el que se encuentra el robot en un momento determinado, para lo que almacena datos sobre las coordenadas en las que se posiciona, la orientación y la velocidad, tanto lineal como angular.

- *Laser:* es otra clase del modelo de datos que utiliza la aplicación. Contiene toda la información sin procesar de un sensor láser en un momento determinado, tanto la información sobre distancias como el número de mediciones, rango de barrido del láser, etc.

- *Medition:* esta es la clase que sirve como entrada a *DataProcesser*. Contiene todos los datos que corresponden a un instante determinado.

- *OutputMedition:* una vez que han sido procesados, los datos en bruto del log de Player/Stage son transformados en unos adecuados para su trabajo con la base de reglas que se ha definido en los requisitos. *OutputMedition* contiene toda la información que se necesita para constituir un ejemplo con el que trabaja la herramienta de aprendizaje.

- *OutputFileBuilder:* al igual que la clase *InputFileParser*, que procesa el archivo de entrada para generar los objetos con los que trabaja el procesador de datos, esta clase toma como entrada una lista de objetos *OutputMedition* y los escribe en un fichero para generar un conjunto de ejemplos adecuados para la herramienta de aprendizaje.

En un instante determinado, el fichero de logs de Player/Stage contiene varias líneas que detallan la siguiente información:

```
tiempo [... x 2] "position2d" [... x 3] pos:x pos:y pos:yaw vel:x vel:y vel:yaw stall

tiempo [... x 2] "laser" id [... x 3] ang:min ang:max ang:resolution dist:max N
[N x [medition flag]]
```

donde: *pos* representa la posición actual del robot, en metros para las distancias lineales, y en radianes para las el ángulo; *vel* representa la velocidad del robot medida en m/s para las velocidades lineales, y en rad/s para la velocidad angular; *ang* representa la información sobre el rango que explora el medidor láser, y *dist* el rango máximo que puede alcanzar una medición. Por su parte, N es el número de mediciones que contiene ese sensor.

Hay que destacar que, en el formato escrito arriba, para cada instante determinado se escribe una línea para determinar la posición del robot, y luego una línea por cada sensor cuya información se

describe. En este caso concreto, al tener sólo un sensor láser, cada instante vendrá determinado por la línea de la posición y una línea para describir la medición del láser.

El formato que debe tener un ejemplo es el siguiente:

```
dist:frontal dist:dcha dist:izda obj:distancia obj:angulo vel
conseq:vel_lineal conseq:vel_angular
```

donde en este caso *dist* representa la distancia del robot a los obstáculos; *obj* representa la información del robot respecto al objetivo, y *vel* es la velocidad lineal en el antecedente. Para el consecuente se añaden los campos etiquetados como *conseq*, que representan la velocidad lineal y angular.

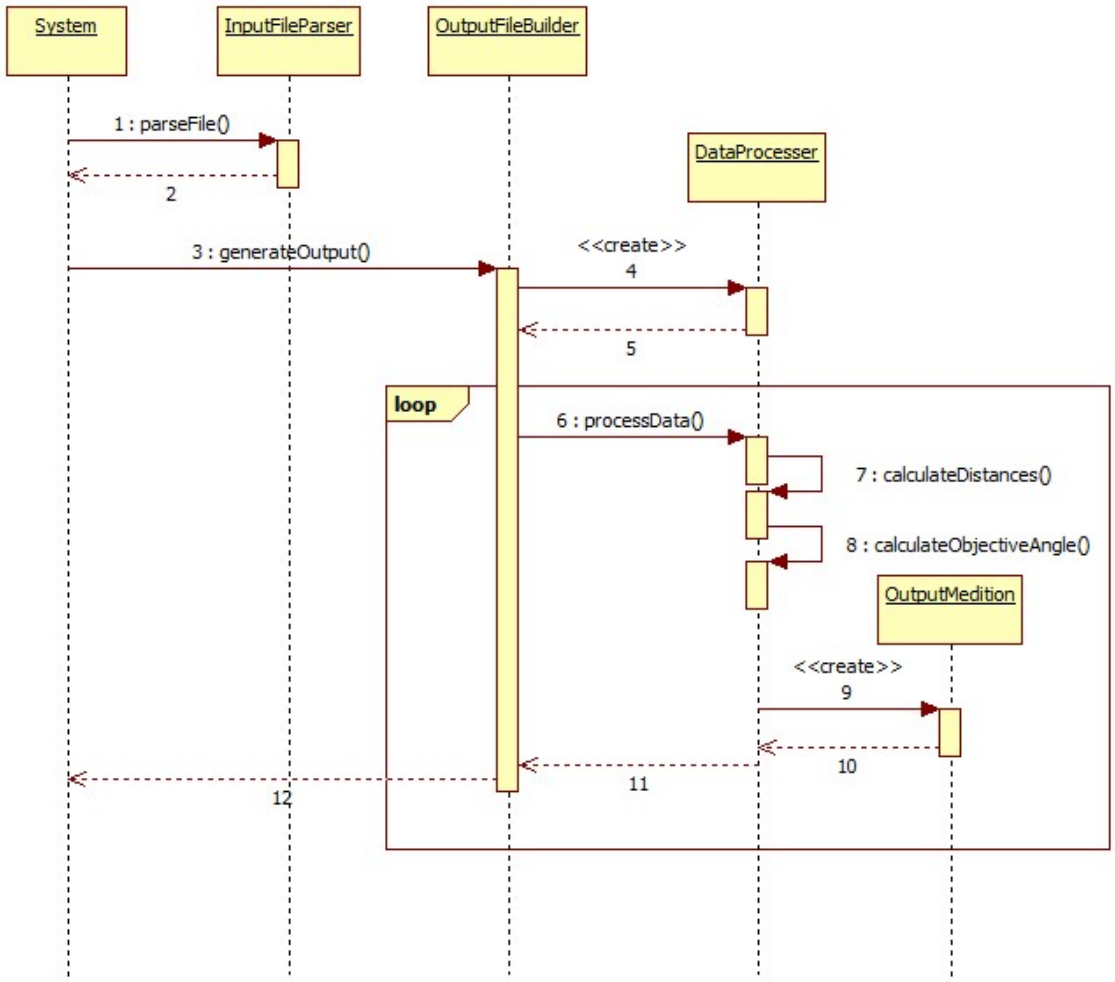

Figura 6.8: Diagrama de secuencia para las operaciones realizadas por el componente intérprete de logs de Player/Stage.

En la figura 6.8 se puede ver un diagrama de secuencia que detalla el orden en el que se realizan las operaciones para la ejecución del intérprete de logs de Player/Stage sobre un archivo de entrada.

En él se puede ver cómo en primer lugar se realiza la operación del lectura sobre el fichero fuente de datos, y posteriormente se llama a la clase encargada de la construcción del fichero de salida. Esta clase será la encargada de instanciar a la que procesa los datos de entrada y realiza transformaciones sobre ellos, e itera sobre cada una de las mediciones que se han leído del log de Player/Stage para realizar los cálculos correspondientes y escribir los datos transformados en el archivo de salida.

Generador de ejemplos sintéticos

En este caso, este componente tiene como misión la generación de un fichero de ejemplos, aunque sin una base como era el caso anterior. Para la realización de un fichero de ejemplos sintéticamente se debe disponer de la información de los rangos de las variables de entrada y de los niveles en los que se quiere discretizar ese espacio. Debido a que el número de ejemplos que genera este componente es demasiado elevado, es aquí donde se debe realizar la aplicación de un algoritmo de reducción de ejemplos. El diseño de clases del generador sintético de ejemplos se detalla en la figura 6.9.

Cuando se ha procedido a la implementación de la reducción del número de ejemplos según el requisito RF-15, que se ha detallado en 4.2, se han detectado una serie de problemas. A pesar de que la idea inicial del algoritmo parece adecuada para este proyecto, finalmente su aplicación ha resultado en una pérdida significativa de calidad del conjunto de ejemplos final, debido a la eliminación excesiva de ejemplos del conjunto inicial, que no permitía un aprendizaje de calidad. Por tanto, a la hora de la implementación se utiliza otra idea para abordar el cumplimiento de este requisito, que se describe en esta misma sección.

A continuación se detalla el papel que cada clase tiene dentro del diseño de este componente:

- *SinteticDataGenerator:* es la clase que implementa la funcionalidad central de este componente, que es la generación de todos los ejemplos posibles dados los rangos de las variables de entrada y los intervalos de discretización definidos por el usuario. La funcionalidad principal que implementa es la generación de todos los antecedentes posibles, dejando la elección de los consecuentes en manos de otras clases.

- *SinteticDataEvaluator:* dado un ejemplo sintético generado, halla una puntuación de su antecedente con un consecuente dado. Esta operación sirve como base para encontrar el mejor consecuente posible para un ejemplo, buscando aquel que mejor puntuación obtenga de todo el conjunto de posibles consecuentes.

- *ExamplesReductor:* es la interfaz que deben implementar todas las clases que deseen realizar la función de reducir el número de ejemplos de un conjunto. Se utiliza como parte de la implementación del patrón de diseño Estrategia, para poder añadir cualquier clase que implemente a esta interfaz al diseño sin necesidad de añadir más cambios.

- *ExamplesReductionBeams:* contiene el código necesario para ejecutar la reducción de ejemplos tal y como se describe en la sección 3.2.2. Solo hay posibilidad de que haya una instancia de esta clase en el programa, por lo que implementa el patrón de diseño *Singleton* para modelarlo.

- *ExamplesReductionIB2:* contiene la codificación de la idea inicial respecto a la reducción de ejemplos, tal y como se describe en el algoritmo descrito en la sección 3.2.2. Al igual que en el

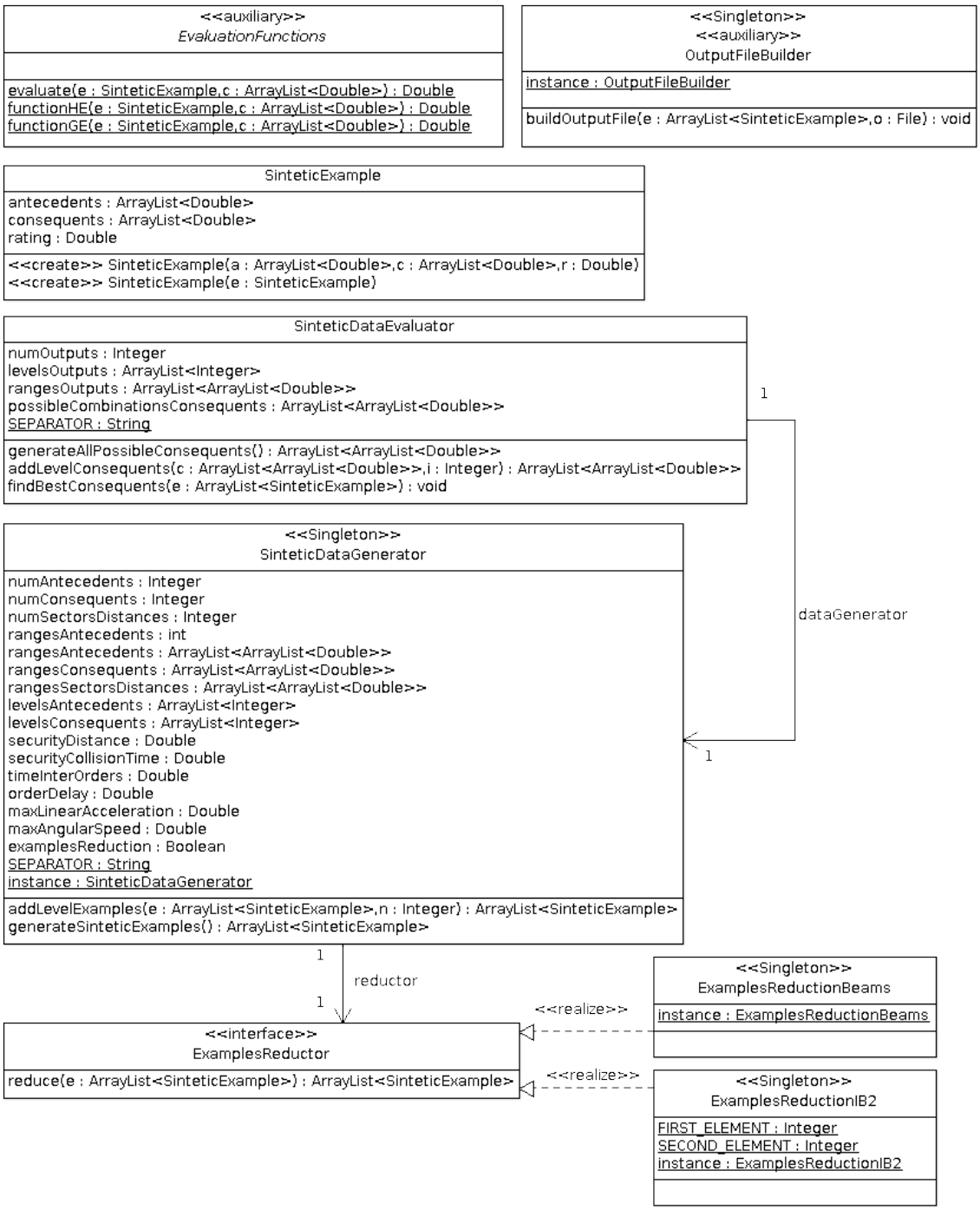

Figura 6.9: Diagrama de clases que detalla la estructura del generador de ejemplos sintéticos.

caso anterior, implementa el patrón *Singleton* para modelar esta situación.

- *SinteticExample:* es una clase de modelado de datos. Contiene tanto los valores del antecedente como del consecuente para un ejemplo determinado. También tiene la posibilidad de almacenar

la puntuación que ese ejemplo ha recibido según la función de evaluación.

- *EvaluationFunctions:* es una clase que funciona como un *Helper* de la clase *SinteticDataEvaluator*, y que contiene todas las funciones que se utilizan para asignar a una combinación de antecedentes y consecuentes una puntuación. Se define como una clase abstracta, no necesita instanciarse para realizar su función, ya que sólo depende de los parámetros que se le transmiten.
- *OutputFileBuilder:* es una clase *Helper* al igual que la que contiene las funciones de evaluación. Sólo se requiere una instancia de la misma para poder cumplir con la función para la que fue diseñada, que es la escritura de un conjunto de ejemplos en un fichero con el formato de entrada adecuado para la herramienta de aprendizaje, por lo que implementa el patrón *Singleton*.

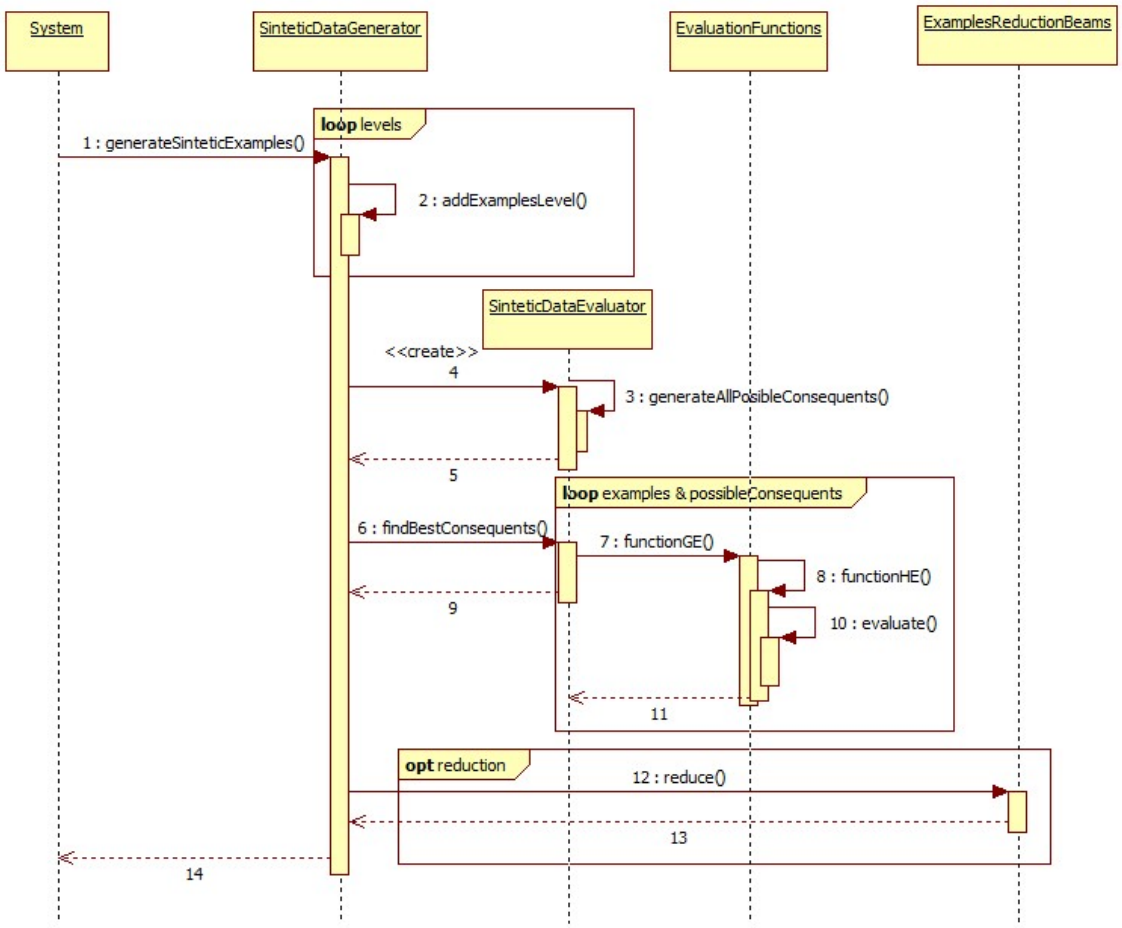

Figura 6.10: Diagrama de secuencia para las operaciones realizadas por el componente de generación de ejemplos sintéticos.

En la figura 6.10 se puede visualizar de modo gráfico mediante un diagrama de secuencia cuál es el orden en el que se realizan las operaciones para generar el conjunto de ejemplos sintéticos. En primer lugar se generan todos los posibles antecedentes de los ejemplos, mediante un método que se ejecuta tantas veces como variables haya en el antecedente de los ejemplos. Posteriormente se generan

todas las posibles combinaciones de consecuentes, y se realiza la asignación a cada antecedente del consecuente que mejor se adapte a él, de la forma en la que se describe en las funciones de puntuación de los ejemplos.

Opcionalmente, si el usuario de la aplicación así lo decide, se puede utilizar un algoritmo de reducción del número de ejemplos, que se describieron en la sección 3.2.2.

6.3.2. Algoritmo de aprendizaje

Es el componente de más complejidad de cuantos participan en el diagrama de arquitectura descrito en la sección 6.1. Se encarga de llevar a cabo la implementación del algoritmo evolutivo. Debido a la complejidad de su diseño, es mucho más claro analizarlo por partes: el modelo de datos y la lógica de negocio por separado. Se considera modelo de datos a todas aquellas clases cuya función principal no es la introducción de nuevas funcionalidades al programa, sino modelar el conjunto de objetos con los que va a trabajar el algoritmo. Bajo el concepto de lógica de negocio se engloba el conjunto de código que implementa las funcionalidades del sistema, esto es, el que realiza las operaciones. Hay que tener en cuenta que las clases de modelado de datos pueden tener lógica de negocio incluida, aunque no es la finalidad principal de la clase. El modelo de datos se muestra en la figura 6.11.

A continuación se detalla para cada clase en el modelo qué es lo que representa, así como las posibles funcionalidades que están implementadas en cada una de ellas:

- *ProblemPopulation:* representa la población con la que trabaja el algoritmo evolutivo. Son variables de la clase, no de instancia, la iteración de la población y el ID del último individuo que se ha añadido, para evitar duplicidades entre ellos. Esta clase contiene una colección de instancias de la clase *ProblemIndividual*, que son los que forman la población. Implementa la interfaz *Population* de la librería JEVA.

- *ProblemIndividual:* en el modelo de datos es la representación de un individuo de la población, contiene información sobre el genotipo, *ProblemGenotype*, y el fenotipo, *ProblemPhenotype*. Cada individuo tiene unos valores de evaluación, o *fitness*, un identificador y una *historia* contenida en una instancia de la clase *History*. Como lógica de negocio almacena el análisis sobre si dicho individuo cubre o no un determinado ejemplo, y el borrado de las reglas que no son disparadas por ninguno de los ejemplos en el conjunto de entrenamiento. Implementa la interfaz *Individual* de JEVA.

- *History:* representa toda la información sobre las transformaciones que un individuo puede sufrir desde que se crea hasta que finalizan las operaciones que se realizan sobre él. En concreto, puede almacenar información sobre el cruce y la mutación, en qué momento se creó, y un campo de libre formato donde se puede colocar cualquier información adicional que se desee.

- *ProblemGenotype:* es la implementación de la interfaz *Genotype* de JEVA para este problema concreto. La información que contiene para modelar el genotipo de un individuo es una colección de instancias de la clase *CodedRule*, por lo que en realidad lo que contiene es una base de reglas codificada.

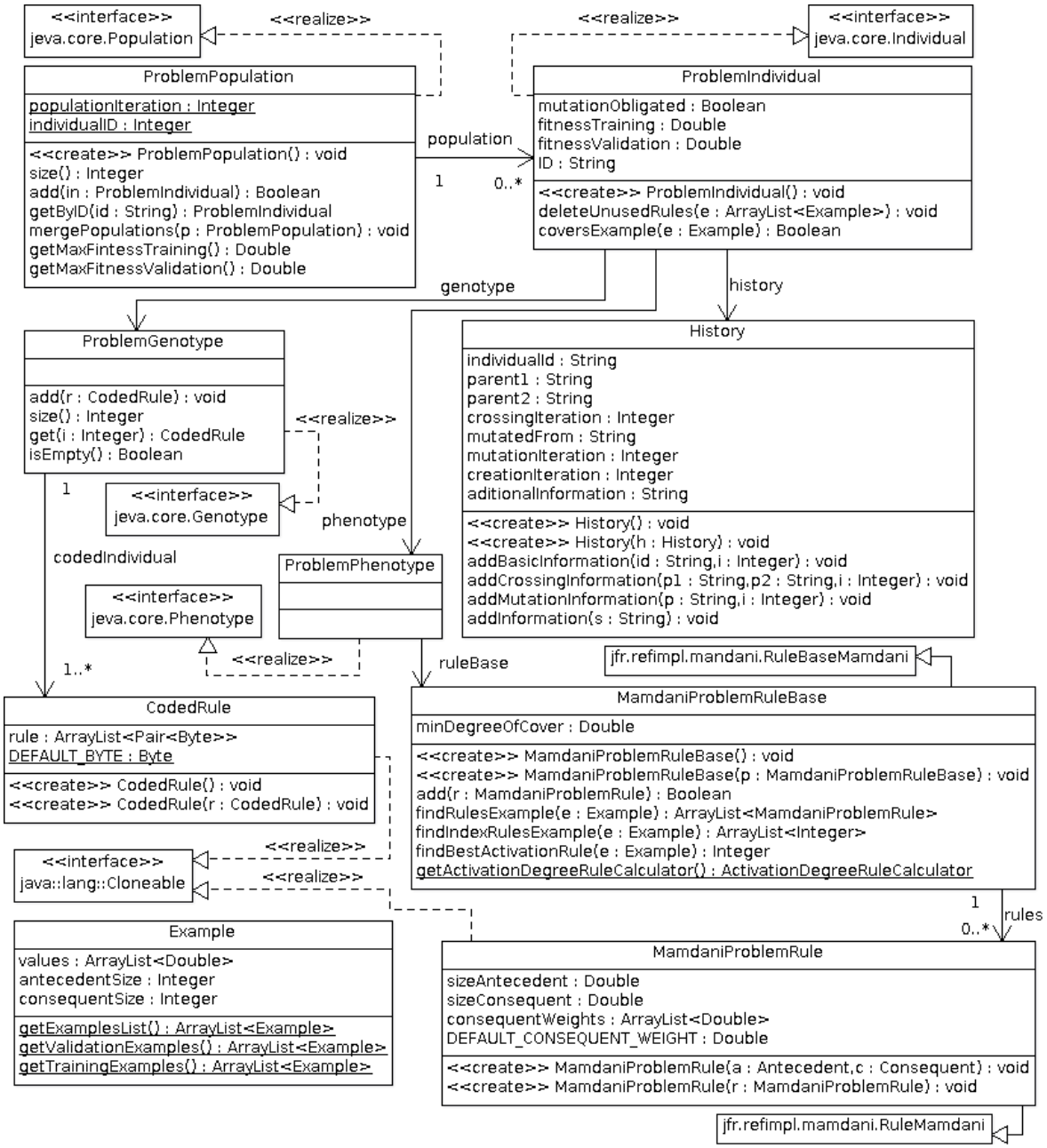

Figura 6.11: Diagrama de clases para el modelo de datos del algoritmo de aprendizaje.

- *CodedRule:* representa una regla codificada de forma que cada variable del antecedente o del consecuente solo ocupe una pareja de bytes. Se utiliza esta forma codificada para la inicialización y operaciones que tengan que realizar modificaciones sobre la base de conocimiento.

- *ProblemPhenotype:* se corresponde con la implementación del fenotipo de un individuo para este problema concreto, para lo que realiza la interfaz *Phenotype* de JEVA. Su contenido es una base de reglas representada por la clase *MamdaniProblemRuleBase*.

- *MamdaniProblemRuleBase:* contiene una colección de instancias de la clase *MamdaniProblemRule*. Representa una base de reglas, no codificada como en el caso anterior. Es una extensión de la clase *RuleBaseMamdani* de la librería JFR, que no se ha utilizado de modo directo para añadir algunos métodos que faciliten la lógica de negocio desde la propia clase que codifica la base de reglas. En ella se encuentran métodos para buscar las reglas que se activan con un determinado ejemplo.

- *MamdaniProblemRule:* representa una regla borrosa de la base de conocimiento. Esta regla se encuentra definida por una serie de conjuntos borrosos. La implementación de la esta clase hereda de *RuleMamdani* de la librería JFR, que se ha extendido para poder añadir la característica a la regla de que cada consecuente pudiese tener un peso diferente, así como información sobre la composición de la regla en la propia instancia que la contiene.

- *Example:* contiene toda la información que el ejemplo necesita para ser utilizado dentro de la aplicación, esto es, una lista de valores que lo forman, y una especificación del formato (tamaño del antecedente y del consecuente). Contiene también algunos métodos de lógica de negocio, como es la carga de fichero de toda la lista de ejemplos, o sólo la obtención del conjunto de entrenamiento y de validación.

Conocido el modelo de datos, es posible ahora describir el diseño de la lógica de negocio de un modo más sencillo, pues se conocen ya los objetos con los que la aplicación debe trabajar. En este proyecto la lógica de negocio esta formada por el propio algoritmo evolutivo y todos sus operadores, como se puede ver en la figura 6.12. A pesar de que hay otras clases que contienen funcionalidades de la aplicación, se consideran dentro de la categoría de clases auxiliares, y serán descritas más adelante.

En términos de diseño, además de la función de evaluación inicialmente descrita en los requisitos, es necesario implementar la nueva que se ha descrito en la sección 3.9. A continuación se describen para cada una de las clases, representadas en la figura 6.12, qué aportaciones realizan al modelo de la lógica de negocio, así como el rol que cada una de ellas toma respecto de las demás:

- *Algorithm:* es una interfaz utilizada para definir el algoritmo que, sea cual sea su implementación, siempre deberá tener al menos un método que permita su ejecución, y los parámetros que se definen en ella, que sirven para definir la condición de parada que tendrá.

- *EvolutionarySystem:* implementa la interfaz anterior, es la clase principal del programa, pues de ella depende la ejecución de la totalidad del algoritmo, mediante llamadas a métodos al resto de clases. Es en esta clase donde se organiza la ejecución de todo el código, se realizan las llamadas a los diferentes operadores, y se decide el flujo de ejecución principal del programa.

- *EvolutionarySystemBuilder:* utilizado para la implementación del patrón de diseño *Builder*, su principal cometido es la creación de instancias de la clase *EvolutionarySystem*, utilizando una serie de parámetros para la construcción que son leídos de ficheros donde se guardan las opciones para personalizar la ejecución del algoritmo de aprendizaje. En estos ficheros se detallan, por ejemplo, los diferentes operadores que se desea utilizar. En base a su contenido se crea el algoritmo a ejecutar, de forma completamente dinámica.

- *IndividualCrosser:* es la clase que contiene el código necesario para el cruce entre individuos. Implementa la interfaz *Crosser* de la librería JEVA, aunque además de cumplir la especificación

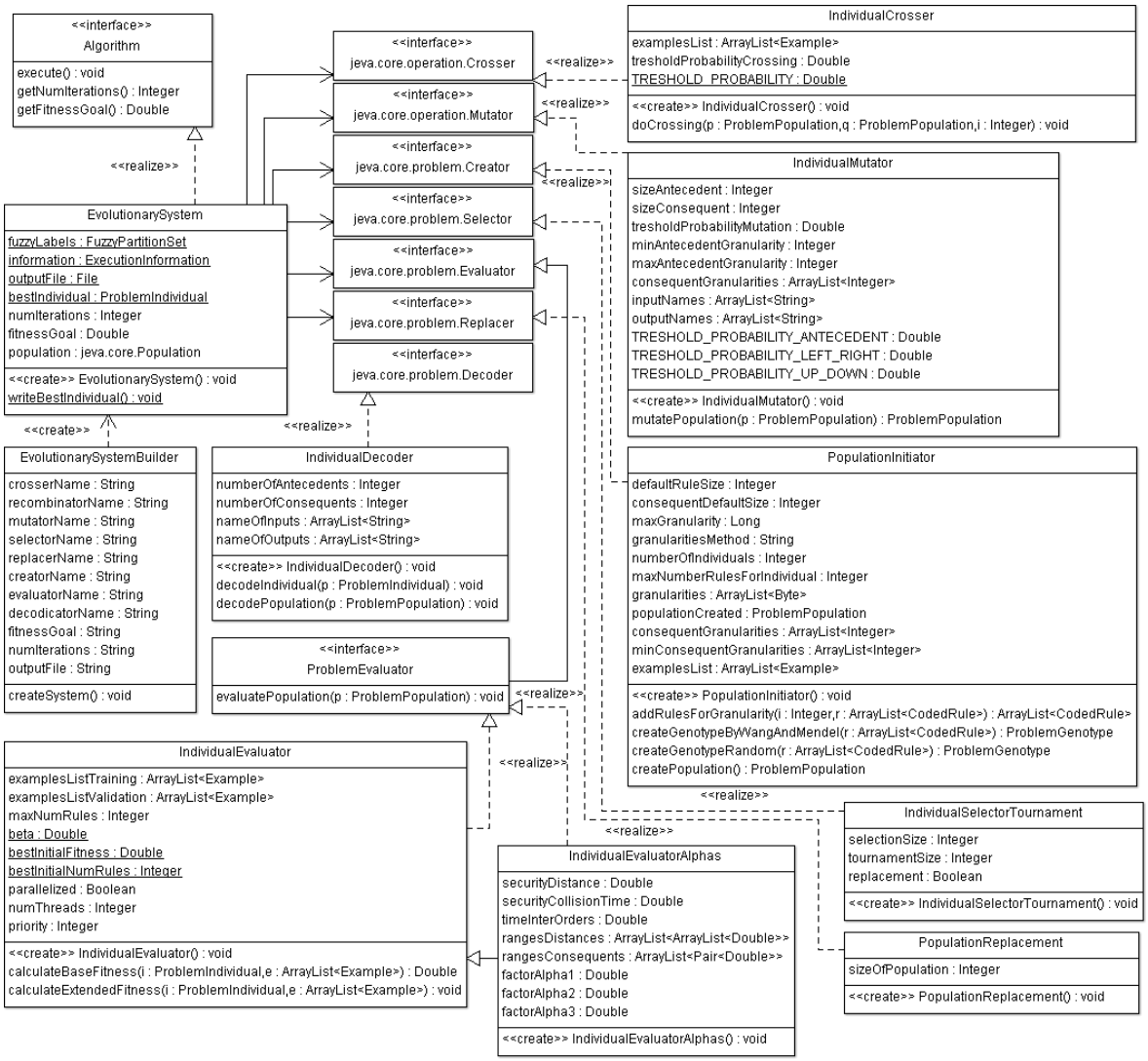

Figura 6.12: Diagrama de clases de la lógica de negocio del componente de aprendizaje.

añade algunos métodos para hacer más usable el código que implementa, permitiendo de esta manera que toda la población ejecute el operador de cruce en base a la dimensión del torneo utilizado por la selección, y no tenga que ser una operación externa a la clase.

- *IndividualMutator:* al igual que el resto de los operadores del algoritmo, también implementa una interfaz de la librería JEVA, en este caso *Mutator*. Se añaden, del mismo modo que el caso anterior, métodos para facilitar la ejecución del algoritmo y hacer internas operaciones que, en caso contrario, deberían hacerse fuera de esta clase, como por ejemplo la aplicación del operador de mutación a toda la población con la que el algoritmo de aprendizaje está trabajando.

- *PopulationInitiator:* su finalidad es la inicialización de la población antes de comenzar el proceso de aprendizaje, esto es, la creación de la población inicial. Aglutina los diferentes métodos de inicialización de un individuo que se recogen en los requisitos, descritos en la sección 4.2. Implementa a la interfaz *Creator* de la librería JEVA, añadiendo mecanismos para facilitar el

uso de la clase, como la creación automática de la población inicial en un único paso.

- *IndividualDecoder:* en esta clase se produce la decodificación del genotipo de un individuo para construir su fenotipo equivalente, para esto se utilizan los conjuntos de etiquetas que se han generado en la fase de inicialización del algoritmo, mapeando los valores entre las etiquetas codificadas y sin codificar.

- *IndividualSelectorTournament:* es la implementación de la interfaz *Selector* de la librería JEVA. En esta clase se produce la ordenación de los individuos de la población según el algoritmo de torneo, generando una población de selección, que es la que se utilizará como base para el cruce.

- *PopulationReplacement:* contiene el código necesario para generar la población que sobrevive a una iteración del algoritmo y, por tanto, que pasa a la siguiente, esto es, aplica el operador de reemplazamiento sobre la población para quedarse con los individuos que sobreviven de ella, descartando a los sobrantes. Implementa la interfaz *Replacer* de la librería JEVA.

- *ProblemEvaluator:* define una interfaz que deben cumplir todos los operadores de evaluación aplicados a este problema concreto para poder ser compatibles e intercambiables entre sí, sin realizar ningún esfuerzo a mayores de codificación que la modificación del parámetro en la clase que genera el algoritmo, *EvolutionarySystemBuilder*. Este mecanismo utiliza un patrón Estrategia que permite tomar la decisión en tiempo de ejecución de qué algoritmo utilizar para evaluar un individuo. La propia interfaz define una serie de métodos que permiten la aplicación del operador sobre toda la población, aunque éstos deberán implementarse en todas las clases que la utilicen.

- *IndividualEvaluator:* implementa el mecanismo inicial de evaluación de un individuo, tal como se describe en los requisitos. Como tal, implementa la interfaz anteriormente descrita.

- *IndividualEvaluatorAlphas:* es una subclase de la anterior. Se ha utilizado el mecanismo de herencia para poder tener definidos los atributos de la clase *IndividualEvaluator* sin tener que volver a codificarlos de nuevo. Esta clase implementa el nuevo método de evaluación descrito en esta misma sección.

Representación de las etiquetas borrosas

El conjunto de etiquetas borrosas utilizadas durante el proceso de aprendizaje es estático, en el sentido de que son inicializadas *a priori* del proceso de aprendizaje, y no son modificadas en ningún momento.

Para su almacenamiento se debía elegir una estructura que permitiese su fácil búsqueda, teniendo en cuenta además que las diferentes etiquetas borrosas están distribuidas según su nivel de granularidad y su variable, por lo que el conjunto de elementos es bastante elevado y debe estar bien organizado.

Para solucionar este problema se utilizó como idea el patrón de diseño *Composite*, definido en la sección 6.2.5, se definió un componente, *FuzzyPartitionSet*, que está capacitado para contener una colección de elementos de sí mismo. Sin embargo, las etiquetas borrosas, instancias de *FuzzySet*, también son objetos que implementan este componente, desarrollando así de modo completo el patrón de diseño descrito anteriormente.

De esta manera, además de tener una organización eficiente de todas las etiquetas borrosas organizadas por niveles y por variables, la búsqueda de un elemento puede realizarse recorriendo en profundidad el árbol hasta encontrar el elemento que se busca, lo cual es una solución mucho más eficiente que recorrer longitudinalmente un conjunto de etiquetas almacenadas en una estructura plana, como un *array*. La implementación de esta solución se detalla en el diagrama que contiene la figura 6.13 donde, por simplicidad, se omiten los atributos de las clases que forman parte de la estructura, pues algunas de ellas provienen directamente de la utilización de librerías externas, como *JFR*.

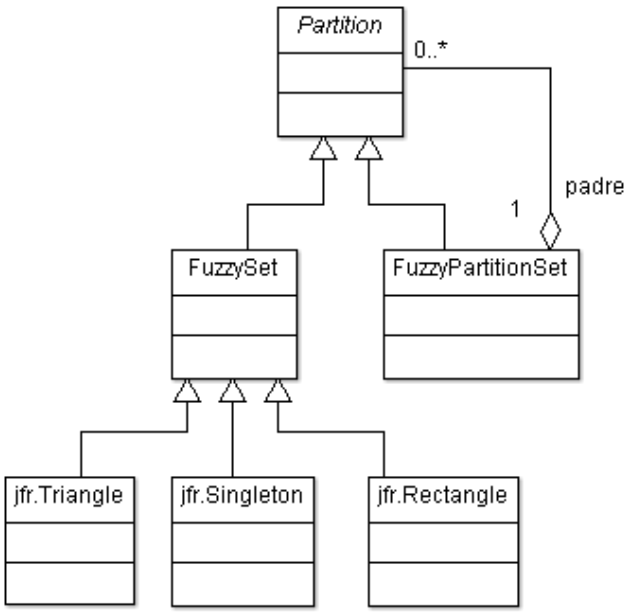

Figura 6.13: Dieño de la estructura de almacenamiento de las etiquetas borrosas utilizando el patrón *Composite*.

6.3.3. Salida del proceso de aprendizaje

Una vez finalizada la ejecución del algoritmo evolutivo que permite el aprendizaje de una base de reglas, ésta debe convertirse a un formato determinado para que pueda ser leída por el controlador borroso. El formato de salida debe ser el mismo que el que acepta el componente de lectura del fichero que contiene la base de reglas, que se encuentra implementado como parte de la librería *QFTR*.

El fichero de la base de reglas se divide entre cabecera y cuerpo. La cabecera está formada por información sobre la composición de la base de conocimiento, para poder estructurarla correctamente, mientras que el cuerpo está formado por una colección de reglas descritas secuencialmente.

Por una parte, la cabecera está formada por las siguientes líneas, donde debe detallarse el número de reglas que contiene el fichero, y la composición de cada una de ellas, esto es, el número de variables

que tiene el antecedente, y el número de variables del consecuente, como se muestra en el siguiente ejemplo:

```
Number of rules: 31
Number of antecedents: 6
Number of consequents: 2
```

A continuación se escriben secuencialmente cada una de las reglas que contiene el individuo, donde cada una de ellas está determinada por el siguiente formato:

```
Rule 0:
0 # 1 # 0.0 0.0 0.0 0.5
-1 #
0 # 1 # 0.0 0.0 0.0 0.5
0 # 1 # 0.0 0.0 0.0 3.0
-1 #
0 # 1 # 0.0 0.0 0.0 0.16666666666666666
0.01 0.01 0.01 0.01 w 1.0
-0.174532889 -0.174532889 -0.174532889 -0.174532889 w 1.0
```

Como se puede comprobar, cada regla va precedida del texto *Rule* y el número de regla, empezando en 0. A continuación, se escriben en cada línea un antecedente o consecuente. Los antecedentes están especificados por el siguiente formato: el inicio de la línea es común siempre que se vaya a especificar una etiqueta borrosa, y a continuación se coloca una tupla de 4 valores, que determinan la etiqueta borrosa de la variable de entrada.

Cuando la variable del antecedente no es tenida en cuenta para el proceso de inferencia, se define colocando en la línea correspondiente *-1* ♯. En nuestro caso, esto sucede cuando la granularidad de esa variable es 1, equivalente a decir que se representa mediante un rectángulo, lo que representa que su valor no se tiene en cuenta en esa regla.

Los consecuentes se definen de manera similar a los antecedentes, en primer lugar se coloca la tupla de 4 valores que determina la etiqueta borrosa, y a continuación una w y un número dentro del intervalo $[0, 1]$, que indica el peso del consecuente. Por defecto en este problema se considerará que los pesos de todos los consecuentes tienen el máximo valor.

La razón por la que las etiquetas borrosas se definen mediante 4 puntos es porque el mecanismo de inferencia implementado en la librería QFTR está realizado basado en trapecios, por lo que se necesitan los cuatro puntos para describirlo, tal y como se muestra en la figura 6.14, independientemente de que dentro de la aplicación tenga otro tratamiento.

En el caso de esta aplicación, donde las etiquetas de los antecedentes están representadas mediante triángulos, la conversión en un trapecio es simple, sólo hay que pasar de escribir la etiqueta como una tupla de tres valores, (A, B, C) a escribirla como una de cuatro (A, B, B, C). En el caso de los consecuentes, donde la etiqueta borrosa está definida por un único valor, lo que se hace es repetirlo hasta que el trapecio quede definido.

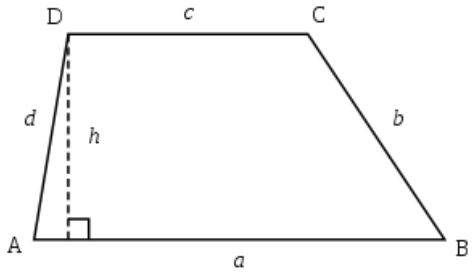

Figura 6.14: Definición de los conjuntos borrosos como trapecios en el fichero de salida.

6.3.4. *Logs* generados

Durante la etapa de análisis de requisitos, se ha identificado la necesidad de que la aplicación genere una serie de archivos de registro donde se detalle toda la actividad que corresponde al proceso de aprendizaje.

Para facilitar la lectura de estos registros, se ha diseñado el programa de tal forma que se produzcan los siguientes archivos durante la ejecución del algoritmo:

- Un archivo donde se detalle el *ranking* de la población para cada una de las iteraciones del algoritmo. Éste se imprimirá de modo ordenado, detallando para cada individuo su *fitness* de entrenamiento y de validación.

- Para cada iteración del algoritmo, se genera un archivo con extensión *.log* seguido del número de iteración a la que corresponda el log. Cada uno de los archivos contendrá de modo detallado el fenotipo y el genotipo de cada individuo, así como las operaciones que se han realizado sobre él para que haya llegado al estado actual.

  De esta forma, el hecho de numerar la extensión de cada uno de ellos sirve para identificar de modo más rápido el punto de ejecución del que se desean examinar los registros de actividad.

La implementación de la parte del sistema que realiza la escritura de los logs se ha hecho a partir del patrón de diseño *Observer*, tal y como se puede ver en el diagrama de clases que contiene la figura 6.15.

Cada una de las clases que implementan la interfaz *Observer* de Java representa uno de estos dos tipos de logs generados. Por una parte, la clase *LogCreator* es la que escribe los registros donde se detalla todo el contenido de la población al final de cada iteración, y la clase *SummaryCreator* es la que se encarga de la escritura del fichero donde se hace un resumen de la evolución de la población, escribiendo un *ranking* por cada una de las iteraciones.

Finalmente, la clase auxiliar *LogManagement* es la clase mediante la cual se inicializan las anteriores, y se detalla qué objeto es el que tienen que tomar como observado. Al finalizar la ejecución de cada iteración, se enviará una señal a todos los objetos que ejercen de observadores, para que realicen las funciones que implementan, en este caso el registro de dichos cambios en un conjunto de ficheros. Como muestra, se detalla una parte del contenido que genera, por una parte, la clase *LogCreator*:

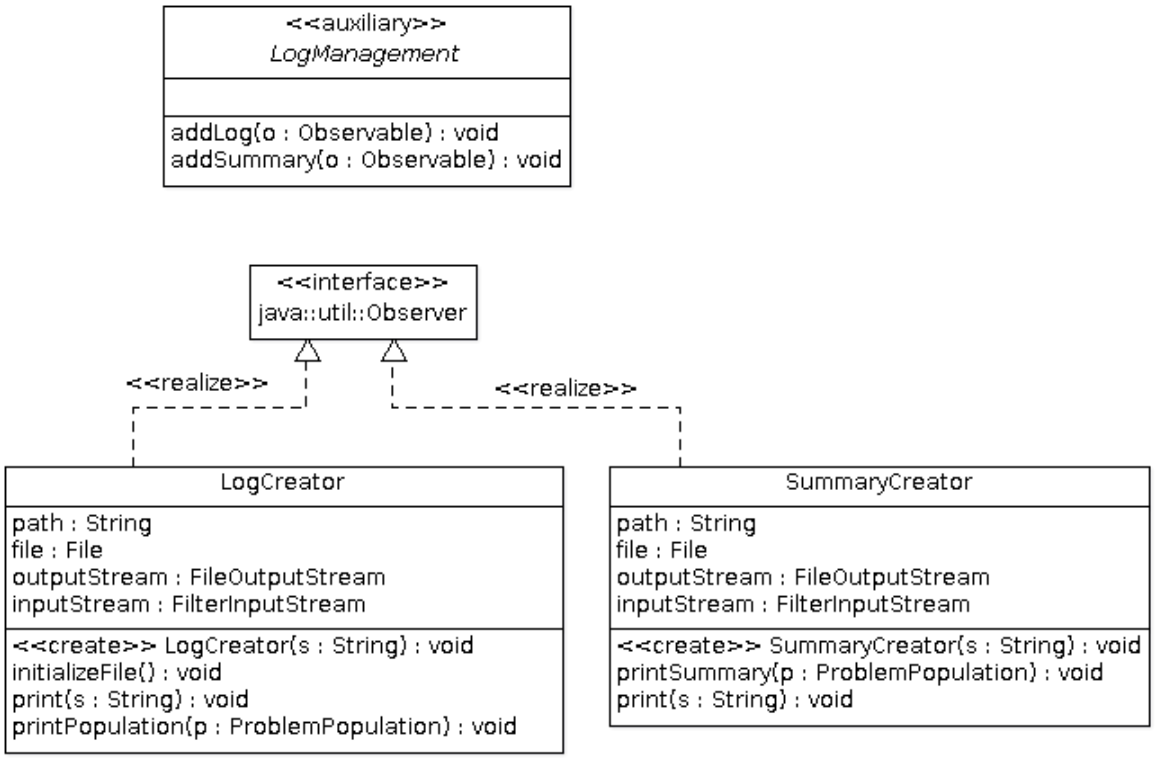

Figura 6.15: Diagrama que representa la implementación del escritor de *logs*.

```
------------------------------------------------------------------------
Execution of the program: Robot control with evolutionary algorythm.
Begin time: Tue Jul 06 00:24:01 CEST 2010
Population iteration: 1
------------------------------------------------------------------------
-------------------------
Individual: IT1IN39
Created in iteration 1
Mutated from: IT0IN1
Mutation iteration: 1
Aditional information:
Basic fitness: 479625.1297187313
Beta: 0.05
Best initial fitness: 479625.1297187313
max Num rules: 200
Individual size: 56
Extended fitness: 473230.1279891482
-------------------------
Information about the Phenotype:
0.  (frontDistance, 0.0, 0.0, 0.6)
```

```
(rightDistance, 0.0, 0.0, 0.2)
(leftDistance, 0.0, 1.0)
(objectiveDistance, 0.0, 0.0, 3.0)
(objectiveAngle, -0.7854, 0.7854)
(speed, 0.0, 0.0, 1.0)
(linealSpeed, 0.01)
(angularSpeed, -0.523598667)

⋮

55.  (frontDistance, 2.4, 3.0, 3.0)
(rightDistance, 0.8, 1.0, 1.0)
(leftDistance, 0.0, 1.0)
(objectiveDistance, 0.0, 3.0, 3.0)
(objectiveAngle, -0.7854, 0.7854)
(speed, 0.0, 0.0, 1.0)
(linealSpeed, 0.56)
(angularSpeed, -0.17453288900000002

Information about the Genotype:
0.  [ [6, 1][6, 1][1, 1][2, 1][1, 1][2, 1][9, 1][9, 1] ]

⋮

55.  [ [6, 6][6, 6][1, 1][2, 2][1, 1][2, 1][9, 6][9, 4] ]
-------------------------
```

Por otra parte, la clase *SummaryCreator*, que genera el contenido del resumen de la población en cada una de las iteraciones, guarda un archivo de texto de la forma siguiente:

```
---------------------------------------------------------------------
Summary of the contents of the population: Robot control with evolutionary algorythm.
Begin time: Tue Jul 06 00:23:41 CEST 2010
---------------------------------------------------------------------
Iteration 1
Individual Fitness T Fitness V
--------- --------- ---------
IT0IN1 473230.1279891482 473230.1279891482
IT1IN39 473230.1279891482 473230.1279891482
IT0IN85 469194.8084202579 469194.8084202579

⋮

IT1IN116 433310.0412624169 433310.0412624169
Best fitness training: 473230.1279891482
```

```
Best fitness validation: 473230.1279891482
Mean of fitness training: 447021.19158412353
Mean of fitness training: 447021.19158412353
```

El fichero resumen se guarda en el directorio de ejecución del programa, mientras que los archivos detallados de registro se generan en una carpeta, llamada *logs*, debido a que su número puede resultar muy variable. El contenido de esta carpeta se vacía cada vez que el algoritmo empieza una nueva ejecución, para evitar mezclar datos de diferentes ejecuciones en el caso de que el número de iteraciones no sea el mismo.

6.3.5. Interfaz gráfica

Uno de los requisitos del proyecto, el RF-6, especificado en la sección 4.2, detalla que se debe realizar de alguna forma un sistema que permita la visualización de la evolución de la población conforme el algoritmo evolutivo se está ejecutando, de tal forma que se puedan detectar anomalías durante la ejecución por parte del usuario de la aplicación.

Para dar cumplimiento a este requisito se ha diseñado una interfaz gráfica que permita el acceso a algunas de las funcionalidades más básicas de la herramienta de aprendizaje, entre otras cosas la visualización de una gráfica en la que se detalle el valor de puntuación para la población en cada una de las iteraciones del algoritmo, y la modificación de algunos de los parámetros de aprendizaje más utilizados.

En esencia, la interfaz gráfica se compone de tres ventanas donde se desarrolla toda la interacción con el usuario. Desde la ventana principal, que se muestra en la figura 6.16, se permite la ejecución del algoritmo de aprendizaje, su interrupción, la visualización del progreso del aprendizaje mediante una gráfica dinámica que, además, se puede personalizar de tal forma que muestre los valores de *fitness* sólo para la población, o también para cada uno de los individuos que se han colocado como los mejores de la población en algún momento de la ejecución. Una barra de progreso en la parte inferior de la ventana informa al usuario de cuántas iteraciones del algoritmo faltan para completar el proceso de aprendizaje.

Los diferentes menús de la parte superior de la ventana principal permiten el acceso a otras opciones, tales como el guardado y la carga de estadísticas de otras ejecuciones para permitir su visualización en la gráfica, o la apertura de la ventana de opciones. Dicha ventana se divide en dos pestañas internamente:

- La pestaña *Población* muestra una serie de parámetros modificables desde la interfaz gráfica, que tienen que ver con la personalización de la población que se utiliza para el proceso de aprendizaje, tales como el tamaño de la población, el tamaño máximo de los individuos, y la definición completa de las variables que participan en las reglas, tanto en el antecedente como en el consecuente.

  Se ha incluido esta funcionalidad como parte de la interfaz gráfica porque la modificación de los ficheros de opciones puede resultar engorrosa y dar lugar a errores, de tal modo que si este proceso se realiza desde un entorno más amigable, además de resultar más claro, se minimizarán

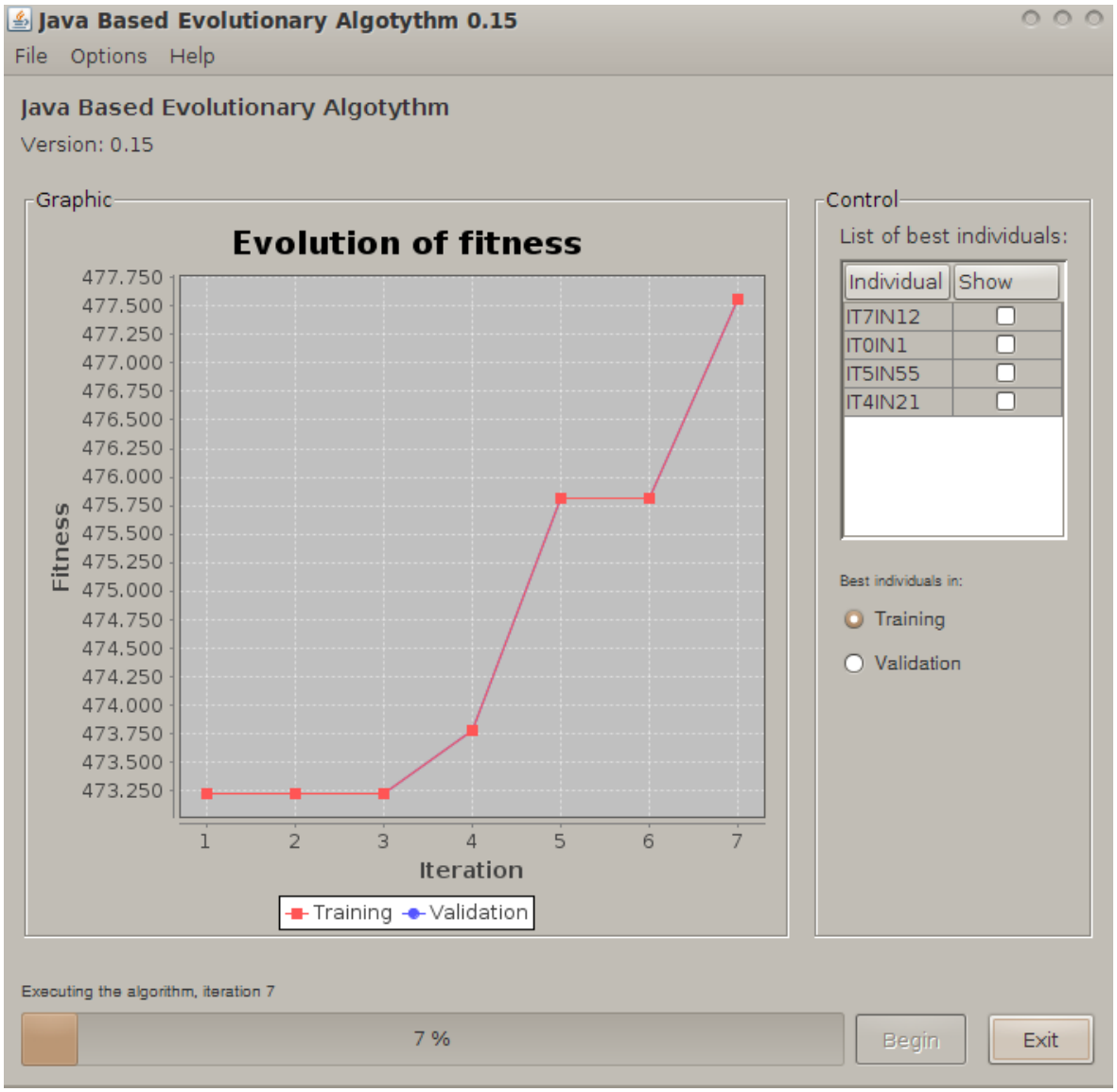

Figura 6.16: Ventana principal de la herramienta de aprendizaje.

los errores por parte del usuario y, finalmente, los que se produzcan podrán ser tratados desde la propia interfaz gráfica.

En la figura 6.17 se muestra el diseño que se ha realizado para esta parte que muestra el conjunto de opciones de la población personalizables desde el entorno visual.

- Por otra parte, la pestaña *Sistema* realiza las operaciones análogas a la pestaña *Población*, pero con los parámetros generales del proceso de aprendizaje. Desde esta parte se permite la personalización de los siguientes parámetros: el fichero de ejemplos utilizado por el algoritmo evolutivo para la introducción de conocimiento experto; los pesos para la función de evaluación de cada una de las fuentes de error (pesos de α_i); el valor del parámetro β, también para la función de evaluación; la distancia de seguridad del robot; el número de iteraciones y la posibilidad de

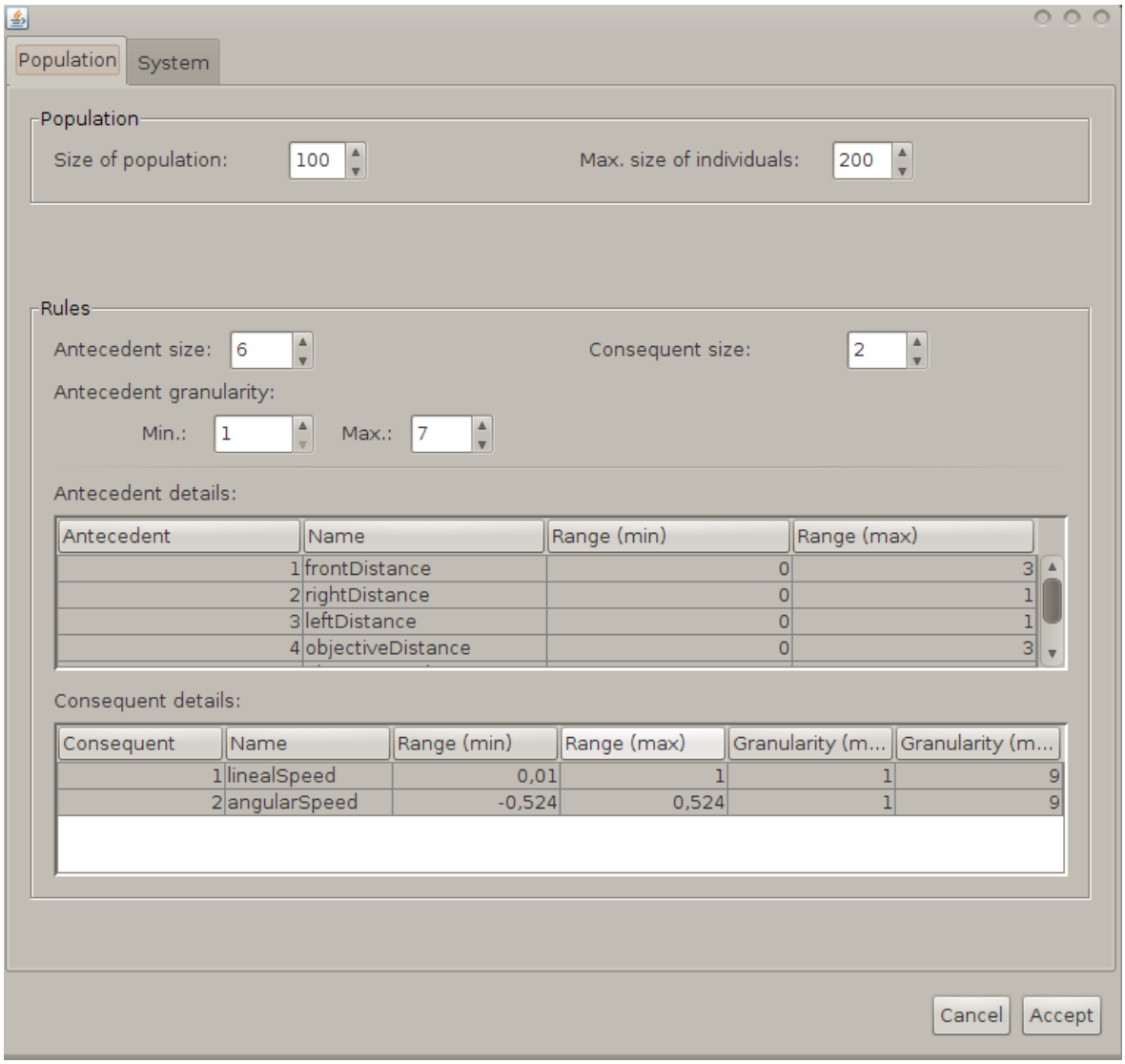

Figura 6.17: Ventana de opciones de la población de la herramienta de aprendizaje.

utilizar o no paralelización para ejecutar el algoritmo de aprendizaje.

El diseño para esta parte de la ventana de opciones del sistema se muestra en la figura 6.18.

Cuando se aceptan los cambios que se han introducido en la ventana de opciones, se sobreescriben los ficheros de preferencias utilizados por la herramienta para determinar las opciones con las que se ejecutará el algoritmo. Teniendo en cuenta que no todas las opciones son modificables desde la interfaz gráfica, se ha realizado la implementación de esta funcionalidad de tal modo que la sobreescritura sólo conlleve la modificación de los parámetros que se han modificado desde el entorno visual, conservando el resto de las opciones intactas.

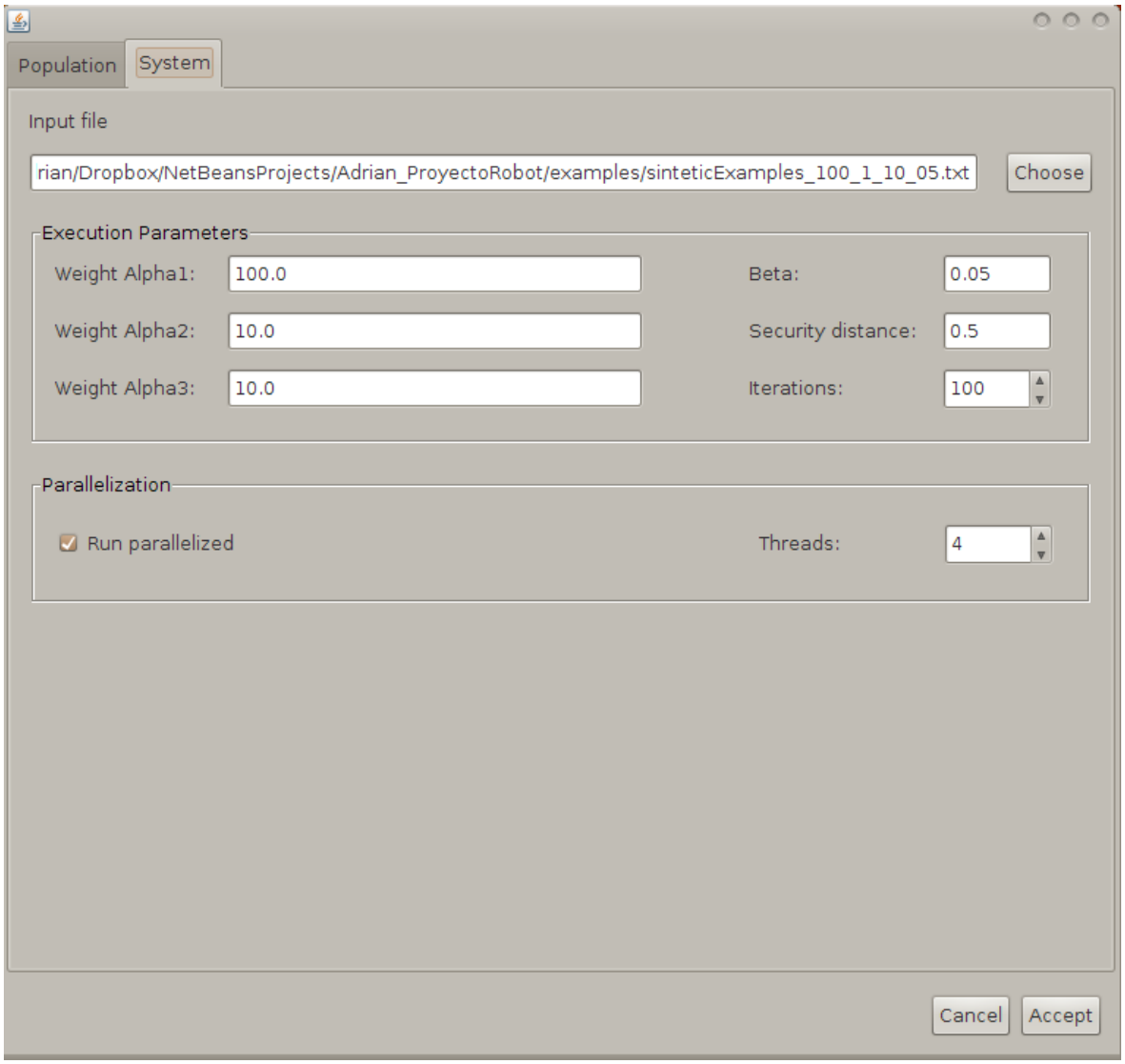

Figura 6.18: Ventana de opciones generales, o del sistema, de la herramienta de aprendizaje.

6.4. Controlador borroso

Es el componente más sencillo, ya que numerosas funcionalidades que dependen de él están implementadas en las librerías que se han descrito en la sección 4.4, concretamente la librería *Robot* y *QFTR*, que es la que se encarga de la comunicación con los diferentes sensores del robot, y de realizar la inferencia con la base de conocimiento que se ha generado.

En esencia este componente tan sólo tiene una clase que ejecuta los métodos de recogida de información sobre el entorno, realiza transformaciones sobre ellos para adaptarlos a la base de conocimiento, de la misma forma en la que se realiza la transformación de datos del formato de *log* de Player/Stage al formato de ejemplos descrito en la sección 6.3.1, y finalmente realiza el proceso de inferencia de la base de reglas obtenida durante el proceso de aprendizaje.

Capítulo 7

Validación y pruebas

La realización de pruebas, así como la etapa de validación del proyecto, son pasos esenciales para asegurar la calidad del producto final. Las pruebas de software intentan verificar que los distintos componentes de la aplicación realizan cada una de las funciones para las que han sido diseñados correctamente. Existen diferentes tipos de pruebas que pueden ser realizadas sobre el software implementado:

- *Pruebas de unidad:* se aplican sobre cada uno de los módulos de código que son representativos para la realización de las funciones de los componentes que forman parte de la aplicación. Su finalidad es comprobar a bajo nivel el funcionamiento del programa, para evitar errores de programación que den como resultado una ejecución incorrecta de las operaciones. Debido a su definición, son consideradas pruebas de *caja blanca*, donde lo que se pretende probar es, paso a paso, qué hace el software implementado.

  Las pruebas unitarias son específicas para cada uno de los módulos de código sobre las que se ejecutan. Debido al coste temporal que supondría diseñar pruebas unitarias para cada uno de los módulos de modo formal, no se van a tener en cuenta para este proyecto de software. En cualquier caso, no resultan muy adecuadas para el tipo de herramientas desarrolladas, donde existe una importante base aleatoria y, por tanto, la definición de pruebas formales es mucho más complicada, debido a la imposibilidad de asegurar que dos ejecuciones van a tener el mismo resultado si se realizan sobre las mismas entradas.

- *Pruebas funcionales:* son pruebas basadas en las funcionalidades que debe implementar el programa, que intentan asegurar, mediante el estudio de las entradas y de las salidas de cada uno de los módulos del programa, su correcto funcionamiento, esto es, que la ejecución ha devuelto los resultados previstos para las entradas con las que se ha ejecutado, sin estudiar el modo en el que realiza las operaciones.

  Este tipo de pruebas, englobadas dentro de la categoría de *caja negra*, buscan en primer lugar dividir el algoritmo implementado en diferentes módulos que puedan ser probados independientemente, estudiando qué datos recibe cada uno de ellos, y si las salidas que se registran tras su ejecución son adecuadas. Si se conocen previamente las funciones que debe implementar el

software, también pueden enfocarse a la demostración de que cada una de estas funcionalidades están bien implementadas.

Las pruebas de validación tienen un enfoque diferente a las pruebas de software. En estas últimas lo que se busca es verificar el cumplimiento de los distintos requisitos definidos por el cliente, así como comprobar que sirve al propósito para el que se ha diseñado. A lo largo de este capítulo se detallará el conjunto de pruebas tanto de software como de validación que se han llevado a cabo, especificando para cada una de ellas el resultado.

7.1. Pruebas funcionales

A continuación se detallan cada una de las pruebas de funcionamiento que se han llevado a cabo sobre el software construido, especificando en qué consiste cada una de ellas y el resultado de su ejecución:

Generación de un ejemplo partiendo de la información de Player/Stage

Descripción: partiendo de la información de una medición del programa de simulación Player/Stage, se aplica la herramienta de conversión para generar un ejemplo apto para ser utilizado en el proceso de aprendizaje. Para superar esta prueba se debe comprobar que el cálculo para cada una de las variables del antecedente y del consecuente del ejemplo es correcto.

Resultado: realizado correctamente.

Asignación de un consecuente a un ejemplo sintético

Descripción: teniendo la información del antecedente de un ejemplo generado sintéticamente, el objetivo de esta prueba es comprobar que la función de evaluación se aplica correctamente y permite la selección, de entre todos los consecuentes posibles, del que más se adecúa a los valores del antecedente. La superación de esta prueba requiere la comprobación de los valores de evaluación asignados a cada una de las combinaciones de antecedente y consecuente, confirmando que se lleva a cabo la selección de la mejor de ellas.

Resultado: realizado correctamente.

Reducción de ejemplos

Descripción: dado un conjunto de ejemplos, se pretende reducir su número mediante la aplicación de un algoritmo que seleccione aquellos realmente representativos dentro del conjunto, eliminando el resto, ya que no proporcionan información adicional significativa. Esta prueba lo que pretende es comprobar que realmente los ejemplos que forman parte del conjunto final son representativos, esto es, no son redundantes entre sí.

Resultado: comportamiento inesperado.

Acciones tomadas: implementación de un nuevo algoritmo para la reducción de ejemplos, realizando de nuevo las pruebas de funcionalidad sobre éste para comprobar su funcionamiento.

Resultado final: realizado correctamente

Generación de la población inicial

Descripción: la ejecución del algoritmo de aprendizaje comienza por una etapa de inicialización donde se llevan a cabo numerosas operaciones, que culminan con la generación de un conjunto de individuos que conforman la población inicial, y que provienen de dos métodos de inicialización diferentes. Esta prueba pretende la comprobación de que cada uno de los métodos de inicialización se ha llevado a cabo correctamente, realizando la creación de parejas de individuos donde los antecedentes son iguales, y los consecuentes dependen del método de generación utilizado para cada individuo.

Resultado: realizado correctamente.

Evaluación de un individuo

Descripción: es una de las partes de la herramienta de aprendizaje cuyo correcto funcionamiento debe garantizarse. Para superar esta prueba es necesaria la comprobación de la consistencia de los valores de cada uno de los componentes que forman parte de la función de evaluación, lo cual permitirá identificar posibles errores en la obtención de los datos necesarios para calcularlos. El valor final de puntuación otorgado a los individuos debe ser proporcional a su calidad.

Resultado: comportamiento inesperado.

Acciones tomadas: redefinición del conjunto de acciones destinadas a evaluar un individuo, reimplementando el módulo que realiza su evaluación, y realización de las pruebas de funcionalidad una vez finalizada.

Resultado final: realizado correctamente

Cruce entre dos individuos

Descripción: se debe observar el resultado del proceso de cruce entre dos individuos, esto es, identificar la descendencia de una pareja determinada, y comprobar que la asignación de las reglas a los individuos se lleva a cabo de modo correcto, comprobando que no existen duplicidades, o inconsistencias, verificando además que el número de reglas del conjunto de los padres y de la descendencia obtenida no registra un cambio muy elevado.

Resultado: realizado correctamente.

Mutación de un individuo

Descripción: la superación de esta prueba está condicionada a que durante la modificación de un individuo para añadir cambios a las reglas que lo componen no se conserven reglas inconsistentes o redundantes, vigilando además que no se colocan valores situados fuera de rango en el genotipo, lo cual

no tendría una correspondencia en el fenotipo e induciría a errores. Para esto, hay que comprobar que en cada tupla que codifica una regla, $[granularidad, etiqueta]$, *etiqueta* es siempre un valor contenido entre 1 y *granularidad*, y que este último se encuentra entre 1 y $granularidad_{max}$, que es definido por el usuario previamente a la ejecución del algoritmo.

Resultado: realizado correctamente.

Reemplazamiento de la población

Descripción: se debe comprobar el contenido de la población antes de aplicar este operador, y también posteriormente. Es necesario verificar que la ordenación de individuos según su *ranking* se realiza correctamente, para lo que se pueden utilizar los *logs* de la aplicación. El tamaño de la población en cada iteración debe mantenerse constante.

Resultado: realizado correctamente.

Selección de parejas para el cruce

Descripción: para que esta prueba sea exitosa se debe verificar que la aplicación del proceso de selección sobre la población del algoritmo devuelve un conjunto ordenado donde los individuos están organizados por parejas. Debe comprobarse que el conjunto verifique el tamaño de la selección, y que se cumpla que los individuos de mayor *fitness* tengan una mayor aparición que aquellos que no lo tienen. Que esto no ocurra de esta forma podría indicar una implementación incorrecta de este operador.

Resultado: realizado correctamente.

7.2. Validación de requisitos

En esta sección se detallará el grado de cumplimiento de cada uno de los requisitos que forman parte de este proyecto, y que se han detallado en la sección 4.1. Para una mayor claridad se han especificado el número de requisito, el título que lo identifica, y el grado de cumplimiento que se ha logrado, separando además los requisitos funcionales y los no funcionales para facilitar su localización.

7.2.1. Requisitos funcionales

Requisito RF-1: Aprendizaje de una base de reglas borrosas

Estado: cumplido

Justificación: la implementación de la herramienta de aprendizaje se ha completado, y la ejecución del algoritmo evolutivo permite la selección del mejor individuo que la población contiene tras N iteraciones.

Requisito RF-2: Aproximación *Pittsburgh*

Estado: cumplido

Justificación: un individuo de la población contiene un conjunto de reglas que forman una base de conocimiento entera. La aproximación *Pittsburgh* define que se haga de esta forma.

Requisito RF-3: Salida del algoritmo

Estado: cumplido

Justificación: la escritura del fichero de salida con el mejor individuo, seleccionado tras el proceso de aprendizaje, se lleva a cabo tras finalizar éste, en el fichero de salida que determine el usuario en cada ejecución.

Requisito RF-4: Flexibilidad en la definición de la población y el funcionamiento general del algoritmo

Estado: cumplido

Justificación: se definen dos ficheros de opciones, uno para contener aquellas relacionadas con la población y su composición, y otro para las opciones relacionadas con el funcionamiento general del algoritmo. El primero de ellos permite una definición de todos los parámetros que atañen a la población, su composición, la formación de las reglas, etc; mientras que el segundo permite la definición de los parámetros que utiliza el proceso de aprendizaje pero que no se encuentran relacionados con los primeros, sino que tienen que ver más con el funcionamiento general de la herramienta y el algoritmo.

Requisito RF-5: Seguimiento de las operaciones

Estado: cumplido

Justificación: en cada iteración del algoritmo se produce un fichero de log que detalla la población a su más bajo nivel, esto es, para cada individuo se escriben las reglas que contiene tanto en formato codificado como en formato de reglas borrosas, en qué iteración se ha creado, los operadores que han sido aplicados sobre él, etc. Además, a modo de resumen se va escribiendo un fichero con un *ranking* de la población en cada iteración.

Requisito RF-6: Visualización de la evolución de la población

Estado: cumplido

Justificación: se ha diseñado una interfaz visual que contiene una gráfica donde se muestra la evolución del *fitness* de la población. Ésta se actualizará en tiempo real a medida que la ejecución del algoritmo se ejecuta en paralelo, de esta forma se pueden detectar anomalías mientras se está produciendo la ejecución del proceso de aprendizaje.

Requisito RF-7: Uso de ejemplos para el proceso de aprendizaje

Estado: cumplido

Justificación: el algoritmo toma como una de las entradas un fichero donde se encuentran todos los ejemplos que van a ser utilizados durante el aprendizaje. Los ejemplos serán utilizados de modo directo por los operadores de evaluación y de cruce, entre otras cosas.

Requisito RF-8: Generación de ejemplos a partir de Player/Stage

Estado: cumplido

Justificación: se ha producido la implementación del módulo que genera ejemplos partiendo del conocimiento contenido en registros de actividad del programa de simulación, de la forma en la que se detalla en 6.3.1, sometiéndolo a las pruebas de funcionalidad que se describen en 7.1.

Requisito RF-9: Generación de ejemplos sintéticamente

Estado: cumplido

Justificación: se ha implementado un módulo que realiza esta funcionalidad, generando un fichero de ejemplos donde inicialmente sólo se tiene información sobre las variables a las que afecta el aprendizaje, tal y como se explica con detalle en la sección 6.3.1. Se ha sometido a un conjunto de pruebas de funcionalidad para asegurar que los ejemplos generados son correctos, proceso que se ha descrito en 7.1.

Requisito RF-10: Operador de selección

Estado: cumplido

Justificación: la implementación de este operador se ha llevado a cabo según las directrices especificadas en la sección 4.2, explicado de un modo más detallado en la sección 3.5, sin realizar cambios según lo requerido inicialmente. El funcionamiento de este operador se ha incluido dentro de las pruebas de funcionalidad especificadas en la sección 7.1.

RF-11: Operador de cruce

Estado: cumplido

Justificación: este operador se ha implementado según la especificación detallada tanto en la sección 4.2, y más detalladamente en la sección 3.6. La implementación se ha llevado a cabo sin cambios respecto a lo inicialmente. También se ha incluido en el plan de pruebas detallado en la sección 7.1.

RF-12: Operador de mutación

Estado: cumplido

Justificación: la implementación de este operador también se ha llevado a cabo según lo previsto en las especificaciones dadas por el cliente, que se detallan en las secciones 4.2 y 3.7, sin ningún cambio. Se han realizado pruebas sobre su implementación que se han reflejado en la sección 7.1.

RF-13: Operador de reemplazamiento

Estado: cumplido

Justificación: implementación realizada según las especificaciones detalladas en las secciones 4.2 y 3.8. No se han introducido cambios respecto a los requisitos iniciales, y se ha incluido en el plan de pruebas detallado en la sección 7.1, concluyendo exitosamente.

RF-14: Función de evaluación

Estado: cumplido con modificaciones

Justificación: la función de evaluación definida en las secciones 4.2 y, más concretamente, en 3.9, ha tenido que ser redefinida, puesto que la especificación inicial no reflejaba un buen aprendizaje, que es uno de los objetivos del proyecto. La redefinición ha sido explicada en detalle en la sección 6.3.2, realizando también pruebas de funcionalidad sobre él para asegurar su correcto funcionamiento, tal y como se explica en la sección 7.1.

RF-15: Reducción del número de ejemplos

Estado: cumplido con modificaciones

Justificación: inicialmente se ha procedido a la implementación del algoritmo de reducción de ejemplos IB2, descrito en la sección 3.2, pero tras la realización de las pruebas de funcionalidad se ha concluido que este algoritmo no arrojaba unos buenos resultados para este problema concreto, por lo que se procedió a la implementación de uno nuevo, descrito en la sección 6.3.1, sobre el que también se han realizado pruebas que se detallan en la sección 7.1.

RF-16: Inicialización de la población

Estado: cumplido

Justificación: se ha realizado la implementación de los métodos para la creación de la población inicial, del modo en el que aparece descrito en la sección 3.4. Se ha asegurado la correcta implementación de esta funcionalidad mediante la realización de las pruebas descritas en la sección 7.1.

Requisito RF-17: Flexibilidad en la definición de las reglas

Estado: cumplido

Justificación: se ha especificado un fichero de propiedades, *Population.properties*, donde se define el formato de las reglas que utiliza la aplicación. Cambiando ese formato no hay que realizar ninguna

modificación sobre el código para que el algoritmo evolutivo funcione con otro formato de reglas diferente. Además, desde la interfaz gráfica se proporcionan herramientas para modificar la composición de las reglas de un modo más intuitivo que la edición del fichero de opciones a mano.

Requisito RF-18: Formato e información contenida en las reglas

Estado: cumplido

Justificación: los parámetros de la definición de la población se han establecido de tal forma que la definición del formato de las reglas de los individuos coincida con la especificación realizada en los requisitos.

7.2.2. Requisitos no funcionales

Requisito RNF-1: Posibilidad de intercambio en operadores genéticos

Estado: cumplido parcialmente

Justificación: cuando se ha procedido a la implementación de cada uno de los operadores que participan en el proceso de aprendizaje: inicialización, evaluación, selección, cruce, mutación y reemplazamiento, se utilizó una estrategia de diseño que permitiese su sustitución por otros que implementasen sus mismas interfaces. Sin embargo, en algunas ocasiones, como en el operador de cruce, se hizo necesario la llamada a algún método propietario de la implementación, no al de la interfaz que implementa, por lo que en ese caso su compatibilidad con otros operadores no está asegurada.

Requisito RNF-2: Tiempo de ejecución lo más bajo posible y RNF-4: Paralelización de código

Estado: cumplido

Justificación: para realizar las operaciones del algoritmo evolutivo que más tiempo de computación consumen, se han realizado labores de paralelización, por ejemplo a nivel de inicialización de la población, así como del proceso de evaluación. La ejecución de varios hilos simultáneos en la aplicación permite la disminución del tiempo de ejecución que se registra frente a la ejecución de la herramienta sin paralelización. Como estos dos requisitos están directamente relacionados, en tanto que la consecución del requisito RNF-5 influye en el RNF-3, se solucionaron de modo simultáneo con la realización del primero.

Requisito RNF-3: Utilización de la herramienta Player/Stage

Estado: cumplido

Justificación: se ha utilizado dicha herramienta para llevar a cabo la simulación del comportamiento que implementa el controlador borroso, de tal forma que se puedan realizar pruebas que permitan visualizar en distintos entornos cómo responde el robot a las diferentes entradas.

Requisito RNF-5: Independencia de los parámetros de ejecución

Estado: cumplido

Justificación: del mismo modo que se ha realizado con la definición de las reglas y de la población en general, se ha procurado realizar el diseño de la aplicación abstrayendo del código los elementos que son parámetros de creación y de ejecución del algoritmo, como por ejemplo: número de iteraciones, parámetros de evaluación, o fichero de ejemplos utilizado para el aprendizaje, entre otros.

Requisito RNF-6: Uso de librerías

Estado: cumplido

Justificación: se han utilizado librerías que ayudaron a la codificación y el diseño del proyecto en diferentes aspectos, tanto la codificación de la herramienta que permite visualizar la evolución de la población, como la estructura general del algoritmo evolutivo, como el trabajo básico con etiquetas borrosas.

Requisito RNF-7: Comentarios en inglés

Estado: cumplido

Justificación: se ha llevado a cabo toda la documentación, excepto la redacción de la presente memoria, en inglés.

7.3. Ejemplo de ejecución

En esta sección se describirá una ejecución del algoritmo de aprendizaje, de modo que se pueda comprobar el tipo de resultados que ofrece, además de analizar la influencia de algunos parámetros de ejecución, como β que pondera el peso que tiene el tamaño del individuo en su *fitness*.

La salida del algoritmo de aprendizaje es una base de conocimiento que es utilizada por el controlador borroso para en una situación determinada. Uno de los parámetros más influyentes en dicho proceso es el parámetro β, pues su valor influye directamente en el número de reglas que tiene la base de conocimiento que se obtiene como salida. Tras la realización de cada prueba de ejecución, se han analizado los datos de salida y se han comparado con el valor definido en dicho parámetro, los resultados de dicho análisis se detallan en la tabla 7.1.

En la figura 7.1 se representa el impacto que tiene el valor de β respecto a la precisión del controlador aprendido y al número de reglas del mismo. Como se puede ver, cuanto menor es el valor de β, mayor es el número de reglas generadas. Por contra, a mayor beta, menor es el valor de fitness obtenido por el algoritmo de aprendizaje, lo cual indica que la base de conocimiento obtenida es mejor (más precisa). la precisión del aprendizaje.

Se toma a continuación una ejecución concreta, donde se ha definido el valor de β como 0.05. Se registró una curva de aprendizaje con un alto progreso al principio, moderándose a medida que

Valor de β	Tamaño de la base de reglas	Fitness promedio
0.1	≈ 10	490150
0.05	≈ 20	482291
0.025	≈ 35	470617
0.01	≈ 60	448448
0.001	≈ 80	426507

Tabla 7.1: Tamaño de la base de conocimiento en relación con el valor de β.

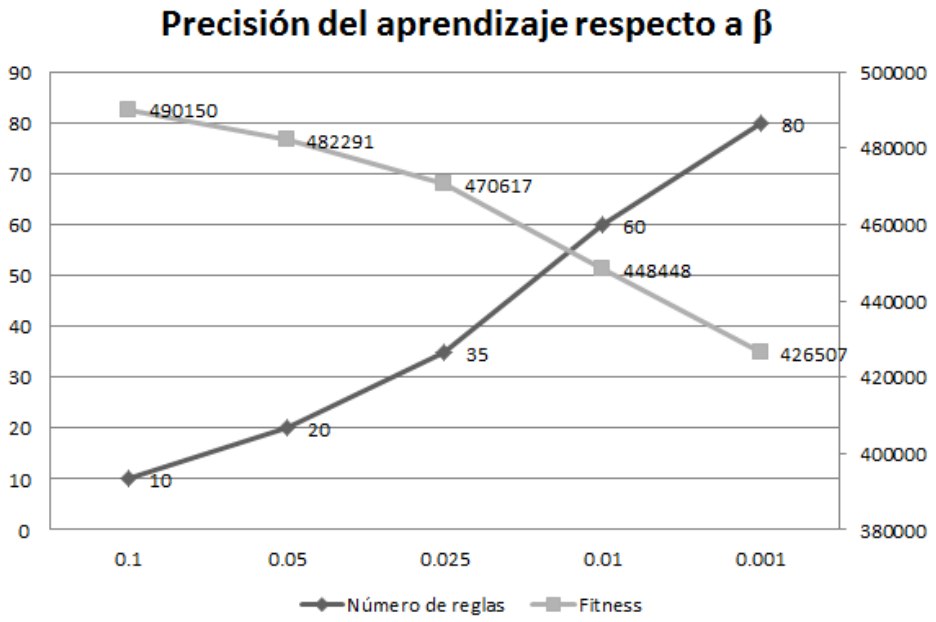

Figura 7.1: Representación del número de reglas y el *fitness* respecto al valor de β en diferentes ejecuciones del algoritmo de aprendizaje.

avanzaban las iteraciones del algoritmo evolutivo, tal y como se muestra en la figura 7.2. Se ajusta perfectamente al comportamiento típico de un algoritmo evolutivo.

A modo de ejemplo, se citan a continuación un par de reglas aprendidas durante la ejecución del algoritmo evolutivo. Las etiquetas borrosas que las forman pueden estar en tres situaciones distintas (figura 7.3) :

- Representando el valor mínimo: en este caso, la tupla de valores que define la etiqueta es del tipo (A, A, A, B), es decir, un triángulo con dos vértices en el valor mínimo de la etiqueta, y uno en el valor máximo. Representa valores muy altos.
- Representando un valor intermedio: la etiqueta está centrada en un valor, y su base abarca el intervalo entre el valor mínimo y el máximo de dicho conjunto borroso. Es una tupla de valores del tipo (A, B, B, C). Representa valores *moderados* de una variable.
- Representando el valor máximo: la etiqueta se define mediante una tupla del tipo (A, B, B, B).

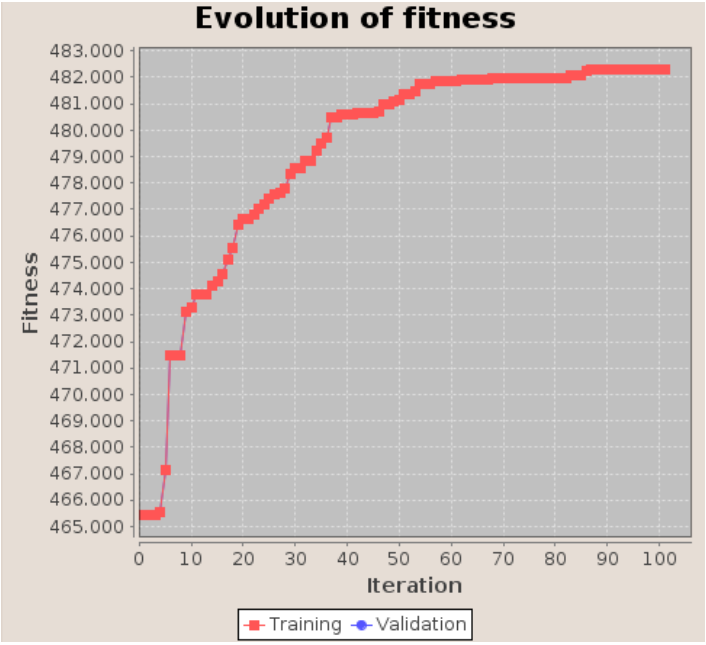

Figura 7.2: Evolución del *fitness* del mejor individuo de la población durante la ejecución del algoritmo.

Figura 7.3: Situaciones que un conjunto borroso puede representar

El vértice de la etiqueta en forma triangular está situado en el valor máximo del conjunto. Representa valores muy bajos.

A continuación se analiza el contenido de un par de reglas que se han extraído de la base de conocimiento aprendida:

```
0 # 1 # 0.0 0.0 0.0 1.0
-1 #
-1 #
0 # 1 # 0.0 0.0 0.0 3.0
0 # 1 # -1.570796 -1.570796 -1.570796 1.570796
0 # 1 # 0.0 0.0 0.0 0.25
```

```
  0.01 0.01 0.01 0.01 w 1.0
  -0.40724340766666667 -0.407243407666 -0.407243407666 -0.407243407666 w 1.0
```

La regla tiene 6 antecedentes y 2 consecuentes. A continuación se representa el significado de cada una de las variables del antecedente:

1. La etiqueta de la variable *distancia frontal* representa un valor *muy bajo*
2. El valor de *distancia izquierda* no se tiene en cuenta
3. El valor de *distancia derecha* no se tiene en cuenta
4. La etiqueta de la variable *distancia al objetivo* representa un valor *muy bajo*
5. La etiqueta de la variable *ángulo al objetivo* representa un valor *muy a la izquierda*
6. La etiqueta de la variable *velocidad del robot* representa un valor *muy bajo*

y del consecuente:

1. La etiqueta de la variable *velocidad lineal* representa un valor *muy bajo*
2. La etiqueta de la variable *velocidad angular* representa un valor *moderadamente a la izquierda*

```
-1 #
-1 #
0 # 1 # 0.0 1.0 1.0 1.0
-1 #
-1 #
0 # 1 # 0.0 0.0 0.0 0.16666666666666666
0.23 0.23 0.23 0.23 w 1.0
0.17453288899999997 0.174532888999 0.174532888999 0.174532888999 w 1.0
```

Para esta regla, el significado del antecedente es:

1. El valor de *distancia frontal* no se tiene en cuenta
2. El valor de *distancia izquierda* no se tiene en cuenta
3. La etiqueta de la variable *distancia derecha* representa un valor *muy alto*
4. El valor de *distancia al objetivo* no se tiene en cuenta
5. El valor de *ángulo al objetivo* no se tiene en cuenta
6. La etiqueta de la variable *velocidad del robot* representa un valor *muy bajo*

y del consecuente:

1. La etiqueta de la variable *velocidad lineal* representa un valor *moderadamente bajo*

2. La etiqueta de la variable *velocidad angular* representa un valor *ligeramente a la derecha*

El resultado que se ha obtenido cuando se ha ejecutado el controlador con la base de reglas aprendida ha sido el que se refleja en la figura 7.4. En este ejemplo de ejecución hay que tener en cuenta que el objetivo se ha situado en el punto $P_F(1, 1)$, partiendo de la posición inicial del robot en $P_0(-5, -3)$. Dado el entorno en el que se ejecuta la simulación, para alcanzar el objetivo debe pasar entre los dos obstáculos que se encuentran en su camino. Como se puede ver en la imagen, el recorrido del robot empieza colocándose con el ángulo adecuado para llegar al objetivo, y posteriormente bordea el obstáculo que se encuentra en su camino, hasta que finalmente alcanza la meta.

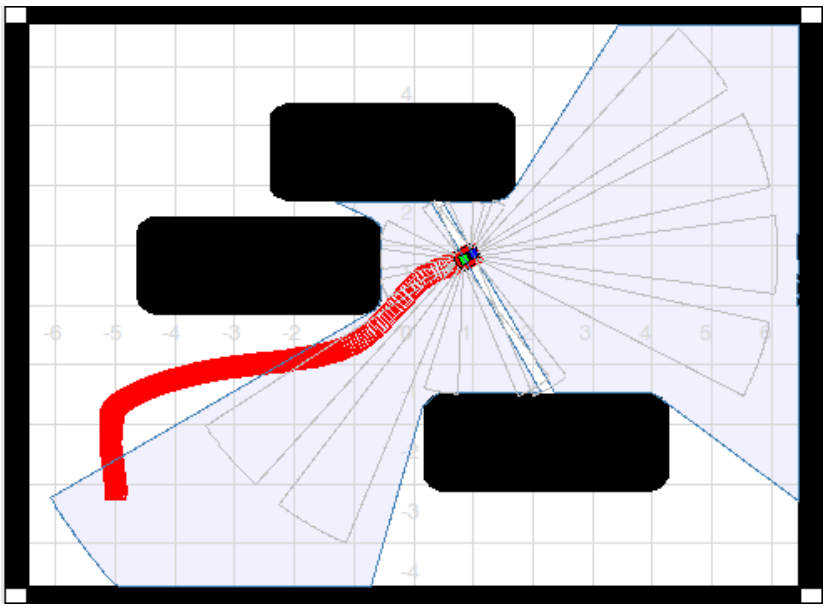

Figura 7.4: Recorrido realizado por el robot para alcanzar el objetivo situado en $P_F(1, 1)$.

Capítulo 8

Conclusiones

El objetivo principal de este Trabajo de Fin de Grado ha sido la realización de una herramienta que permite la navegación autónoma de un robot móvil a través de un entorno no estructurado, donde se pueden encontrar obstáculos no conocidos previamente, en condiciones suficientes de seguridad. Esta herramienta está compuesta de un conjunto de aplicaciones que permiten llevar a cabo la función para la que han sido pensadas inicialmente.

El presente documento, memoria de dicho trabajo, contiene una explicación detallada del conjunto de actividades llevadas a cabo para hacer posible la realización de este proyecto, desde que éste se puso en marcha hasta su etapa de validación. A continuación se enumera el trabajo realizado:

- Se ha realizado la implementación de una herramienta que permite el aprendizaje de una base de reglas borrosas, que modela el comportamiento que debe ejecutar el robot cuando se encuentra en movimiento para alcanzar un punto objetivo. Esta herramienta está basada en un algoritmo evolutivo que selecciona la mejor solución en base a la puntuación que le asigna una función de evaluación definida explícitamente para resolver un problema de navegación.
- La herramienta de aprendizaje permite la utilización de ejemplos durante el proceso de aprendizaje, de tal forma que sirva como referencia para seleccionar la base de reglas que mejor solución dé al problema de navegación.
- Se han creado utilidades para la generación de ficheros de ejemplos a partir de datos obtenidos de una simulación llevada a cabo por la herramienta Player/Stage, completamente compatibles con la herramienta de aprendizaje implementada.
- Por si lo anterior no fuese suficiente para llevar a cabo el aprendizaje, se ha implementado una utilidad que, utilizando la definición de las variables de entrada y salida de la base de reglas, construye un conjunto de ejemplos completo, sobre el que opcionalmente se puede ejecutar un algoritmo que permite seleccionar sólo los más significativos.
- Se ha implementado la herramienta de aprendizaje de tal forma que las reglas de la base de conocimiento puedan tener cualquier definición para el antecedente y el consecuente, siempre que sean reglas de tipo Mamdani, sólo con definirlas dentro de las opciones de la población del algoritmo evolutivo.

- Se ha implementado la posibilidad de ejecutar el proceso de aprendizaje paralelizado, para aprovechar al máximo los recursos del equipo en el que se ejecute y minimizar el tiempo de ejecución del mismo.
- Se ha implementado un controlador borroso que toma los datos de los sensores del robot y los transforma al formato del antecedente de la base de conocimiento que utiliza para realizar la toma de decisiones de control. Su implementación es completamente independiente de la base de reglas utilizada y los puntos de inicio y objetivo definidos.
- Se proporciona la posibilidad de ejecutar el entorno de aprendizaje y las diferentes utilidades mediante una interfaz gráfica, con la intención de hacerlas más amigables y fáciles de utilizar de cara al usuario final de la aplicación.

Todas estas tareas garantizan el cumplimiento de los objetivos iniciales del proyecto, definidos con detalle en la sección 2.5, y de los requisitos establecidos por el cliente, que se han descrito en la sección 4.1. Desde el punto de vista formativo, la realización de este proyecto fue beneficioso para adquirir conocimientos y experiencia sobre el ámbito del proyecto, que inicialmente era desconocido.

Sobre la solución generada existe la posibilidad de realizar mejoras y ampliaciones, de tal forma que se pueda aumentar el número de funcionalidades implementadas, y componer una herramienta cuya calidad sea superior a la de esta versión. Entre las mejoras que se han identificado como posibles son las siguientes:

- En la actualidad las etiquetas borrosas se han generado de manera uniforme a lo largo del universo de discurso de las variables a las que representan. Una posible mejora que influiría muy positivamente en la calidad del aprendizaje sería definir las etiquetas de manera no uniforme.
- Permitir el trabajo con otros tipos de reglas, como las TSK, ya que actualmente las únicas que es posible utilizar son las de tipo Mamdani.

Apéndice A

Manual de usuario

El presente capítulo pretende realizar una explicación breve sobre cómo dar los primeros pasos para poner en funcionamiento la herramienta. Se incluyen una guía de instalación, ejecución y un breve manual de uso.

A.1. Ficheros de configuración

Cuando la herramienta de aprendizaje se ejecuta, busca en el directorio del usuario dos ficheros de opciones, *System.properties* y *Population.properties*, dentro de un directorio *properties*. En el caso de no encontrarlos, crea esta pareja de ficheros, inicializando sus valores de configuración con unos por defecto. En las siguientes secciones se describe detalladamente el contenido de cada uno de ellos, especificando el significado de cada uno de los parámetros que los forman.

A.1.1. El fichero *System.properties*

Contiene las opciones generales del sistema de aprendizaje, cuyos parámetros se describen a continuación:

Parámetro	**Descripción**
fitnessGoal	Valor de *fitness* que se debe alcanzar para que se lleve a cabo la finalización del proceso de aprendizaje
weightAlpha1	Peso asignado al primer componente individual del error en la evaluación
weightAlpha2	Peso asignado al segundo componente individual del error en la evaluación
weightAlpha3	Peso asignado al tercer componente individual del error en la evaluación

securityDistance	Distancia de seguridad alrededor del robot que se considera para el aprendizaje
selectorIndexSeed	Número de semilla aleatoria utilizada para el proceso de selección
mutatorIndexSeed	Lo mismo que en el caso anterior, pero para el proceso de mutación
consequentIndexSeed	Análogo que en los anteriores, pero utilizado para la generación del consecuente aleatoriamente
crosserIndexSeed	Como en los casos anteriores, pero utilizada para el proceso de cruce
granularityIndexSeed	Semilla utilizada para la generación de las granularidades de los individuos en la inicialización
maxAngularAcceleration	Es el valor máximo que la aceleración angular puede tomar. Se utiliza para descartar las acciones aprendidas que no se pueden llevar a cabo por esta limitación
minRangeOfCovering	Grado a partir del cual se considera que una regla se dispara con unos valores del antecedente. Debe ser un valor en el intervalo $[0, 1]$
maxAcceleration	Valor máximo para la aceleración angular de los efectores del robot. Se utiliza para descartar acciones que no se pueden llevar a cabo debido a esta limitación
securityCollisionTime	Tiempo de seguridad establecido para antes de que se produzca una colisión
timeInterOrder	Tiempo que transcurre entre una orden y la siguiente, para poder aplicar el modelo de movimiento del robot
examplesFile	Ruta del fichero que contiene el conocimiento necesario para llevar a cabo el proceso de aprendizaje
randomSeeds	Conjunto de semillas utilizadas para la generación de números aleatorios. Deben estar separadas por comas unas de otras
randomNumber.min	Parámetro utilizado por el generador de números aleatorios, para cuando se creen dentro de un intervalo. Éste valor define su mínimo
randomNumber.max	Análogo al anterior, pero definiendo el máximo del intervalo de generación
rateExamplesValidation	Porcentaje de ejemplos dentro del conjunto que se reservan para validación. De ser 0, los dos *fitness* son iguales. Se expresa mediante un valor en el intervalo $[0, 1]$

tresholdProbabilityCrossing	Probabilidad de cruce entre dos individuos, expresada como un número en $[0, 1]$
tresholdProbabilityMutation	Análogo al parámetro anterior, pero utilizado para definir la probabilidad de mutación de un individuo
betaEvaluatorParameter	Parámetro utilizado en la evaluación para ponderar la importancia que tiene en la puntuación de un individuo su tamaño. Debe estar en el intervalo $[0, 1]$
numThreads	Número de hilos de ejecución que se disparan en el caso de ejecutar el algoritmo paralelizado. Debe ser un número par
priority	Grado de prioridad que se asigna a la ejecución del algoritmo dentro del sistema operativo. Por defecto sus valores son *max* o *min*, en otro caso se le asigna una prioridad normal
parallelized	Parámetro que define si se desea ejecutar el algoritmo con o sin paralelización, puede ser definido a *true* o *false*
numIterations	Condición de parada del algoritmo, que finaliza tras alcanzar las iteraciones que se determinan

Finalmente, se definen una serie de parámetros que son utilizados para la creación del algoritmo evolutivo, donde se definen los componentes que se desean utilizar para cada una de las funciones. En este caso se definen lo que representa cada uno de ellos y además el valor que toman por defecto. Estos parámetros no deberían ser cambiados para asegurar un correcto funcionamiento del algoritmo:

Parámetro	**Descripción**	**Valor**
evaluatorOperator	Clase que se utiliza para el paso de la evaluación de los individuos	*IndividualEvaluator-Alphas*
replacerOperator	Operador de reemplazamiento	*PopulationReplacement*
inputEvaluator	Clase que calcula el grado de disparo de una regla	*InputEvaluatorWeighted-Mean*
mutatorOperator	Implementación del operador de mutación de un individuo	*IndividualMutator*
createOperator	Operador que inicializa la población con la que el algoritmo de aprendizaje va a trabajar	*PopulationInitiator*
selectorOperator	Clase que contiene la implementación del operador de selección de la población	*IndividualSelector-Tournement*

crossOperator	Implementación del operador de cruce que se va a utilizar en el algoritmo	*IndividualCrosser*
decoderOperator	Nombre de la clase que implementa el operador de decodificación del genotipo de un individuo en su fenotipo	*IndividualDecoder*
activationDegreeCalculator	Clase que implementa el método de cálculo para la obtención del grado de activación de una etiqueta borrosa	*ActivationRuleCalculator-Minimum*

A.1.2. El fichero *Population.properties*

Define las características de la población utilizada para el proceso de aprendizaje. Los parámetros que contiene son los siguientes:

Parámetro	**Descripción**
consequentGranularities	Define el conjunto de granularidades máximas para las variables del consecuente de las reglas. Deben ir separados por comas
minConsequent-Granularities	Análogo al parámetro anterior, pero para representar las granularidades mínimas del consecuente, se definen del mismo modo que el parámetro anterior
inputRanges	Define el rango de las variables de entrada. Cada uno de los rangos se especifica en el formato $[inicio\sharp fin]$ y separados por comas
outputRanges	Es análogo al parámetro anterior, pero definiendo los rangos de las variables de salida. Se definen de la misma forma que en el caso de las variables de entrada
numSectors	Número de sectores en los que se divide el láser que obtiene las entradas
rangeSector0 ,*rangeSector1*, *rangeSector2*	Para cada sector en los que se divide el láser, se tiene un campo que define su ángulo de inicio y su ángulo de fin, que deben ir detallados de la forma $inicio, fin$
inputNames	Define los nombres de las variables de los antecedentes. Deben ir separados por comas

outputNames	Análogo al parámetro anterior, pero representando los nombres de los consecuentes. Se definen del mismo modo que en el caso anterior
sizeOfSelection	Este parámetro contiene el tamaño que debe tener la selección de los individuos para el cruce
selectionWith-Replacement	Representa si la operación de selección tiene activada la opción de hacerlo con o sin reemplazamiento de individuos
sizeOfTournament	Define el tamaño del torneo utilizado para la selección
maxNumberOfIndividuals	Es el tamaño que tiene la población, utilizado para realizar el reemplazamiento de la población
minAntecedent-Granularity	Granularidad mínima que pueden tener cualquiera de las variables del antecedente de las reglas
maxAntecedent-Granularity	Análogo al parámetro anterior, pero representando la granularidad máxima
generateGranularitiesBy	Método mediante el cual se generan las granularidades de los antecedentes. Este es un parámetro que no se debe cambiar para asegurar el correcto funcionamiento del programa
maxRulesIndividual	Representa el tamaño máximo que pueden tener los individuos de la población
ruleSize	Define el número de variables que contiene el antecedente de las reglas
sizeOfConsequent	Define el tamaño del consecuente, esto es: el número de las variables que tiene el consecuente

A.2. Ejecución de las herramientas

A.2.1. Generador de ejemplos

Existen dos maneras de ejecutar la herramienta de generación de ejemplos, mediante interfaz gráfica o sin ella. Si se ejecuta mediante la interfaz, sólo se debe ejecutar el archivo *.jar* que la contiene. Esto se puede realizar mediante la interfaz gráfica del sistema operativo, o ejecutando el siguiente comando en consola:

```
java -jar aplicacionEjemplos.jar
```

Lo que dará como resultado la aparición de la ventana principal de la aplicación:

En esta herramienta se pueden distinguir dos modos de ejecución. Se puede acceder a ellos, bien desde la ventana principal de la aplicación, o bien seleccionando la opción correspondiente en el menú *Options*. Son los siguientes:

Figura A.1: Ventana principal de la herramienta de generación de ejmplos.

- Generador de ejemplos a partir de *logs* de Player/Stage. Se puede acceder a él en el menú *Options* → *Player/Stage Interpreter Mode*. La ventana de la aplicación que se muestra es la que se muestra en la figura A.2.

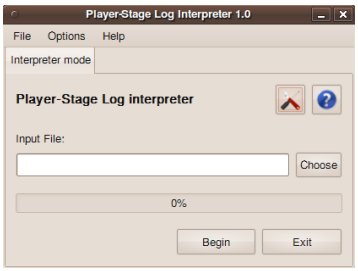

Figura A.2: Vista de la herramienta en el modo de intérprete de *logs* de Player/Stage.

  Como se puede ver en la figura A.2, la herramienta permite la selección de un archivo de entrada, y el comienzo del procesado del archivo seleccionado. El archivo de salida se selecciona al empezar la ejecución del procesado. Se pueden cambiar algunas opciones de funcionamiento si se selecciona la opción representada mediante el icono de las herramientas.

- Generador de ejemplos sintéticamente. Es accesible desde el menú *Options* → *Sintetic Generator Mode*. La vista que se muestra (figura A.3) sólo permite la selección de un archivo de salida donde guardar los ejemplos generados.

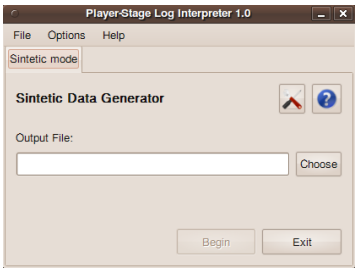

Figura A.3: Vista de la herramienta en el modo de generador de ejemplos sintéticos.

La ejecución de esta herramienta también se puede realizar sin modo gráfico, esto es, desde consola. En este caso, se puede hacer mediante el comando:

```
java -jar aplicacionEjemplos.jar -interpreter archivoEntrada archivoSalida
```

para ejecutarlo en modo intérprete de *logs* de Player/Stage, o bien mediante el comando:

```
java -jar aplicacionEjemplos.jar -sintetic archivoSalida
```

para hacerlo en modo generador de ejemplos sintéticos. En cualquier caso, la ayuda de ejecución del programa está disponible ejecutando:

```
java -jar aplicacionEjemplos.jar -h
```

A.2.2. Herramienta de aprendizaje

Del mismo modo que en el caso de la herramienta de generación de ejemplos, la aplicación se puede ejecutar a través de una interfaz gráfica o sin ella. Si lo que se desea es realizarlo mediante la primera opción, debe ejecutarse el fichero *.jar* a través del modo gráfico del sistema operativo, o en consola a través del comando:

```
java -jar herramientaAprendizaje.jar
```

De realizarlo de esta forma, se abrirá la ventana principal de la aplicación (figura A.4), donde se puede ver, mediante una gráfica, la evolución del *fitness* de la población en cada una de las iteraciones del algoritmo. En la parte derecha de la pantalla se puede ver un sistema de control para seleccionar qué información se ve en dicha gráfica.

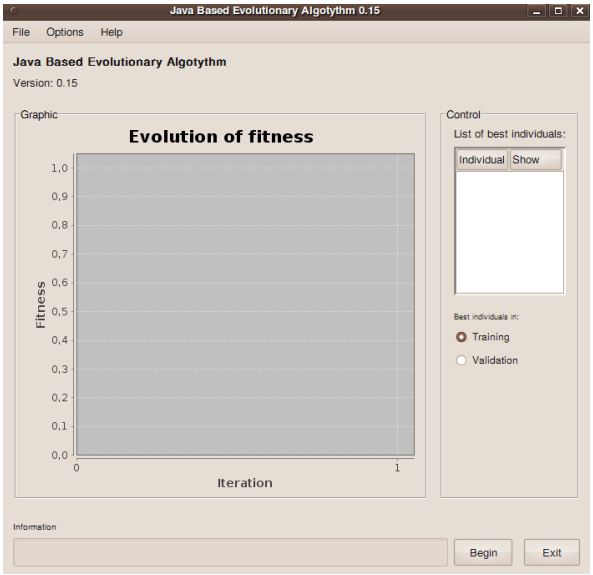

Figura A.4: Vista de la ventana principal de la herramienta de aprendizaje.

Se pueden modificar algunos parámetros de ejecución de este algoritmo de modo directo mediante la interfaz gráfica, esto se puede realizar mediante la apertura de la ventana de opciones, en el menú *Options* → *Edit Preferences*. En dicha ventana hay dos pestañas, la que corresponde a la modificación de los parámetros de la población, en la pestaña *Population* (figura A.5) y los parámetros del sistema, en la pestaña *System* (figura A.6).

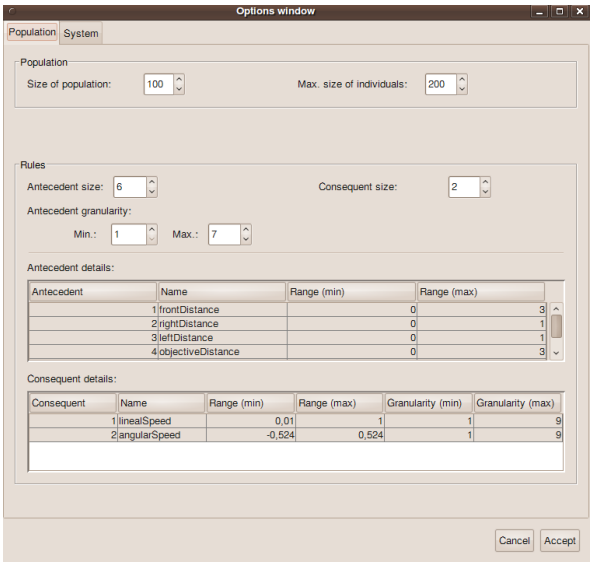

Figura A.5: Vista de la pestaña *Population* en la ventana de opciones.

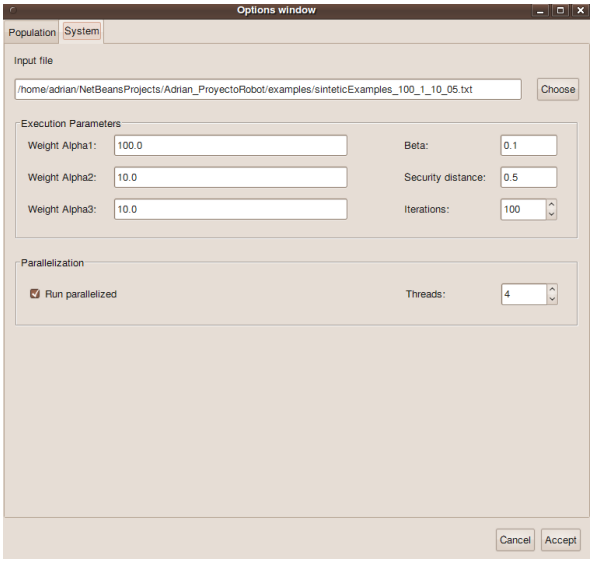

Figura A.6: Vista de la pestaña *System* de la ventana de opciones.

Si se desea ejecutar desde la consola del sistema, sin utilizar la interfaz gráfica que proporciona la herramienta, también es posible, mediante el comando:

```
java -jar herramientaAprendizaje.jar outputFile
```

La ayuda de ejecución de esta aplicación, en caso de ser necesaria, es accesible mediante la ejecución de:

```
java -jar herramientaAprendizaje.jar -h
```

A.3. Controlador borroso

Se ejecuta mediante el comando:

```
./controlador baseReglas inicioX inicioY objX objY
```

Para poder ejecutarse, necesita que esté en funcionamiento el programa Player/Stage, que se ejecuta mediante el comando:

```
robot-player archivoConfiguracion
```

La instalación de esta herramienta, en distribuciones Linux basadas en Debian, se puede realizar directamente desde los repositorios, instalando los siguientes paquetes:

```
sudo apt-get install robot-player stage robot-player-dev
```

Bibliografía

[1] *The Gazebo Project.* `http://playerstage.sourceforge.net/doc/Gazebo-manual-0.8.0-pre1-html/`, August 2007.

[2] J. E. Smith A. E. Eiben. *Introduction to Evolutionary Computing.* Natural Computing Series. Springer, 2007.

[3] David W. Aha, Dennis Kibler, and Marc K. Albert. *Instance-Based Learning Algorithms*, volume 6 of *Machine Learning.* Kluwer Academic Publishers, 1991.

[4] Brian Gerkey et al. *The Player Robot Device Interface.* `http://playerstage.sourceforge.net/doc/Player-2.1.0/player/`, September 2005.

[5] David Gilbert and Thomas Morgner. *JCommon API Documentation.* JFree.org, `http://www.jfree.org/jcommon/api/index.html`, April 2009.

[6] David Gilbert and Thomas Morgner. *JFreeChart API Documentation.* JFree.org, `http://www.jfree.org/jfreechart/api/javadoc/index.html`, April 2009.

[7] IEEE Computer Society. *Recommended Practice for Software Requirements Specifications*, `http://ieeexplore.ieee.org/servlet/opac?punumber=5841`, 1998.

[8] IEEE Computer Society. *Standard for Software Life Cycle Processes - Risk Management*, `http://ieeexplore.ieee.org/servlet/opac?punumber=7300`, 2001.

[9] IEEE Computer Society. *Standard for Developing a Software Project Life Cycle Process*, `http://ieeexplore.ieee.org/servlet/opac?punumber=11045`, 2006.

[10] J.M. Mendel L.X. Wang. Generating fuzzy rules by learning from examples. *IEEE Trans. on Systems, Man, and Cybernetics*, 22:1414–1427, 1992.

[11] M. Mucientes and J. Casillas. Quick design of fuzzy controllers with good interpretability in mobile robotics. *IEEE Transactions on Fuzzy Systems*, 15(4):616–632, August 2007.

[12] F. Hoffmann L. Magdalena O. Cordón, F. Herrera. *Genetic Fuzzy Systems: evolutionary tuning and learning of fuzzy knowledge bases*, volume 19 of *Advances in Fuzzy Systems - Applications and Theory.* World Scientific, 2001.

[13] The Object Management Group, `http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/`. *UML 2.3 Infrastructure Specification*, May 2010.

[14] Jennifer Owen. *The Player/Stage Tutorial 2nd Edition*. http://www-users.cs.york.ac.uk/~jowen/player/playerstage-tutorial-manual.pdf, April 2010.

[15] William C. Wake Steven John Metsker. *Design Patterns in Java*. Pearson Education, 2006.

[16] Sun Microsystems, http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html. *Javadoc 5.0 Tool Guide*, 2004.

[17] Tigris.org, http://argouml-stats.tigris.org/documentation/manual-0.30/. *ArgoUML User Manual*, 2009.

[18] Richard Vaughan et al. *The Stage Robot Simulator*. http://playerstage.sourceforge.net/doc/Stage-3.2.1/, October 2009.