Edición Semiautomática de Comportamientos en Entornos de Videojuegos

proyecto de investigación presentado por

Gonzalo Flórez Puga

dirigido por la profesora

Mª Belén Díaz Agudo

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Septiembre 2007

Índice general

1. Motivación y objetivos **7**

2. Estado del arte **11**

2.1. Edición de comportamientos en distintos entornos de simulación 11
2.2. Técnicas para la especificación de comportamientos 15
2.2.1. Sistemas de razonamiento basado en reglas 15
2.2.2. Máquinas de estado y autómatas finitos 17
2.2.3. Redes neuronales 21
2.2.4. Algoritmos evolutivos 23
2.2.5. Técnicas de planificación 24
2.3. Introducción al razonamiento basado en casos 28
2.3.1. El ciclo CBR 29
2.3.2. CBR textual 36
2.4. Aprendizaje y juegos 40
2.4.1. JV^{2}M 42
2.5. Herramientas para la edición de comportamientos 45
2.5.1. BrainFrame 45
2.5.2. Simbionic 47
2.5.3. MindEditor y Replicant Toolkit 47

3. Edición y generación semiautomática de comportamientos **51**

3.1. El editor gráfico de comportamientos 52
3.1.1. Modelos de juego 52
3.1.2. El editor de comportamientos 53
3.1.3. Generadores de código 59
3.2. El sistema de razonamiento basado en casos 61
3.2.1. Consultas por funcionalidad 61
3.2.2. Integración en el editor 67
3.2.3. Consultas por estructura 69
3.2.4. El sistema CBR fuera del editor 72
3.3. Procedimiento de aplicación 74

4. Evaluación del generador de comportamientos **81**

4.1. Resultados del sistema de CBR 81
4.2. Integración con Soccerbots 85
4.3. Integración con JV^{2}M 90
4.3.1. El modelo de Far Cry 90
4.3.2. El modelo de JV^{2}M 91

4.4. Integración con Neverwinter Nights 96
4.5. Integración con otros entornos de simulación 99

5. Conclusiones y líneas futuras de trabajo 101
5.1. Líneas futuras de investigación 102

Apéndice A: Ficheros XML 107

Índice de figuras

2.1. Esquema de un comportamiento 13
2.2. Integración de un sistema de reglas en un videojuego 15
2.3. Ejemplo de autómata finito 18
2.4. Ejemplo de máquina de Mealy 18
2.5. Ejemplo de máquina de Moore 19
2.6. Máquina de estado jerárquica y jerarquía 21
2.7. Esquema de una neurona artificial 21
2.8. Red neuronal 22
2.9. Formulación de un plan en GOAP [40] 25
2.10. Descomposición de tareas 27
2.11. El ciclo CBR [12] 30
2.12. Suposición en los sistemas de CBR 32
2.13. CBR textual 37
2.14. Representación vectorial de documentos y consultas 39
2.15. Ciclo learning-by-doing 41
2.16. Captura de pantalla de JV^{2}M 44
2.17. Estructura de BrainFrame 45
2.18. El editor gráfico de BrainFrame 46
2.19. Diagrama de bloques de Simbionic 47
2.20. Estructura de Replicant Toolkit 48
2.21. Captura de pantalla de MindEditor 49

3.1. Diagrama de bloques del editor de comportamientos 52
3.2. Estructura de un modelo de juego 54
3.3. Captura de pantalla del editor de comportamientos 56
3.4. Página de propiedades de un nodo atómico 57
3.5. Página de propiedades de un nodo compuesto 58
3.6. Página de propiedades de un conector 58
3.7. Herramientas disponibles en el editor 59
3.8. Clase abstracta generador 60
3.9. Comportamiento con dos transiciones para la misma entrada 60
3.10. Estructura de la base de casos 64
3.11. Relación entre los atributos y los nodos de un comportamiento 66
3.12. Parámetros de una consulta a la base de casos 68
3.13. Resultados de la recuperación de los casos 68
3.14. Formulario para salvar un comportamiento 69
3.15. Guardar un caso en la base de casos 70
3.16. Ejemplo de un patrón de comportamiento 70
3.17. Interfaz gráfica de usuario para realizar consultas por estructura 71

3.18. Adaptación de un nodo en un comportamiento recuperado 72
3.19. Perdida de consistencia en las aristas al sustituir un nodo 72
3.20. Diagrama de clases del sistema CBR 73
3.21. Comportamiento de ejemplo 75
3.22. Pantalla de inicio 75
3.23. Carga del modelo de juego 76
3.24. Configuración de los nodos 76
3.25. Configuración del comportamiento `Ir a pelota` 77
3.26. Consulta CBR 78
3.27. Resultados de la búsqueda 79
3.28. Configuración de las aristas 79
3.29. Resultado de la edición del comportamiento 79
3.30. Generador de código 80

4.1. Resultados de la búsqueda de un delantero 82
4.2. Resultados de la búsqueda de un portero 84
4.3. Adaptación de los comportamientos recuperados 84
4.4. Dimensiones del terreno de juego de Soccerbots 86
4.5. RCEI en Neverwinter Nights 98

5.1. Uso de modelos de juego genéricos 103

Capítulo 1

Motivación y objetivos

El objetivo que persigue cualquier juego es el de ofrecer entretenimiento. Este cometido se puede lograr de muy diversas formas. En el caso de los videojuegos, por ejemplo, una presentación gráfica impactante o una buena historia pueden hacer que el juego sea entretenido, pero normalmente esto no es suficiente. Es necesario, además, que el juego suponga un desafío para el jugador. Un método adecuado para conseguir esto es dotando de inteligencia a los contrincantes (y también a los aliados) del jugador [8].

La creación de comportamientos inteligentes no es una tarea trivial. Se trata en muchos casos de una labor difícil y tediosa. En primer lugar, es necesario identificar qué entidades del juego necesitan mostrar un comportamiento inteligente y qué tipo de comportamiento debe ser (si debe ser agresivo con el jugador, debe ayudarle, huir de él, etc.). A continuación hay que diseñar cada uno de los comportamientos y, por último, implementarlo e integrarlo en el juego. A esto hay que sumar el impedimento de que, en muchas ocasiones, los diseñadores del juego no tienen los conocimientos técnicos necesarios para programar los comportamientos dentro del mismo, por lo que es necesario que comuniquen sus ideas a un programador para que este realice el último paso.

Una solución parcial a este problema son los lenguajes de `script`. Los lenguajes de `script` son lenguajes de programación simples (poseen generalmente tipado dinámico y poco estricto) y, en la mayoría de los casos, son interpretados, lo que evita tener que compilar y enlazar la aplicación cada vez que se modifican. Son muy adecuados para partes del juego que requieran personalización o modificaciones frecuentes [9]. Al ser más simples que los lenguajes de programación tradicionales, resultan más fáciles de aprender.

Para facilitar el diseño de los comportamientos se pueden tener en cuenta dos factores, comunes a la mayoría de los videojuegos existentes. En primer lugar, los comportamientos son modulares. Un comportamiento, especialmente si se trata de un comportamiento complejo, se puede descomponer en subcomportamientos más simples. En segundo lugar, y derivado de lo anterior, los comportamientos más simples suelen ser comunes a varios comportamientos complejos dentro del mismo juego, e incluso dentro de diferentes juegos del mismo género. Por ejemplo, en el caso de un juego de fútbol, el comportamiento “defender” podría estar compuesto por los comportamientos “ir hacia el balón” y “despejar”, mientras que otro comportamiento “atacar” utilizaría, por ejemplo, “ir hacia el balón”, “driblar” y “tirar a puerta”. Esta característica se puede aprovechar para ahorrar trabajo a la hora de construir un nuevo comportamiento, utilizando los comportamientos básicos como bloques de construcción para construir los comportamientos más elaborados.

Como respuesta a estas necesidades se propone en este trabajo la construcción de un editor gráfico de comportamientos, capaz de almacenar y reutilizar comportamientos pre-

viamente diseñados, y con un diseño genérico, en el sentido de que sea capaz integrarse con diferentes juegos.

El prototipo inicial de este editor basa la construcción de comportamientos en el uso de máquinas de estado finito jerárquicas. Las máquinas de estado finito son una herramienta adecuada, versátil y útil para representar comportamientos. Entre sus ventajas destaca que se pueden representar gráficamente. Además, su representación gráfica es intuitiva y muy descriptiva. Por otro lado, la teoría en la que se basan las máquinas de estado finito está formalizada y ha sido estudiada desde mediados del pasado siglo [38, 22]. A todo esto hay que añadir que el paso de una máquina de estado finito a código ejecutable es un proceso automatizable, y existen en la literatura diversas aproximaciones que ofrecen resultados de gran eficiencia en su funcionamiento [17, 5, 27].

Además, sería conveniente que el editor fuera capaz de aprovechar la modularidad de los comportamientos que se van a crear. Para esto, se vale de la combinación de dos técnicas.

Mediante las máquinas de estado finito jerárquicas, mencionadas anteriormente, se puede definir un comportamiento como una jerarquía de máquinas de estado en la que cada uno de los estados puede contener otra máquina de estado. De esta manera se permite la integración, de un modo sencillo, de comportamientos simples dentro de otros comportamientos más elaborados.

La otra técnica empleada en el editor es el razonamiento basado en casos o CBR (del inglés Case Based Reasoning). El CBR consiste en un conjunto de técnicas para el desarrollo de sistemas basados en el conocimiento que, para solucionar un problema, reutiliza las soluciones a problemas parecidos previamente resueltos. Las técnicas de CBR utilizan en su beneficio el hecho de que los problemas tienden a repetirse y, por lo tanto, la experiencia en problemas anteriores es un recurso muy valioso [12].

Para resolver un problema mediante CBR, se parte de un conjunto de problemas previamente resueltos, llamados *casos*. Dentro de esta *base de casos* se buscan los problemas de mayor similitud con el se quiere resolver. Esta fase se conoce con el nombre de *recuperación*. A continuación, la solución al problema recuperado debe *adaptarse* a los requisitos del problema propuesto. Por último, el nuevo problema resuelto puede añadirse a la base de casos para que que este disponible para posteriores consultas. Es lo que se conoce como fase de *aprendizaje*.

En resumen, el uso de máquinas de estado finito jerárquicas nos da el "hueco" donde alojar los comportamientos básicos que conforman comportamientos más complejos, y el CBR resulta una herramienta inestimable a la hora de buscar y adaptar comportamientos ya implementados para que se adecúen a las necesidades del diseñador, "rellenando" así estos huecos.

En una fase inicial de uso, el editor permitirá construir comportamientos basados en un conjunto de primitivas básicas, pues carece de un conjunto de problemas resueltos que sean utilizados como referencia. Con el uso, además de las primitivas básicas, se tiene acceso a un conjunto de constructores de mayor granularidad, en la forma de los comportamientos previamente editados. El sistema de CBR permite acceder de forma eficiente a estos constructores.

A grandes rasgos, los requisitos planteados inicialmente que debe cumplir el editor son tres: sencillez de uso, aplicabilidad a distintos entornos de simulación y asistencia al usuario en la creación de comportamientos.

Sencillez de uso

Una de las principales ideas hacia las que se ha movido el diseño del editor es la separación entre los roles del programador/diseñador del juego y el diseñador de comportamientos para los personajes, de manera que para crear los comportamientos in-

teligentes no sean necesarios concimientos técnicos sobre el juego para el que se están diseñando ni sobre su arquitectura.

Aplicabilidad

Otro objetivo fundamental es que la generación de comportamientos no se limite a un solo juego o género de juegos, sino que permita, utilizando diferentes modelos de juego, que se pueda generalizar su uso a cualquiera de ellos.

Asistencia al usuario

Dado que la creación de comportamientos inteligentes puede ser una tarea bastante tediosa se pretende ofrecer al usuario herramientas que faciliten su creación. Así pues, se proporciona una interfaz de búsqueda que emplea razonamiento basados en casos para encontrar y reutilizar comportamientos creados con anterioridad, que cumplan determinados criterios de búsqueda proporcionados por el usuario, ya sea de manera explícita, mediante una consulta directa, o implícita, basándose en las acciones del usuario.

Gracias al editor de comportamientos se proporciona un formato intermedio, basado en una representación gráfica sencilla, que facilita el diseño y la implementación de comportamientos inteligentes. De esta manera se acelera el proceso de pruebas y corrección de la inteligencia artificial en entornos de simulación y videojuegos, permitiendo prototipados más rápidos y visuales.

Las técnicas anteriormente expuestas y otras estudiadas durante el desarrollo del trabajo se tratan en mayor detalle en el capítulo 2, en el que adicionalmente se presentan varios entornos de simulación y las diferentes metodologías empleadas para desarrollar comportamientos. En el capítulo 3 se describe el prototipo de aplicación desarrollado para el presente trabajo y su funcionamiento, tratando por separado el proceso de edición manual de comportamientos y el uso del razonamiento basado en casos para obtener nuevos comportamientos a partir de otros implementados previamente. En el capítulo 4 se muestran algunos de los resultados obtenidos por el editor aplicado a diferentes entornos de simulación. Finalmente, en el capítulo 5 se exponen las conclusiones y se tratan las futuras líneas de trabajo.

Capítulo 2

Estado del arte

2.1. Edición de comportamientos en distintos entornos de simulación

La utilización de técnicas de inteligencia artificial ha estado ligada a los videojuegos prácticamente desde el origen de estos, hacia los años 70. En sus inicios, se aplicaba en máquinas que funcionaban con monedas, en juegos como *Pac-Man*, *Pong* o *Space Invaders*, y su finalidad era conseguir que el jugador siguiera introduciendo dinero en la máquina. Este tipo de juegos utilizaban conjuntos de reglas simples y acciones, combinadas con toma de decisiones aleatoria para conseguir comportamientos menos predecibles. A partir de entonces, la aplicación de técnicas de inteligencia artificial ha acompañado en su evolución a los videojuegos, tímidamente al principio, convirtiéndose con el paso del tiempo en un componente fundamental y diferenciador.

Los videojuegos se pueden clasificar en géneros dependiendo de diversos factores, pero no existe una clasificación aceptada universalmente. De hecho, no existe un acuerdo sobre qué criterios se deben emplear para definir esta clasificación. A esto se suma la dificultad de que, en muchos casos, un juego no pertenece a una única categoría, sino que puede tomar características propias de diferentes géneros.

En [42], Marc Prensky propone la siguiente clasificación:

- Acción: en esta categoría se incluyen los juegos de *scroll* lateral[1], juegos de laberintos como *Pac-man*, juegos de plataformas[2] (*Wonder Boy*, *Donkey Kong* o la saga de *Super Mario*), carreras de coches y persecuciones (excepto simuladores), y otros juegos, como *Missile Command* en el que el jugador debe disparar a diferentes objetos que caen desde la parte superior de la pantalla. Posteriormente, se han sumado a este género los juegos de acción en tres dimensiones, en los que el jugador controla un personaje, desde un punto de vista subjetivo (juegos de acción en primera persona) o colocado justo detrás de él (juegos de acción en tercera persona), que interactúa con un entorno tridimensional. Algunos ejemplos son *Wolfstein 3D*, *Doom*, *Quake* o *Medal Of Honor*.

- Aventura: juegos caracterizados por la investigación, que suelen incluir exploración del entorno, interacción con distintos personajes y objetos, integrados en el entorno

[1] Juegos generalmente en dos dimensiones, en los que el personaje atraviesa la pantalla horizontalmente de izquierda a derecha, mientras se enfrenta a diferentes obstáculos y enemigos.

[2] Juegos en los que el avatar del jugador debe ir saltando de una plataforma a otra, mientras esquiva a sus enemigos.

y manejados por el juego, resolución de rompecabezas, etc. Este género tuvo gran auge durante los años 80 y 90, gracias a los juegos de aventura basados en texto (por ejemplo, *Forbidden Quest* o *La aventura original*) en primer lugar y, posteriormente, a las llamadas aventuras gráficas (las sagas de *Monkey Island*, *Space Quest* o *Sam and Max*).

- Rol: este género procede de los juegos de rol de mesa, como *Dungeons & Dragons* o *Rune Quest*, en los que cada jugador es representado por un personaje con un conjunto de aptitudes que determinan las tareas que puede hacer y su habilidad para realizarlas, lo que se denomina la ficha del personaje. Estos juegos se desarrollan principalmente en la imaginación del jugador, sin necesidad de utilizar un tablero. En su conversión al mundo de los videojuegos, los juegos de rol han conservado la mayoría de sus características. Siguen manteniendo el uso de fichas de personaje donde se reflejan sus habilidades, que se pueden mejorar mediante la experiencia, suelen desarrollarse en mundos de fantasía y dan gran importancia a la interacción con otros personajes, tanto que se ha creado un nuevo subgénero llamado MMORPG o *Massive Multiplayer Online Role Playing Games* (Juegos de Rol Multijugador Masivo en línea) en el que gran cantidad de jugadores (cientos o incluso miles) se relacionan socialmente dentro del mundo en el que se desarrolla el juego. Algunos ejemplos de videojuegos de rol son *Baldur's Gate* y *Neverwinter Nights*, basados en *Dungeons & Dragons*, *Diablo*, *Ultima* o *World Of Warcraft*, uno de los últimos y más exitosos exponentes de MMORPG.

- Lucha: en este género de videojuegos el jugador se enfrenta con un personaje controlado por la máquina o por otro jugador en una lucha un contra uno. Generalmente, el jugador puede escoger su avatar entre diferentes personajes. Cada uno de ellos tiene diferentes habilidades ofensivas y defensivas en forma de *combos*, combinaciones de movimientos que dan lugar a ataques más potentes. A diferencia de los juegos de rol, en este género, las habilidades del avatar del jugador se corresponden directamente con la habilidad del jugador para manejar los mandos. Ejemplos de este género son la saga *Street Fighter* o *Tekken*.

- Rompecabezas: en estos juegos el jugador debe resolver diferentes rompecabezas lógicos, poniendo a prueba su habilidad, su agilidad mental o su memoria. *Lemmings*, *Tetris* o, más recientemente, *Brain Training*, son algunos ejemplos de este tipo de videojuegos.

- Simulación: el objetivo de estos juegos es imitar de una forma más o menos fidedigna una experiencia que se puede realizar en el mundo real. Dentro de esta categoría tienen cabida juegos de conducción, simuladores de vuelo, empresariales o simuladores sociales (simulación de ciudades o pequeñas comunidades donde interactúan diferentes agentes). Algunos ejemplos son *X-Plane*, *Sim City*, *F1 Challenge* o *The Sims*.

- Deportes: en este caso, es el contenido del juego y no la forma de jugarlo el factor determinante del género del juego. En este género se incluyen los juegos deportivos en los que el jugador controla a un deportista o a un equipo, bien sea a través de los dispositivos de entrada habituales o mediante otros dispositivos que intentan representar elementos del deporte en cuestión, como, por ejemplo *Manx TT Superbike*, un videojuego de carreras de motos en el que el mando de control es una motocicleta sobre la que se sitúa el jugador y debe inclinarse hacia los lados para girar.

- Estrategia: son juegos que se basan más en la capacidad para tomar decisiones por parte del jugador que en su habilidad para manejar los controles o en el azar. En general,

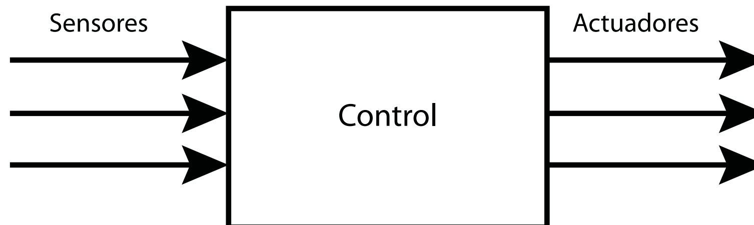


Figura 2.1: Esquema de un comportamiento

todos los participantes en el juego tienen el mismo grado de conocimiento sobre sus elementos al principio del mismo, y comienzan en situación similar, por lo que están en igualdad de condiciones para ganar. Dentro de este género se puede distinguir entre estrategia en tiempo real, donde las acciones de los distintos jugadores se desarrollan en paralelo, y estrategia por turnos, en los que los participantes se van turnando uno tras otro para realizar las acciones. Algunos ejemplos de juegos de estrategia son *Command & Conquer* o la serie *Age of Empires*, dentro de los juegos en tiempo real, y *Civilization* o *Heroes of Might and Magic* en el subgénero de estrategia por turnos.

Generalmente, la ejecución de un juego consiste en la ejecución de un bucle, que recibe el nombre de bucle de juego. Este bucle se repite continuamente, hasta que el usuario decide salir del juego. En cada iteración se realizan todas las tareas necesarias para el desarrollo del juego, como la captura de la entrada de los periféricos de control (ratón, teclado, mando de juego...), el cálculo de la influencia del modelo físico en las entidades del juego (lo que en terminología de videojuegos se conoce como la *física* del juego), el *render* (la representación gráfica por pantalla del entorno de juego) o la ejecución de los comportamientos de las entidades del juego. En cuanto a la estructura del bucle, existen varias aproximaciones que ofrecen distintas ventajas e inconvenientes [47], si bien, queda fuera del alcance de este trabajo profundizar en las características de cada una de ellas.

Un comportamiento se puede definir como un conjunto de acciones o reacciones que realiza una entidad, generalmente en relación con su entorno. Esto es particularmente cierto en el ámbito de los videojuegos. En términos generales, una entidad dentro de un videojuego recopila información sobre el entorno mediante una serie de sensores, equiparables a los sentidos de los seres vivos, y, dependiendo de ésta, realiza determinadas acciones. Para llevar a cabo estas acciones utiliza los actuadores.

En los sistemas sensoriales de los juegos, la inteligencia artificial observa periódicamente el entorno en el que habita, en contraste con los sentidos reales, que son estimulados independientemente de si se desea o no (no podemos dejar de oír o de ver a voluntad). De esta manera se pueden emular las capacidades sensoriales limitando la cantidad de recursos utilizada para ello [35].

El módulo de control es el responsable de decidir qué actuadores es necesario activar en cada momento para un determinado estado de los sensores, como se muestra en la figura 2.1. Es en el módulo de control donde se ubicará la inteligencia artificial que guía el comportamiento. La interfaz que utiliza la inteligencia artificial para comunicarse con el juego será precisamente el conjunto de sensores, a través de los que recibe la información acerca de su entorno, y el conjunto de actuadores, que le permiten llevar a cabo las acciones necesarias.

En general, el conjunto de sensores y actuadores es único y diferente para cada juego, pero dentro de un mismo género de juegos existen una serie de características y acciones comunes que suelen darse en todos ellos.

Tipo	Parámetros	Descripción
Agresivo	$distancia = 100, salud = 10$	Una entidad agresiva intenta atacar aunque el jugador esté lejos, y no huye hasta que su salud está muy baja.
Huidizo	$distancia = -1, salud = 100$	Una entidad huidiza no ataca nunca al jugador. Huye de él en todos los casos ya que la salud máxima que puede tener es 100.

Tabla 2.1: Comportamientos parametrizados mediante ficheros de configuración

Existen diversas formas de incorporar un comportamiento al bucle de juego. Cada una de ellas aporta diferentes ventajas e inconvenientes.

- Integrado como parte del código: en este caso, los comportamientos se incluyen dentro del código fuente del juego. De esta manera, su ejecución es más eficiente, ya que lo que se ejecuta finalmente es código compilado. Por otra parte, cualquier modificación de un comportamiento, por pequeña que sea, implica volver a compilar todo el programa.

- Mediante `scripts`: implementando los comportamientos en un lenguaje de `scripts` permite realizar modificaciones sobre ellos sin que sea necesario volver a compilar la aplicación y sin renunciar a la flexibilidad que ofrecen los lenguajes de programación de alto nivel. El principal inconveniente es que son lenguajes interpretados, lo que resta eficiencia a su ejecución. Además, es necesario incluir un intérprete del lenguaje en el juego.

- Mediante ficheros de configuración: como solución intermedia, se puede utilizar una o varias plantillas de comportamiento, incluidas en el código, que se adaptan a cada entidad mediante un conjunto de parámetros incluidos en un fichero de configuración. Un ejemplo muy simple de esta aproximación puede ser un comportamiento en el que, cuando la distancia al jugador es menor que un determinado valor, la entidad va hacia él y le ataca, pero si su salud es menor que otro valor, se aleja para recuperarse. Los valores de *distancia* y *salud* se cargan de un fichero y pueden definir distintos tipos de comportamiento, como los que se muestran en la tabla 2.1. Esta aproximación aporta algunas de las ventajas y de los inconvenientes de las dos anteriores. Por un lado, dado que las plantillas forman parte del código compilado, su ejecución es más eficiente que en el caso de los `scripts`. Además, se pueden llevar a cabo modificaciones sobre los comportamientos ajustando los parámetros de configuración. En ese caso no será necesario volver a compilar todo el juego. El inconveniente más importante es que carecen de la flexibilidad de los lenguajes de alto nivel, dando lugar, cada plantilla, a comportamientos en cierto modo similares, aun cuando se modifiquen los parámetros de configuración.

Se observa que, en todo caso, la manera de incluir un comportamiento en un juego siempre pasa por, o bien utilizar ficheros de configuración, lo cual resta bastante flexibilidad, o implementarlo en algún lenguaje de programación. Puede tratarse de un lenguaje de programación clásico, como C++ o Java, un lenguaje de `scripts`, como Lua o Python, o un lenguaje desarrollado específicamente para el juego en cuestión, como QuakeC, desarrollado para Quake. En cualquiera de estos tres casos, el diseñador de comportamientos, que en muchos casos no es un programador, debe aprender y utilizar con cierta destreza un lenguaje de programación.

Una posible solución a este problema es usar un entorno de diseño que utilice una representación intermedia de los comportamientos que sea más adecuada para los diseñadores de

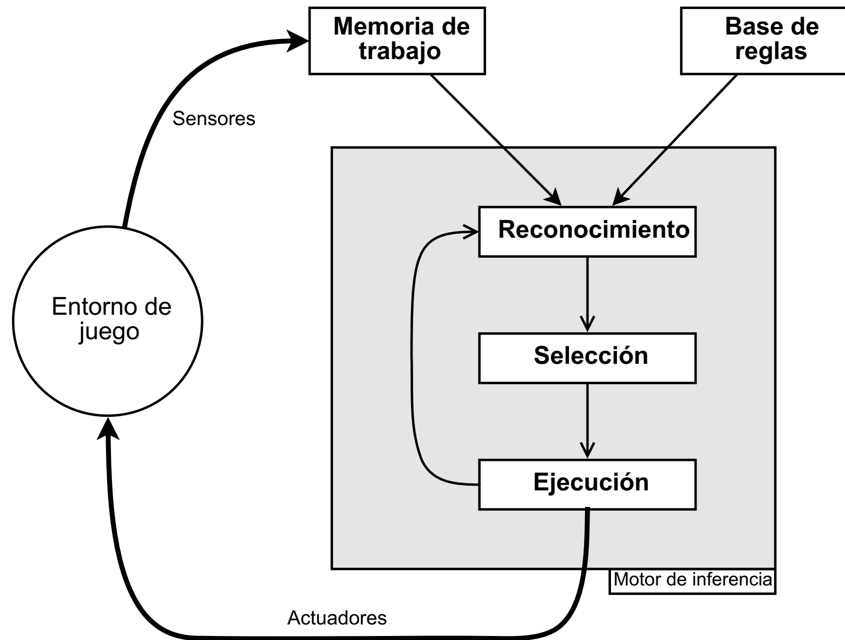


Figura 2.2: Integración de un sistema de reglas en un videojuego

comportamientos. Este entorno debe conocer, al menos, el conjunto de sensores y actuadores del juego y permitirá, en última instancia, generar la implementación del comportamiento en alguna de las formas mencionadas anteriormente.

En cuanto a la representación intermedia, lo más habitual es escogerla de entre las distintas técnicas de diseño de comportamientos. En general, algunas de las propiedades deseables de la representación intermedia es que sea conocida por los diseñadores, que sea fácilmente entendible (por ejemplo, que tenga una representación visual sencilla) y que el paso a la implementación se pueda realizar de una manera sencilla.

En lo referente al diseño de comportamientos, existen multitud de técnicas que resultan apropiadas para esta tarea. Cada una de ellas tiene sus características y puede ser más o menos adecuada según el caso. En la siguiente sección se revisan algunas de estas técnicas.

2.2. Técnicas para la especificación de comportamientos

Desde que se vio como una necesidad incluir inteligencia artificial en los videojuegos para potenciar algunos de sus aspectos, se han utilizado diferentes técnicas, más o menos sofisticadas, y con mayor o menor éxito.

En esta sección se analizan algunas de estas técnicas, y su uso para la representación de comportamientos inteligentes. Las técnicas estudiadas son los sistemas de reglas (2.2.1), las máquinas de estado finito (2.2.2), las redes neuronales (2.2.3), los algoritmos evolutivos (2.2.4) y algunas técnicas de planificación (2.2.5).

2.2.1. Sistemas de razonamiento basado en reglas

Los sistemas de razonamiento basado en reglas son una técnica de representación ampliamente utilizada. Basan su funcionamiento en la utilización de reglas del tipo:

SI `condición` **ENTONCES** `acción`

que establecen patrones (situación, acción) que indican al sistema cómo actuar en una determinada situación.

Los sistemas de reglas están formados por tres elementos:

- Una base de reglas, que contiene conocimiento en forma de reglas de producción como las mencionadas anteriormente.
- Una base de hechos o memoria de trabajo, que contiene una representación del estado actual.
- Un motor de inferencia, una colección de algoritmos que controlan la actividad del sistema.

La interpretación de las reglas se realiza en ciclos de operación. Cada ciclo consta de tres fases:

1. Reconocimiento: durante esta fase se comparan las condiciones de las reglas con la memoria de trabajo, para saber así cuáles de ellas son aplicables.
2. Selección: de entre todas las reglas reconocidas como aplicables en la fase anterior se selecciona una de ellas.
3. Ejecución: se ejecuta la acción asociada a la regla seleccionada. Esta acción provocará cambios en la memoria de trabajo, que hará que sea posible seleccionar nuevas reglas.

Los sistemas de reglas pueden utilizarse para representar comportamientos dentro de un videojuego. Para ello, se pueden crear reglas en las que se formulen condiciones sobre el conjunto de sensores que ofrece el juego y acciones que se correspondan con la activación de actuadores. La memoria de trabajo sobre la que actúan las reglas vendrá dada por el estado del juego en cada momento, es decir, por los valores de los sensores. La figura 2.2 muestra cómo se integra el sistema de reglas dentro de videojuego.

A continuación se enumeran algunas de las ventajas e inconvenientes más importantes de los sistemas de reglas:

- Ventajas:
  - Existe una separación clara entre el conocimiento, dado por las reglas, y el control, mediante el motor de inferencia. Además, la parte de control es altamente reutilizable. Diferentes sistemas tendrán diferentes bases de reglas, pero el mismo motor de inferencia.
  - Cada regla es una unidad de conocimiento que puede ser añadida, modificada o eliminada independientemente del resto del conjunto de reglas.
  - Las reglas pueden ser una forma bastante natural de expresar el conocimiento, especialmente en el dominio de los comportamientos en videojuegos, si tratamos con comportamientos reactivos[3].
  - El modelo es independiente de la representación escogida para las reglas y la memoria de trabajo.
- Inconvenientes:

[3]Los agentes reactivos son aquellos que, en cada instante de tiempo, sólo evalúan cual será su próxima acción, y lo hacen en función del contexto actual.

- La ejecución del proceso de reconocimiento de patrones para las condiciones de las reglas es, en general, bastante ineficiente.
- Dada una base de reglas, es difícil determinar cuáles son las acciones que va a provocar. Aunque cada regla es fácilmente entendible por separado, se pierde la visión global del conjunto de reglas.
- Para construir un comportamiento complejo la base de reglas puede crecer mucho, haciendo que sean muy difíciles tareas como el mantenimiento o la depuración.

2.2.2. Máquinas de estado y autómatas finitos

Una máquina de estado finito es un modelo de computación con una cantidad limitada de memoria, que recibe el nombre de *estado*. Cada máquina tiene un número finito de estados posibles y una función de transición que determina cómo cambia el estado en el tiempo de acuerdo a las entradas [9].

Se pueden diferenciar dos categorías de máquinas de estado:

- Autómatas finitos: no generan una salida hasta alcanzar un estado terminal. Suelen usarse para reconocimiento de patrones o clasificación de secuencias.
- Máquinas de estado finito: ofrecen una salida cada vez que se consume un símbolo de la entrada.

Autómatas finitos

Un autómata finito puede definirse mediante una tupla de cinco elementos: $A = \langle \Sigma, Q, \delta, q_0, F \rangle$. Donde:

- Σ es el alfabeto de entrada (es decir, el conjunto de valores que se pueden encontrar a la entrada del autómata).
- Q es un conjunto finito de estados.
- δ es la función de transición $\delta : Q \times \Sigma \longrightarrow Q$. Mediante esta función se asigna un estado final q a cada par de estado original p, valor de entrada v, de manera que, cuando el estado del autómata es p y se encuentra en la entrada el valor v, el estado pasa a ser q: $\delta(p, v) = q$.
- $q_0 \in Q$ es el estado inicial del autómata.
- $F \subseteq Q$ es el conjunto de estados finales o de aceptación.

Gráficamente, un autómata finito se suele representar mediante un grafo dirigido [9]. Los nodos del grafo representan los diferentes estados, mientras que las aristas, etiquetadas con los elementos de Σ, representan las transiciones. Así pues, una arista que va desde el nodo p al q, etiquetada con el valor v representará la transición $\delta(p, v) = q$.

En el autómata representado en la figura 2.3,

- $\Sigma = \{v, w\}$
- $Q = \{p, q\}$
- $\delta(p, v) = q$
- $\delta(p, w) = p$

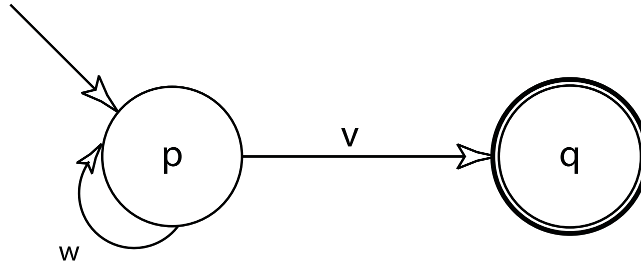


Figura 2.3: Ejemplo de autómata finito

- $q_0 = \{p\}$
- $F = \{q\}$

Para las combinaciones de estado y entrada que no están definidas para la función δ se considera que existe una transición a un estado de no aceptación del que no se puede salir [30].

Máquinas de estado finito

Las máquinas de estado finito (FSM o Finite State Machine) se diferencian de los autómatas finitos en que pueden producir una salida mientras se está procesando la entrada. Por lo tanto, deja de tener sentido el conjunto de estados de aceptación, y, en su lugar, se añaden una función de salida y un alfabeto de salida.

Se definen mediante una tupla de seis elementos: $A = \langle \Sigma, Q, \delta, q_0, O, \lambda \rangle$. Los cuatro primeros valores son los mismos que en el caso de los autómatas finitos. Con respecto a los dos restantes, O es el alfabeto de salida (el conjunto de símbolos que pueden formar la salida) y λ es la función de salida.

Existen dos tipos de máquinas de estado que difieren en su función de salida.

Máquinas de Mealy En las máquinas de Mealy la función de salida λ depende del estado actual y del símbolo en la entrada: $\lambda : Q \times \Sigma \longrightarrow O$.

Gráficamente, la función de salida suele representarse en las aristas, junto con la entrada a la que se asocia

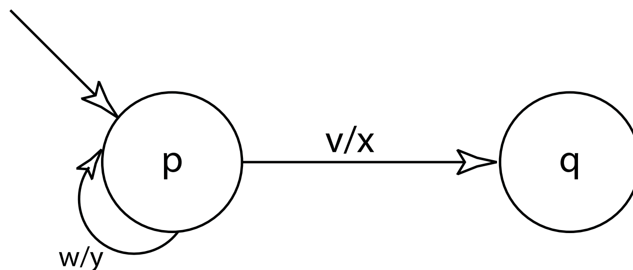


Figura 2.4: Ejemplo de máquina de Mealy

En el ejemplo de la figura 2.4,

- $O = \{x, y\}$
- $\lambda(p, v) = x$
- $\lambda(p, w) = y$

Máquinas de Moore Las máquinas de Moore se diferencian de las anteriores en que la función de salida sólo depende del estado actual, pero no de la entrada.

Esto se refleja en la representación gráfica etiquetando los nodos con el valor de salida.

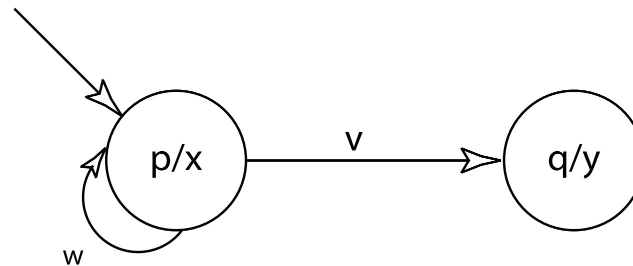


Figura 2.5: Ejemplo de máquina de Moore

Se puede ver un ejemplo de máquina de Moore en la figura 2.5, donde:

- $O = \{x, y\}$
- $\lambda(p) = x$
- $\lambda(q) = y$

Uso de máquinas de estado en la representación de comportamientos

Generalmente, las máquinas de estado finito se emplean para generar una salida en función de una cadena de entrada. Esta salida puede indicar si se reconoce o se acepta la cadena de entrada. Pero la salida también puede emplearse como un mecanismo de control.

En el caso de la representación de comportamientos, la salida se interpreta como una sucesión de llamadas a los actuadores y la entrada debe ser la información recibida por los sensores. Por tanto, en las aristas se encontrarán condiciones sobre los sensores que provocarán una transición de estados cuando se hagan ciertas; en los nodos habrá llamadas a los actuadores que deberán ser activadas al alcanzar el estado en cuestión.

El uso de máquinas de estado para la representación de comportamientos ofrece varias ventajas. Las más importantes se enumeran a continuación:

- Son intuitivas y simples de implementar, diseñar y visualizar.
- La teoría subyacente está formalizada.
- Han sido probados para implementar la inteligencia artificial en diversos juegos y aplicaciones con buenos resultados.

Frente a estas ventajas también se encuentran otros inconvenientes:

- Cuando son grandes, al añadir muchos estados y transiciones, pueden llegar a ser complejas y farragosas.

- Existen determinadas tareas simples para las que no resultan adecuadas, por ejemplo, contar.

- Su diseño no puede modificarse en ejecución.

Para resolver el primero de estos problemas se introducen las máquinas de estado finito jerárquicas, de las que se habla en la siguiente sección, que simplifican en gran medida el diseño de las máquinas de estado cuando estas alcanzan ciertas dimensiones, además de ofrecer otras ventajas.

A partir de una máquina de estados se puede obtener un sistema de reglas equivalente. En [11] se muestra un método sistemático para hacerlo. Según este método, se generan dos grupos de reglas, uno para controlar las transiciones entre estados y otro para obtener la salida. Además de las condiciones de las aristas, se añade una condición sobre el estado. Las estructura de las reglas para la salida es la siguiente:

$$\textbf{SI } estado = < estado > \textbf{ ENTONCES } < salida >$$

y la estructura de las reglas para las transiciones entre un estado de origen y un estado de destino:

$$\textbf{SI } estado = < estado\ origen > \wedge < transicion > \textbf{ ENTONCES}$$
$$estado = < estado\ destino >$$

Supongamos que partimos de una máquina de estados como la de la figura 2.5. Las reglas asociadas a las salidas serán:

$$\textbf{SI } estado = p \textbf{ ENTONCES } x$$
$$\textbf{SI } estado = q \textbf{ ENTONCES } y$$

Las reglas asociadas a las transiciones:

$$\textbf{SI } estado = p \wedge v \textbf{ ENTONCES } estado = q$$
$$\textbf{SI } estado = p \wedge w \textbf{ ENTONCES } estado = p$$

Máquinas de estado finito jerárquicas

Una máquina de estado finito jerárquica (HFSM o Hierarchical Finite State Machine) es muy parecida en su estructura a una máquina de estado finito clásica. Se diferencia de estas en que cada estado puede contener otra máquina de estado subordinada. Estos estados contenedores se denominan *superestados*. De esta manera se define una jerarquía de máquinas de estado en la que cada estado anidado es un componente que da una funcionalidad propia, mientras que la máquina de estado superior ensambla estos componentes de la manera más adecuada para proporcionar una funcionalidad más extensa [9].

A un nivel fundamental las HFSMs no aportan nada al modelo de computación. Además, tampoco reducen el número de estados, sino que los aumenta, al introducir los nuevos superestados. Lo que sí se reduce de manera drástica es el número de transiciones, lo que hace de estos modelos más simples, resultando así más fácil su representación y análisis [23]. De esta manera se soluciona el primero de los inconvenientes mencionados en la sección anterior.

A continuación se detallan algunas de las propiedades de las HFSMs:

- Abstracción: La jerarquía proporciona distintos niveles de detalle para cada una de las máquinas de estado. Al no mostrar los niveles más bajos del árbol jerárquico se pueden ocultar los detalles, considerando el superestado como una *caja negra*.

- Refinación: Se puede afrontar el diseño empezando por los niveles más altos, definiendo superestados abstractos (por ejemplo, comportamientos compuestos de otros comportamientos más simples), para a continuación proceder de la misma forma dentro de cada uno de los superestados. De esta manera se puede seguir un diseño de arriba abajo (top-down).

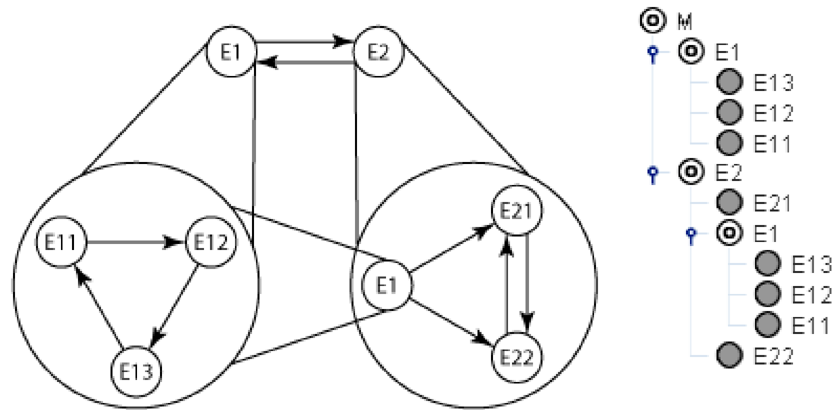


Figura 2.6: Máquina de estado jerárquica y jerarquía

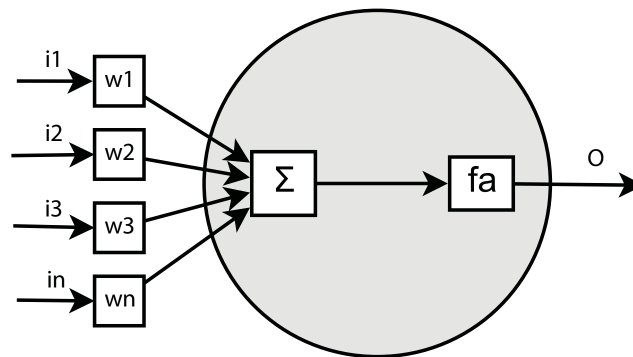


Figura 2.7: Esquema de una neurona artificial

- Modularidad: Cada máquina de estado puede tratarse como un componente modular y, por lo tanto, puede reutilizarse en cualquier lugar y en cualquier nivel de la jerarquía.

Para representar comportamientos inteligentes, las HFSMs se emplean de la misma forma que las FSMs: en las aristas se representarán las condiciones sobre los sensores mientras que en los nodos se encontrarán, bien llamadas a los actuadores si se trata de un nodo simple, o bien una máquina de estado si se trata de un supernodo.

2.2.3. Redes neuronales

Una red neuronal artificial, a la que nos referiremos en adelante simplemente como red neuronal, es un modelo de simulación basado en una simplificación del funcionamiento del cerebro. Una red neuronal está formada por un conjunto de neuronas artificiales conectadas entre sí y procesa la información usando una aproximación conexionista a la computación[4].

Una neurona de nuestro cerebro, está compuesta por un conjunto de dendritas, que recogen señales de otras neuronas, y un axón, responsable de transmitir la señal a las dendritas

[4]El conexionismo trata de modelar los procesos mentales complejos como procesos emergentes en redes interconectadas de elementos simples [1].

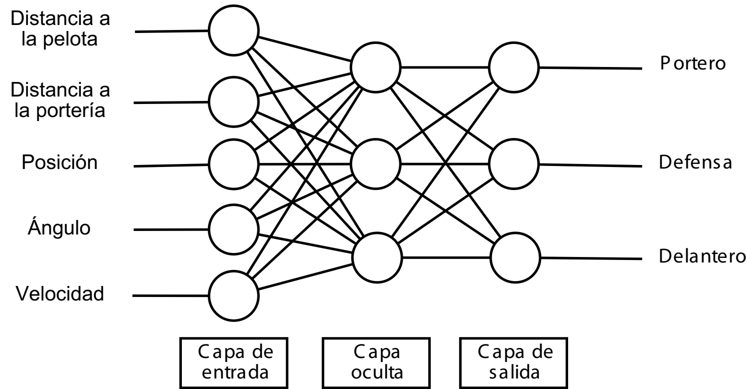


Figura 2.8: Red neuronal

de otras neuronas. De la misma forma, una neurona artificial, como la mostrada en la figura 2.7, a la que, para simplificar, llamaremos neurona, tiene un conjunto de entradas y una salida. Además, cada entrada tiene asociado un peso. Para calcular el valor total de la entrada a una neurona se aplica una función de activación, que suele ser la suma de las entradas ponderada por sus pesos:

$$I = B + \sum_{n}^{j=1} i_j \cdot w_j$$

Al resultado de la función suele añadirse un valor de corrección (bias) B para desplazar la función en el eje de las entradas [46].

Para obtener la salida de la neurona, se aplica una función de activación f_a al valor de entrada I. Por lo tanto, la salida O será igual a:

$$O = f_a(I) = f_a\left(\sum_{n}^{j=1} i_j \cdot w_j\right)$$

Las funciones de activación más habituales son la función escalón, la lineal y la sigmoidal.

Las redes neuronales basan su funcionamiento en el paso de las señales de activación de una neurona a otra. Las neuronas se organizan en tres tipos de capas:

- Una capa de entrada, que consta de neuronas que reciben las diferentes entradas a la red.
- Una capa de salida, formada por tantas neuronas como valores posibles de salida.
- Cero o más capas ocultas, que son las capas intermedias entre la entrada y la salida. Cada una de estas capas puede estar formada por un número arbitrario de neuronas.

Las neuronas de cada capa se conectan con alguna o todas las neuronas de la capa siguiente. Cuando la red recibe una entrada, en forma de activación de algunas de las neuronas de la capa de entrada, calcula, según los pesos y la función de activación que posean, sus valores de salida, que recibirán las neuronas conectadas en la siguiente capa. Esta activación se sucede hasta llegar a la capa de salida.

En la figura 2.8 se muestra un ejemplo de red neuronal.

Una de las características más potentes de las redes neuronales es su capacidad de aprendizaje. Este puede realizarse según distintos paradigmas. Generalmente, la red neuronal se

inicializa con unos valores aleatorios para los pesos de entradas a las neuronas. Si el aprendizaje es supervisado, se parte de un conjunto de pares de ejemplo y solución, que se suministran a la red distintos ejemplos para evaluar su actuación. Por cada ejemplo, se compara la salida obtenida de la red con la salida que debería obtenerse y se ajustan los pesos en consecuencia. En el caso de aprendizaje no supervisado, se proporciona un conjunto de ejemplos y una función de coste y se trata de ajustar los pesos de manera que se minimice la función de coste. Por último, en el aprendizaje por refuerzo, no se proporciona ningún conjunto de ejemplos, sino que estos se obtienen interactuando con el entorno. Cuando la red da unos resultados razonablemente buenos, o cuando se han realizado un determinado número de ejemplos, la red está entrenada y los valores de los pesos se bloquean. Existen diversos algoritmos que automatizan el ajuste de pesos de manera eficiente, como los algoritmos evolutivos o el enfriamiento simulado (simulated annealing).

Las redes neuronales tienen infinidad de aplicaciones en distintos dominios, siendo las más comunes el reconocimiento de patrones, la predicción o el aprendizaje. Dentro del ámbito de los videojuegos, pueden emplearse, entre otras cosas, para representar comportamientos. En la entrada a la red se coloca un subconjunto relevante de los sensores y la salida que devuelve es el comportamiento más adecuado para el estado suministrado. El aprendizaje para este tipo de redes puede ser supervisado, proporcionando el estado de la entrada y el comportamiento que debe presentar en la salida, o por refuerzo. En este último caso, el agente aprende mientras se desenvuelve en el entorno de juego, y el refuerzo, positivo o negativo, viene dado por lo buena o mala que sea su actuación. En la figura 2.8 se muestra un ejemplo de una red neuronal para un comportamiento. En ella, los sensores considerados relevantes para el comportamiento son la distancia a la pelota, la distancia a la portería, la posición, el ángulo y la velocidad. En función de los valores de estos sensores, la red activará en la salida uno de los tres posibles comportamientos contemplados: portero, defensa o delantero.

En [11] se muestra un método para transformar una máquina de estado finito en una red neuronal equivalente. El método implica, en primer lugar, transformar la máquina de estado en un sistema de reglas. A continuación, se crea un nodo de entrada por cada variable en las condiciones de las reglas. Debe haber al menos uno por cada estado. El número de nodos de salida es igual al número de estados. Por último, se añaden los nodos de la capa oculta, que son los que realizan las comparaciones existentes en los antecedentes de las reglas.

2.2.4. Algoritmos evolutivos

El término *algoritmos evolutivos* se refiere a una familia de algoritmos de optimización y búsqueda de propósito general, que utilizan como modelo la evolución y la selección natural.

Un algoritmo evolutivo trabaja con una población de individuos, cada uno de los cuales representa una solución al problema que se desea resolver. Cada individuo se representa mediante un conjunto de valores llamados genes. Esta población se somete a un proceso iterativo de transformaciones y de selección que favorece a los individuos "más adaptados". Cada una de estas iteraciones constituye una generación. Después de un número de generaciones, el mejor individuo de la población se encontrará cerca de una solución óptima al problema.

La estructura general de cualquier algoritmo evolutivo es un ciclo iterativo:

1. Inicializar la población.

2. Evaluar los individuos actuales mediante una función de aptitud, que indica la calidad de solución que representan.

3. Selección de los progenitores, los individuos que se van a reproducir.

4. Aplicación de los operadores de transformación sobre los progenitores. Algunos de los operadores más habituales son la mutación (alteración de alguno de sus genes) y el cruce (obtener nuevos individuos combinando los genes de dos progenitores).
5. Formar una nueva población reemplazando individuos de la población original con individuos de entre el grupo de progenitores y transformados.
6. Volver al paso 2.

Las aplicaciones de los algoritmos evolutivos en el ámbito de los videojuegos son numerosas; en general, pueden aplicarse para cualquier tarea de optimización, especialmente cuando el conocimiento del dominio es escaso o el conocimiento de los expertos es difícil de codificar para reducir el espacio de búsqueda o cuando los métodos tradicionales de búsqueda fracasan [45].

Aunque no son una técnica adecuada para la representación de comportamientos, pueden ser útiles para optimizar alguna de las otras técnicas presentadas, como para obtener los pesos en las redes neuronales.

Otro ejemplo en el que pueden resultar útiles es para ajustar los valores en una representación de comportamientos mediante parámetros. Por ejemplo, en un juego de rol, un personaje debe escoger el conjunto de armas, protecciones y habilidades más adecuado para enfrentarse con sus enemigos. Mediante un algoritmo evolutivo se pueden obtener estos valores, empezando por una configuración aleatoria o predefinida de los mismos y combinándolos en cada generación. En este caso, la función de aptitud sería el resultado de enfrentar o simular un enfrentamiento del personaje con sus enemigos.

Uno de los principales inconvenientes de este tipo de algoritmos es que son caros computacionalmente hablando, por lo que su utilidad en videojuegos, o, al menos, su uso en tiempo de ejecución, es limitada.

2.2.5. Técnicas de planificación

En general, en la implementación de comportamientos para videojuegos, se utilizan aproximaciones reactivas, en las que los agentes reaccionan ante las situaciones del juego mediante una acción o una secuencia de acciones predefinidas [48]. Esta aproximación obliga a que los diseñadores de los comportamientos piensen de antemano en todas las posibilidades que pueden darse y cómo deben reaccionar en ellas los agentes.

Otra posibilidad es utilizar agentes deliberativos, los cuales tienen su propia representación interna del entorno y son capaces de planificar una secuencia de acciones y llevarlas a cabo para conseguir sus fines.

Los planificadores clásicos, como STRIPS (Stanford Research Institute Problem Solver), tratan la planificación como un problema de búsqueda, intentando encontrar un plan que cambie el estado actual del entorno al estado deseado. Estos planificadores, dado un estado inicial, tratan de encontrar un plan para que se cumplan un conjunto de objetivos. Un plan es una secuencia de operadores, cada uno de ellos formado por:

- Una lista de precondiciones, que deben ser ciertas para que se pueda aplicar el operador.
- Una lista de proposiciones que se añadirán al estado actual al aplicar el operador.
- Una lista de proposiciones que se eliminarán del estado actual al aplicar el operador.

Estos sistemas asumen que el estado del mundo no cambia entre la planificación y la ejecución del plan, lo que supone una limitación, especialmente en el ámbito de los videojuegos [48]. Cuando hay un cambio en el estado que hace que el plan no se completar, es necesario realizar un nuevo plan, partiendo del estado actual. Es lo que se conoce como replanificación.

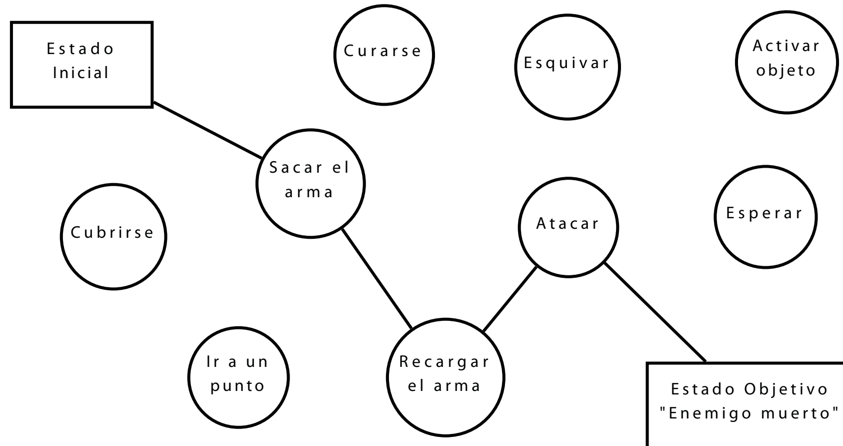


Figura 2.9: Formulación de un plan en GOAP [40]

Planificación de acciones orientada a objetivos

La planificación de acciones orientada a objetivos, o GOAP (Goal-Oriented Action Planning), es una técnica de planificación que permite a los personajes decidir, no sólo qué hacer, sino también cómo hacerlo [39]. Mediante GOAP, pueden encontrar soluciones alternativas a situaciones encontradas durante el juego y pueden gestionar dependencias no previstas durante el diseño.

La "libertad" que otorga GOAP a los personajes en el juego permite que estos muestren comportamientos menos repetitivos y predecibles, y puedan adaptar sus acciones para ajustarse a la situación en la que se encuentran [40].

Dada una situación en el juego, se determina un plan, un conjunto de acciones a ejecutar y el orden apropiado, para satifacer alguno de los objetivos del personaje.

Un objetivo es una condición que un agente desea satisfacer. Cada agente puede tener uno o varios objetivos, pero en un determinado instante de tiempo, sólo uno de ellos estará activo. El objetivo activo es el que controla el comportamiento del agente.

Los objetivos definen qué condiciones deben satisfacerse. Para que sean satisfechas se utilizan las acciones. Una acción es un paso atómico dentro de un plan que hace que el personaje "haga algo". Puede ser ir a una posición, dar una patada a un balón, acelerar un coche, etc.

En cada acción se indica en qué condiciones es posible ejecutarla (precondición) y qué efectos tendrá su ejecución en el estado del mundo. De esta manera se proporciona un mecanismo de secuenciamiento para las acciones.

Para generar un plan, el agente proporciona un objetivo a un sistema de generación de planes o planificador, que busca una secuencia en el espacio de acciones que haga cambiar el estado inicial hasta el estado objetivo. El agente sigue el plan hasta que se complete, se invalide o hasta que cambie de objetivo. Si cambia el objetivo o si, por algún motivo, no se puede continuar con el plan actual, se aborta el plan actual y es necesario generar uno nuevo.

En la figura 2.9 se muestra un plan para matar a un enemigo, en el que se pasa del estado inicial al estado objetivo `Enemigo muerto` mediante las acciones `Sacar el arma`, `Recargar el arma` y `Atacar`.

En cierto modo, este proceso se puede ver como un problema de búsqueda de caminos.

El planificador debe encontrar un camino en el espacio de acciones desde el estado inicial hasta el estado objetivo. Cada acción supone un paso en este camino, que cambia el estado del mundo. Las precondiciones de las acciones determinan cuándo es válido pasar de una acción a la siguiente. De hecho, la búsqueda de un plan puede implementarse mediante el algoritmo A* [40].

En muchos casos puede existir más de un plan válido. Pueden asociarse costes a las acciones de manera que el planificador obtenga el camino de menor coste.

A continuación se exponen algunas ventajas del uso de GOAP [40]:

- Comportamientos adaptativos: si un personaje puede generar diferentes planes para lograr un objetivo en tiempo de ejecución, puede ajustar sus acciones a su entorno actual, y encontrar diferentes soluciones a un problema. Utilizando el ejemplo de la figura 2.9, si el objetivo es matar a un enemigo, el plan puede ser el mostrado:

  ```
  Sacar el arma(lanzagranadas)
  Recargar el arma()
  Atacar(enemigo)
  ```

  Pero si el personaje no tiene en su poder ningún arma, el plan no se podría ejecutar. Lo que ocurre en GOAP es que se busca un plan alternativo. Por ejemplo, supongamos que existe una torreta con un láser cerca del personaje. En ese caso un posible plan alternativo sería:

  ```
  Ir a(torreta)
  Activar objeto(torreta)
  Atacar(enemigo)
  ```

- Comportamientos más simples de diseñar: un comportamiento en el que se prevé cada situación posible y se realiza una acción en consecuencia puede ser bastante difícil de mantener. En el ejemplo anterior, habría que tener en cuenta si el personaje lleva armas o no; en el caso de que no las lleve, si existe una torreta cerca y, si es así, acercarse hasta ella y atacar a su enemigo. Mediante GOAP, es el planificador quien se se encarga de analizar las posibles alternativas existentes y encontrar la mejor opción.

- Comportamientos variables: dado que los planes para lograr un objetivo no están definidos de antemano, sino que se elaboran en tiempo de ejecución según el estado actual del mundo, los personajes pueden mostrar distintos comportamientos incluso para realizar las mismas tareas. El planificador posee un conjunto de acciones que utiliza para formular un plan. Los distintos personajes que aparecen en un juego pueden utilizar diferentes conjuntos de acciones o asignarles diferentes costes, que den lugar a planes distintos.

Redes jerárquicas de tareas

Los planificadores jerárquicos se diferencian de los planificadores tradicionales en que, primeramente, se formula un plan en un espacio de problemas abstracto. Este plan no puede realizarse directamente, ya que está expresado en términos de tareas de alto nivel por lo que se refina, para producir un plan completo en términos de operadores básicos, es decir, de acciones que el agente puede realizar directamente [48].

Por ejemplo, para pasar de un estado inicial a un estado en el que el enemigo ha muerto, se puede trazar un plan compuesto por una tarea de alto nivel `Matar enemigo`. Como se

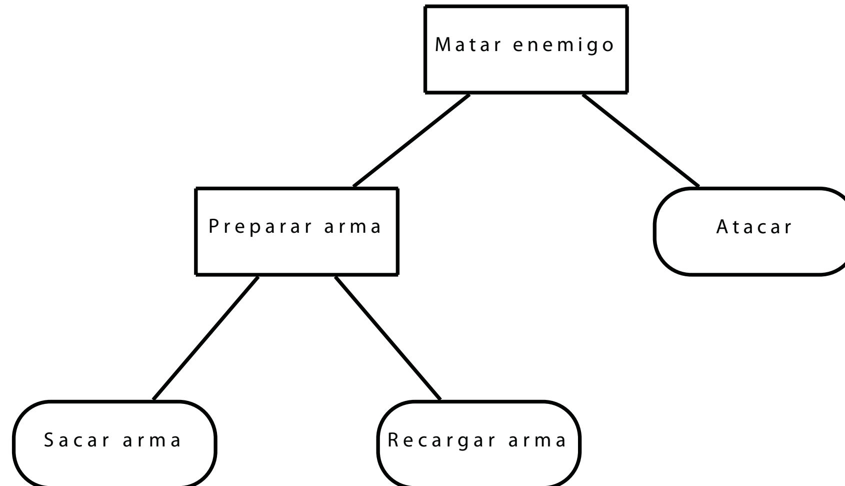


Figura 2.10: Descomposición de tareas

muestra en la figura 2.10, esta tarea se puede descomponer en `Preparar arma` y `Atacar`. `Atacar` es una tarea atómica que el agente puede llevar a cabo, pero `Preparar arma` debe descomponerse a su vez en acciones más simples. La descomposición continúa hasta que todas llegar a un nivel en que todas las tareas son atómicas, por ejemplo, descomponiendo `Preparar arma` en `Sacar arma` y `Recargar arma`.

A continuación analizaremos uno de los formalismos que aplican la planificación jerárquica: las redes jerárquicas de tareas (Hierarchical Task Networks o HTNs).

Una red jerárquica de tareas es una colección de tareas que deseamos que sean realizadas, junto con una serie de restricciones sobre el orden en que pueden llevarse a cabo, la forma en que se instancian sus variables y las condiciones que deben ser ciertas antes o después de que cada tarea se lleve a cabo [16].

Los planificadores clásicos buscan una secuencia de acciones, o plan, para alcanzar un estado que satisfaga determinadas condiciones (los objetivos). El proceso de planificación se desarrolla buscando operadores cuyos efectos hagan cumplir los objetivos y haciendo de las precondiciones de estos operadores nuevos subobjetivos. En el caso de las HTNs, el planificador busca un plan para lograr la red de tareas, que puede incluir otros elementos además de los objetivos. La planificación se realiza mediante descomposición de tareas y resolución de conflictos.

Para obtener un plan, un planificador necesita varios elementos:

- Una red de tareas: un conjunto de tareas que representan acciones que deben ser realizadas. Se pueden distinguir tres tipos de tareas [15]:

  - Tareas primitivas: son tareas que se pueden realizar directamente mediante una acción.
  - Tareas objetivo: son propiedades que deseamos que se hagan ciertas.
  - Tareas compuestas: permiten representar cambios deseados que implican varias tareas objetivo y primitivas.

  La red de tareas también incluye información sobre las restricciones existentes entre ellas.

- Un conjunto de métodos: los métodos asocian cada tarea no primitiva con una red de tareas. De esta manera se indica que, para realizar la tarea no primitiva es necesario realizar las tareas incluidas en la red asociada, respetando sus restricciones. Cada tarea no primitiva puede tener asociados varios métodos, que indican distintas maneras de llevar a cabo la tarea.
- Un conjunto de operadores: los operadores indican cómo se realizan las tareas primitivas. Cada tarea primitiva tendrá asociado un operador. Los operadores incluyen información sobre cuáles son sus efectos en el estado del mundo, pero no tienen precondiciones. Las condiciones de aplicabilidad se determinan mediante los métodos [29].
- Críticos: los críticos se emplean para eliminar los conflictos dentro de un plan tan pronto como sea posible. Por ejemplo, utilizar un recurso al principio de un plan puede suponer que el agente no pueda realizar una tarea posterior que necesita de este recurso. Los críticos intentan detectar este tipo de conflictos lo antes posible. De esta manera se reduce la vuelta atrás y se mejora la eficiencia.

El proceso de planificación sigue el siguiente esquema [16]:

1. Entra en el planificador un problema P.
2. Si P contiene únicamente tareas primitivas, resolver los conflictos en P y devolver el resultado. Si no es posible resolver los conflictos, devolver fallo.
3. Escoger una tarea no primitiva t contenida en P.
4. Escoger una expansión para t (buscar un método m para reducir t).
5. Reemplazar en P la tarea t con su expansión, es decir, la red obtenida de aplicar m.
6. Mediante los críticos, encontrar las interacciones entre las tareas en P y sugerir cómo gestionarlas.
7. Escoger y aplicar uno de los métodos sugeridos por los críticos en el paso anterior.
8. Volver al paso 2.

Cuando un crítico encuentra un conflicto que no se puede resolver, es necesario replanificar. Una de las propiedades más potentes de las HTNs es la replanificación parcial, que supone, en lugar de volver a calcular el plan completo, eliminar la parte del plan culpable de este conflicto y replanificar únicamente esta parte. Gracias a la estructura de árbol de los planes, se puede reemplazar el nodo conflictivo con otra red que realice la misma tarea. Por ejemplo, en el caso de la figura 2.10, si se detectara que el personaje no lleva munición se podría reemplazar el nodo `Preparar arma` por otra tarea como `Preparar torreta`, que se podría descomponer a su vez en `Ir a torreta` y `Activar torreta`.

2.3. Introducción al razonamiento basado en casos

El razonamiento basado en casos, al que nos referiremos en adelante como *CBR* (del inglés, Case-Based Reasoning), es un paradigma de resolución de problemas utilizado para el desarrollo de sistemas basados en el conocimiento. La idea fundamental que se emplea para solucionar un problema consiste en reutilizar soluciones a problemas parecidos ya resueltos en el pasado, adaptándolas a las condiciones del nuevo problema. Para ello se necesita una

colección de experiencias previas, que reciben el nombre de *casos* y se almacenan en una *base de casos*. En términos generales, un caso se compone de la descripción del problema y la solución aplicada para resolverlo [2].

Generalmente, para desarrollar un sistema basado en el conocimiento es necesario tratar con un experto para obtener el conocimiento del sistema. Esto supone:

- Encontrar un experto dispuesto a dedicar tiempo a proporcionar el conocimiento.
- Encontrar una terminología o un lenguaje común entre el experto y el ingeniero del conocimiento.
- Encontrar una manera de formalizar todo el conocimiento obtenido, si esto es posible.
- Encontrar, si es que existen, unos principios aceptados por el conjunto de los expertos, que permita construir el modelo del conocimiento.

Una de las ventajas más notables que aporta el CBR es que sólo necesita un conjunto de problemas resueltos. Para el experto resulta más fácil explicar casos concretos que proporcionar un conjunto de reglas de aplicación general. Esto no quiere decir que se elimine totalmente el problema de la adquisición de conocimiento, aunque sí se reduce. Aún es necesario que el experto proporcione el conocimiento necesario para comparar los casos al buscar problemas parecidos (similitud) y para adaptar una solución al problema actual (adaptación).

Otra ventaja fundamental consiste en que facilita el aprendizaje. En efecto, un sistema CBR puede aprender por simple acumulación de casos, añadiendo los nuevos casos, junto con su solución, a la base de casos. Esto resulta más fácil que generar nuevas reglas para enriquecer el modelo del dominio [12].

2.3.1. El ciclo CBR

De forma general, un problema se puede resolver mediante CBR aplicando el ciclo que se muestra en la figura 2.11. En él se observa que, dado un problema en forma de nuevo caso (*consulta*) se procede a recuperar casos ya resueltos con anterioridad para elegir el mejor de ellos. A continuación se revisa la solución para comprobar que esta es correcta y, opcionalmente, se aprende el nuevo caso, junto con su solución, añadiéndolo a la base de casos.

Representación de los casos

El problema de la representación de los casos consiste en decidir qué debe almacenarse en un caso, cuál es la estructura más adecuada para describir el contenido del caso y cómo debe organizarse e indexarse la base de casos para que la reutilización se realice de forma eficiente.

Por regla general, un caso está compuesto por:

- La descripción del problema.
- La descripción de la solución. Junto con ella, se puede incluir información que facilite el proceso de adaptación. El tipo de información que puede ser útil es, por ejemplo, el proceso seguido para obtener la solución, las alternativas consideradas, cuáles se eligieron y cuáles se descartaron y porqué.
- El resultado de aplicar la solución.

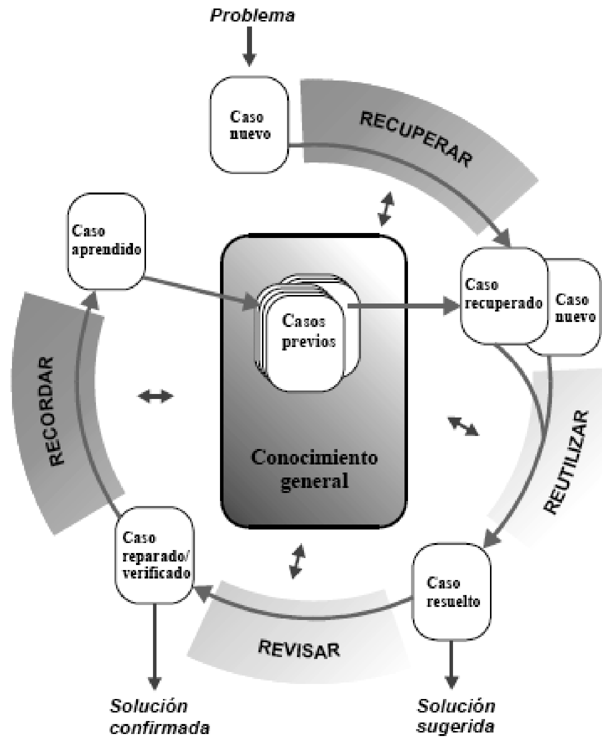


Figura 2.11: El ciclo CBR [12]

En la práctica, no siempre se sigue esta estructura. En ocasiones, por ejemplo, los casos sólo están compuestos por el problema y la solución.

Precio	**Calidad**	**Producto**	**Hojas**	**Cuadriculado**
Medio	Baja	Cuaderno	100	Liso
Alto	Alta	Cuaderno	50	Cuadrícula

Tabla 2.2: Representación plana

Precio	**Calidad**	**Producto**	*Atributos variables*		
			Hojas	**Cuadriculado**	
Medio	Baja	Cuaderno	100	Liso	
Alto	Alta	Cuaderno	50	Cuadrícula	
			Tinta	**Material**	**Duración**
Medio	Media	Bolígrafo	Roja	Metal	Larga
Bajo	Baja	Bolígrafo	Azul	Plástico	Corta

Tabla 2.3: Representación estructurada

Otra consideración a tener en cuenta es cómo estructurar la información que conforma un caso. A grandes rasgos se puede distinguir entre:

- Representación plana: en este caso, todos los casos se representan utilizando los mismos atributos. Se puede ver un ejemplo en la tabla 2.2.

- Representación estructurada: al utilizar esta representación, distintas entidades del dominio pueden tener distintos atributos. En el ejemplo de la tabla 2.3 se observa como los atributos precio, calidad y producto son comunes a todos los casos, pero los restantes dependen del atributo producto.

Por último, es necesario especificar cómo se almacenan los casos en la base de casos.

En el caso más simple se puede emplear una organización lineal. Este tipo de organización está indicado cuando no se prevé que la base de casos vaya a alcanzar un tamaño grande, porque es necesario recorrer toda la base de casos para poder obtener los casos más similares a la consulta. En otro caso, se puede recurrir a una organización estructurada de la base de casos, que permiten un acceso más rápido y eficiente a los casos. A continuación se indican algunas de ellas:

Árboles de decisión A partir del conjunto de casos y fijando una serie de atributos, que actúan como índice, se puede obtener un árbol de decisión mediante el algoritmo ID3 o alguna de sus variantes, de manera que se pueda recuperar en un número mínimo de pasos los casos más relevantes para una consulta dada.

Árboles k-d Son estructuras de datos que provienen del campo de la informática gráfica. Mediante ellos se pueden obtener los k puntos más próximos a uno dado. Los árboles k-d se pueden obtener automáticamente aplicando un algoritmo, que divide el espacio en volúmenes que contienen un número homogéneo de puntos. Aplicados a la recuperación en CBR, se sustituyen puntos por casos, que en lugar de ocupar un lugar en el espacio tridimensional, ocupan un lugar en el espacio de atributos escogidos como índice. El proceso de recuperación consiste en realizar un recorrido en profundidad del árbol, buscando los k vecinos más próximos a la consulta dada.

Recuperación

En la fase de recuperación se obtiene, del conjunto de casos contenidos en la base de casos, un subconjunto con los más similares a la consulta. Es fundamental, por tanto, el concepto de similitud, entendido como la utilidad de la solución al caso previo para resolver el problema actual [12]. En este punto se plantea una aparente contradicción, ya que, por un lado, para saber cuáles son los casos más similares, es necesario comparar la solución de los casos en la base de casos con la solución al caso planteado como consulta, pero para obtener la solución a la consulta es necesario, a su vez, encontrar el caso que guarde mayor similitud con ella.

Para resolver este aparente callejón sin salida se parte de la suposición de que, para dos problemas con descripciones parecidas, existen soluciones parecidas. Por lo tanto, se trata de encontrar un problema similar, ya que se supone que su solución también será similar, y adaptar esta solución al nuevo problema. En la figura 2.12 se muestra un esquema de esta aproximación.

Para el cálculo de la similitud entre dos casos no existe un método estándar que funcione bien siempre, sino que hay diversas aproximaciones que pueden dar mejor o peor resultado dependiendo de la naturaleza de la información que se maneja.

En general se suele distinguir entre medidas de similitud local y global. Las primeras se utilizan para obtener la similitud entre atributos pertenecientes a diferentes casos, mientras que las segundas combinan los resultados de similitud local de todos los atributos para

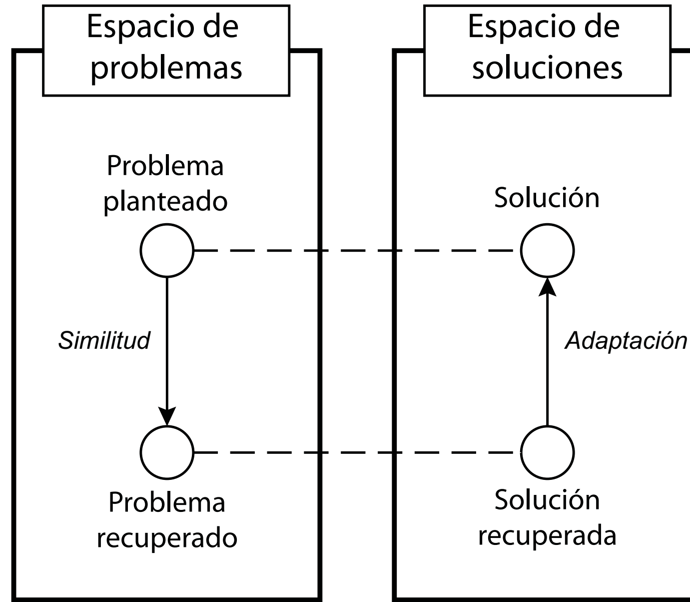


Figura 2.12: Suposición en los sistemas de CBR

obtener un solo valor que mida la similitud entre los casos. A continuación se señalan algunas de las aproximaciones más comunes para ambas medidas:

Similitud local En este caso hay que considerar qué tipo de atributos se están comparando. Atendiendo a este criterio se puede distinguir entre atributos numéricos, símbolos y estructuras más complejas. Para los dos primeros existen algunas funciones estándar que se detallan en la tabla 2.4, distinguiendo además si cada atributo tiene un sólo valor (univaluado) o puede tener varios (multivaluado).

	Univaluado	**Multivaluado**
Simbólico	$sim\,(a,b) = \begin{cases} 0 & \text{si } a \neq b \\ 1 & \text{si } a = b \end{cases}$	$sim\,(a,b) = \frac{card(a \cap b)}{card(a \cup b)}$ $sim\,(a,b) = \frac{card(a \cap b)}{card(O)}$ $sim\,(a,b) = \frac{card(a \cap b)}{\text{mín}(card(a),card(b))}$ $sim\,(a,b) = \frac{card(a \cap b)}{\text{máx}(card(a),card(b))}$
Numérico	$sim\,(a,b) = 1 - \frac{\lvert a-b \rvert}{long(l)}$	$sim\,(a,b) = \frac{long(a \cap b)}{long(a \cup b)}$ $sim\,(a,b) = 1 - \frac{\lvert a_c - b_c \rvert}{long(O)}$ $sim\,(a,b) = \frac{long(a \cap b)}{\text{mín}(long(a),long(b))}$ $sim\,(a,b) = \frac{long(a \cap b)}{\text{máx}(long(a),long(b))}$

O es el conjunto de valores posibles, $long\,(l)$ es el tamaño del intervalo l y a_c y b_c son los puntos centrales de los intervalos a y b.

Tabla 2.4: Funciones de similitud local

En el caso de los atributos simbólicos, si existe una ordenación entre ellos y se puede calcular, tanto su posición en el orden como el número total de valores posibles, se les pueden aplicar las mismas funciones que a los atributos numéricos.

Otras estrategias para el cálculo de la similitud contemplan el uso de tablas de similitudes, en las que se refleja explícitamente el valor de similitud entre cada par de valores de un atributo.

Otra posibilidad es que los atributos simbólicos estén relacionados de una manera jerárquica, en cuyo caso, se puede tener en cuenta esta relación obteniendo funciones de similitud más sofisticadas, como la mostrada en la ecuación siguiente:

$$sim\left(a,b\right) = \frac{prof\left(a \vee b\right)}{\text{máx}\left(prof\left(a\right), prof\left(b\right)\right)}$$

donde la función $prof(n)$ es la profundidad en el árbol jerárquico del nodo n y $a \vee b$ se refiere al concepto de mayor profundidad común a a y b.

Similitud global La similitud global resume los valores de similitud local de cada uno de los atributos del caso en uno sólo, que representa cuál es la similitud entre el caso y la consulta. Para ello puede emplearse cualquier operador de agregación [5]. A continuación se exponen algunos ejemplos, cada uno de los cuales influye de manera diferente en el valor final de similitud global del caso [31]:

- Media geométrica:

  $$\prod_{i \in A} \left(sim_i\left(a_i, b_i\right)^{w_i}\right) \text{ con } \sum_i w_i = 1$$

  donde A es el conjunto de atributos de cada caso y sim es una medida de la similitud local entre atributos.

  Este es un operador de generalización fuerte. Si se aplica este operador para obtener el caso más similar, se obtiene el de mayor similitud local con la consulta en **todos** los parámetros al mismo tiempo.

- Media aritmética:

  $$\sum_{i \in A} \left(w_i \cdot sim_i\left(a_i, b_i\right)\right) \text{ con } \sum_i w_i = 1$$

  donde A es el conjunto de atributos de cada caso y sim es una medida de la similitud local entre atributos.

  El resultado de aplicar este operador es el caso de mayor similitud local en **la mayoría** de los parámetros.

- Media cuadrática:

  $$\sqrt{\sum_{i \in A} \left(w_i \cdot sim_i\left(a_i, b_i\right)^2\right)} \text{ con } \sum_i w_i = 1$$

  donde A es el conjunto de atributos de cada caso y sim es una medida de la similitud local entre atributos. En este caso se trata de un operador elitista. Este operador da mayor valor de similitud global a casos en los que la similitud local es muy alta, aunque sólo sea en **pocos** parámetros.

[5] Los operadores de agregación son objetos matemáticos cuya función es reducir un conjunto de números a un único valor representativo o significativo [13].

Atributo	Consulta	**Caso 1**		**Caso 2**		**Caso 3**	
		Valor	**sim(a, b)**	**Valor**	**sim(a, b)**	**Valor**	**sim(a, b)**
Iniciativa	1	1	1	0	0	1	1
Agresividad	4	2	0,6	3	0,8	4	1
Defensa	2	4	0,6	1	0,8	2	1
Movilidad	5	2	0,4	4	0,8	2	0,4
Comunicatividad	0	2	0,6	1	0,8	4	0,2
Autoestima	5	3	0,6	4	0,8	1	0,2
Media geométrica			**0,611**		0		0,502
Media aritmética			0,633		**0,667**		0,633
Media cuadrática			0,658		0,730		**0,735**

Tabla 2.5: Ejemplos del cálculo de la similitud

Otros operadores comunes son:

- Minkowski:

$$\frac{1}{p}\left[\sum_{i\in A} w_i \cdot sim_i\left(a_i, b_i\right)^r\right]^{1/r} \quad \text{con} \sum_i w_i = 1$$

- Mínimo:

$$\min_{i\in A}\left(w_i \cdot sim_i\left(a_i, b_i\right)\right) \text{ con } \sum_i w_i = 1$$

- Máximo:

$$\max_{i\in A}\left(w_i \cdot sim_i\left(a_i, b_i\right)\right) \text{ con } \sum_i w_i = 1$$

En la tabla 2.5 se muestra un ejemplo de la influencia del operador escogido en el resultado del cálculo de la similitud global. Si se calcula la similitud global aplicando la media geométrica, el caso más parecido a la consulta es el caso 1, con un valor de similitud de 0,611, frente a 0 y a 0,502 de los restantes dos casos. Si se aplica la media aritmética el resultado es el caso número 2, mientras que si se aplica la media cuadrática es el caso 3.

Adaptación

Dentro del ciclo de CBR presentado al principio de esta sección, el proceso de adaptación es el menos estudiado y el menos estandarizado. En muchos sistemas esta fase se obvia o se deja como responsabilidad al usuario, ofreciéndole, en el mejor de los casos, cierta asistencia para llevarla a cabo.

Se pueden distinguir dos tipos de estrategias generales para realizar la adaptación [2]:

- Adaptación transformacional: consiste en reutilizar la solución al caso recuperado aplicando determinados operadores de transformación sobre ella. No trata cómo se resuelve el problema, sino la equivalencia de soluciones. Requiere un fuerte modelo de dominio, los operadores de transformación, y un mecanismo de control para gestionar su aplicación. Por ejemplo, buscamos un cuento en el que exista una princesa llamada Rosamunda que vive en un castillo cerca de un lago. Si la solución de mayor similitud es un cuento en el que aparece una princesa llamada Aglaya que vive en un castillo cerca de un río y le ocurren determinadas cosas, un proceso de adaptación, muy simplista

por otra parte, consistiría en reemplazar las apariciones de Aglaya por Rosamunda, y las referencias al río por un lago.

- Adaptación derivacional: en este caso, lo que se reutiliza es el método mediante el cual se construyó la solución al caso recuperado, y se aplica a la consulta. El caso recuperado debe, por tanto, almacenar cierta información sobre cómo se ha resuelto el problema. Esta información puede incluir detalles tales como una justificación de los operadores empleados, los subobjetivos considerados, las distintas alternativas generadas o los caminos de búsqueda en el espacio de soluciones no satisfactorios. La adaptación derivacional repite los pasos de aplicación del plan de solución recuperado, pero en el contexto del caso introducido como consulta.

Revisión

Durante la fase de revisión se evalúa la solución generada en la fase anterior y, si no es adecuada, se corrige, bien sea utilizando conocimiento específico del dominio o mediante intervención del usuario [2].

La evaluación generalmente supone la aplicación de la solución propuesta al problema real o a una simulación de este. Si esta no es satisfactoria, se pueden detectar los errores y tratar de generar explicaciones para ellos. Esto implica, en la mayoría de los casos, un conocimiento de soluciones erróneas junto con su explicación.

Aprendizaje

Uno de los puntos fuertes del CBR es el aprendizaje, ya que, a un nivel elemental, es muy fácil de llevar a cabo. Se basa, tan sólo, en añadir los nuevos casos consultados a la base de casos.

La finalidad del aprendizaje consiste en mejorar el rendimiento del sistema. Existen dos factores principales que miden el rendimiento: la competencia, que es el rango de problemas que pueden resolverse satisfactoriamente, y la eficiencia, el coste computacional de resolver cada problema.

Atendiendo únicamente a la competencia, esta mejorará según crezca la base de casos dado que:

- Se da una mayor cobertura al espacio de resolución de problemas.
- Es más probable que se recupere un caso más “adecuado” a la consulta.
- Se maximiza la calidad de la solución generada.

Los datos que se suelen conservar de un caso son:

- La descripción del problema.
- La solución.
- El resultado, es decir, si la solución es adecuada al problema, como se mencionó en la fase de revisión.
- Información sobre como se ha obtenido la solución, para poder realizar la adaptación de los casos.

Este tipo de aprendizaje se denomina aprendizaje de casos, frente a otros tipos de aprendizaje como el aprendizaje de conocimiento de recuperación, con el cual se pretende realizar una recuperación más exacta, por ejemplo, mediante el ajuste de los pesos de los distintos parámetros de la consulta o el uso de taxonomías, o el aprendizaje de conocimiento de adaptación, que busca mejorar el proceso de adaptación.

Según lo visto hasta ahora, parece que lo mejor es tener una base de casos lo más grande posible, para mejorar así la competencia. Esto no siempre es así. Si la base de casos crece mucho, las búsquedas en ella son cada vez más lentas, es decir, aparece una degradación de la eficiencia. Es lo que se conoce como el problema de la utilidad [44].

Para resolverlo hay que buscar un compromiso entre la eficiencia y el tamaño de la base de casos, especialmente en sistemas que necesitan bases de casos de gran tamaño. Por ello, es conveniente evitar incluir casos que no aporten información nueva al sistema.

Existen distintas políticas de mantenimiento cuyo objetivo es refinar la base de casos para reducir el problema de la utilidad sin disminuir la competencia del sistema.

2.3.2. CBR textual

De un tiempo a esta parte, uno de los objetivos de la investigación en el ámbito del razonamiento basado en casos ha sido la manipulación de documentos de texto, una tarea relacionada tradicionalmente con la comunidad de Recuperación de Información[6]. Esto se debe a que el CBR se basa en la reutilización de experiencias previas en la solución de un grupo de problemas y, en muchos casos, estas experiencias se almacenan bajo la forma de documentos de texto [34].

El CBR textual es una subdisciplina dentro del CBR en la que los casos están formados, en su totalidad o en parte, por texto no estructurado. Sinopsis de películas, informes médicos, noticias en periódicos o resúmenes de libros son ejemplos de casos en hipotéticos sistemas de CBR textual. En CBR tradicional, antes de procesar cualquiera de estos ejemplos, debería transformarse en una representación estructurada. Este paso no es necesario en CBR textual, ya que los casos están formados por texto plano. Aún así, es habitual realizar un procesamiento previo de los documentos, como se muestra en la figura 2.13.

Para manejar los distintos tipos de conocimiento que pueden aparecer en los documentos, resulta útil separarlo en diferentes capas (*knowledge layers*), cada una de las cuales es la responsable de su gestión [33]:

1. *Keyword layer*: en esta capa se separan los textos en unidades atómicas. Las tareas típicas que se realizan en esta capa son eliminar las palabras vacías (palabras que no aportan significado, como los artículos), extraer las raíces de las palabras, calcular estadísticas acerca de la frecuencia de los términos o realizar un análisis léxico. Esta capa es independiente del dominio.

2. *Phrase layer*: se utiliza un diccionario para reconocer frases dependientes del dominio. El problema que presenta esta capa es que las partes que forman las frases pueden aparecer separadas. Además, el diccionario depende en gran medida del dominio y, en general, debe construirse manualmente.

3. *Thesaurus layer*: contiene información sobre la relación entre las palabras del texto en términos de similitud lingüística. Mediante esta capa se identifican sinónimos y otras relaciones entre términos.

[6]La Recuperación de Información es una disciplina que trata la representación, el almacenamiento, la organización y el acceso a elementos o unidades de información [4].

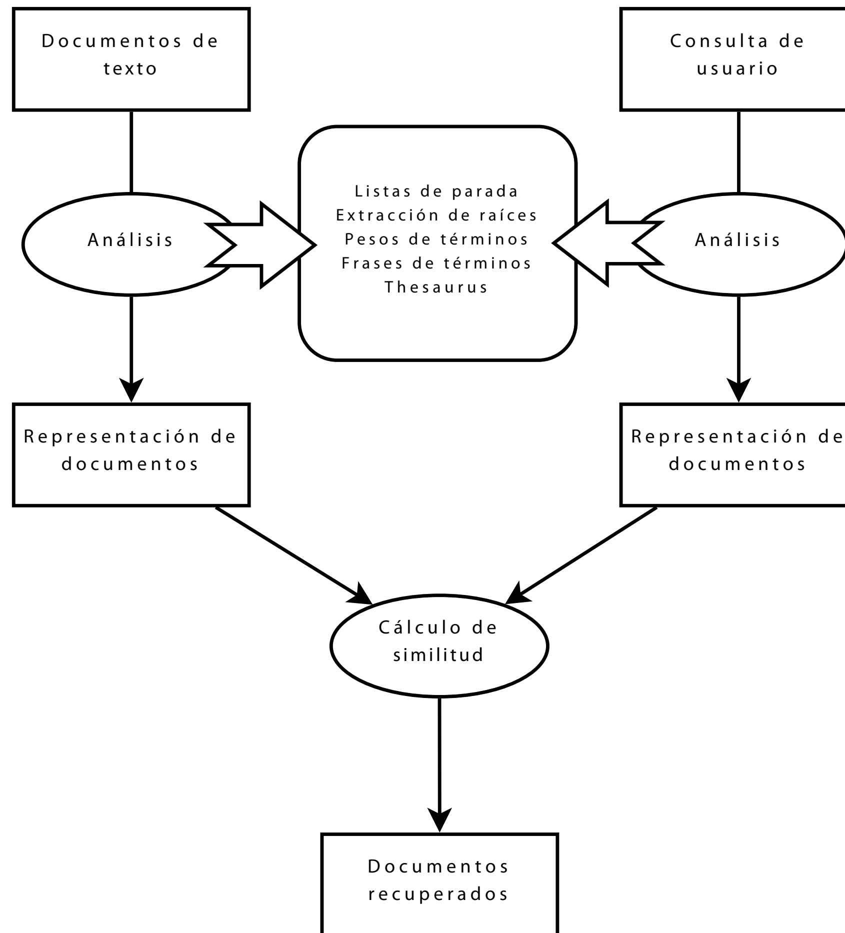


Figura 2.13: CBR textual

4. *Glossary layer*: de la misma forma que la anterior, esta capa relaciona términos entre sí según su similitud. La diferencia consiste en que, en este caso, se tratan relaciones específicas del dominio, y, por tanto, debe emplearse un tesauro específico al dominio para el que se desarrolla el sistema.

5. *Feature value layer*: en casos semiestructurados, formados por texto y atributos, en esta capa se extraen los valores de los atributos. Estos valores suelen encontrarse separados de otras partes del documento.

6. *Domain structure layer*: contiene una descripción de la estructura del dominio y permite, junto con la información extraída anteriormente, una clasificación de los documentos a alto nivel. Mediante esta clasificación se pueden ignorar documentos que no sean relevantes para una búsqueda determinada, mejorando de esta manera la precisión del sistema.

7. *Information extraction layer*: aunque en muchos casos los documentos contienen una sección específica de la que pueden obtenerse los parámetros, las partes de texto de los documentos pueden contener expresiones para las que es más adecuada una repre-

sentación en forma de pares atributo-valor. En esta capa se extrae automáticamente de la descripción textual información estructurada, valores de atributos, etc.

En general, cada capa se asienta sobre la anterior, utilizando sus resultados. Un sistema de CBR textual no debe implementar todas las capas obligatoriamente. Algunas se pueden obviar porque no son necesarias y otras, las que no son específicas del dominio, se pueden reutilizar de otros sistemas ya desarrollados.

El modelo del espacio vectorial

Uno de los métodos más empleados para calcular la similitud entre dos documentos es el modelo del espacio vectorial [36, 4]. Este modelo se basa en la representación de los documentos como vectores en un espacio vectorial común.

Sea t el número total de términos existentes en el sistema, el vector asociado a un documento d_j será:

$$\vec{d_j} = (w_{1,j}, w_{2,j}, \ldots, w_{t,j})$$

donde $w_{i,j}$ es el *peso* del término k_i en el documento d_j. El peso de un término en un documento es un valor no negativo que suele estar relacionado con la frecuencia de aparición del término en el documento. Existen diferentes aproximaciones para obtener un valor para el peso. Más adelante se tratan algunas de ellas.

De la misma forma, se puede asociar un vector de pesos a una consulta c, resultando un vector:

$$\vec{c} = (w_{1,c}, w_{2,c}, \ldots, w_{t,c})$$

Una vez que se han obtenido los vectores de pesos, es necesario compararlos para obtener un valor de similitud. A primera vista, parece que el método más adecuado es obtener la similitud en función de la diferencia entre los vectores de los documentos (o de un documento y la consulta), pero existe el inconveniente de que, documentos que poseen los mismos términos pero con pesos diferentes tendrán valores de similitud bajos [36]. Por este motivo, en lugar de emplearse la diferencia, la similitud de los vectores se calcula en función del coseno de sus ángulos, que se puede obtener aplicando la siguiente fórmula:

$$sim\,(d_i, d_j) = \frac{\vec{d_i} \cdot \vec{d_j}}{\left|\vec{d_i}\right| \left|\vec{d_j}\right|}$$

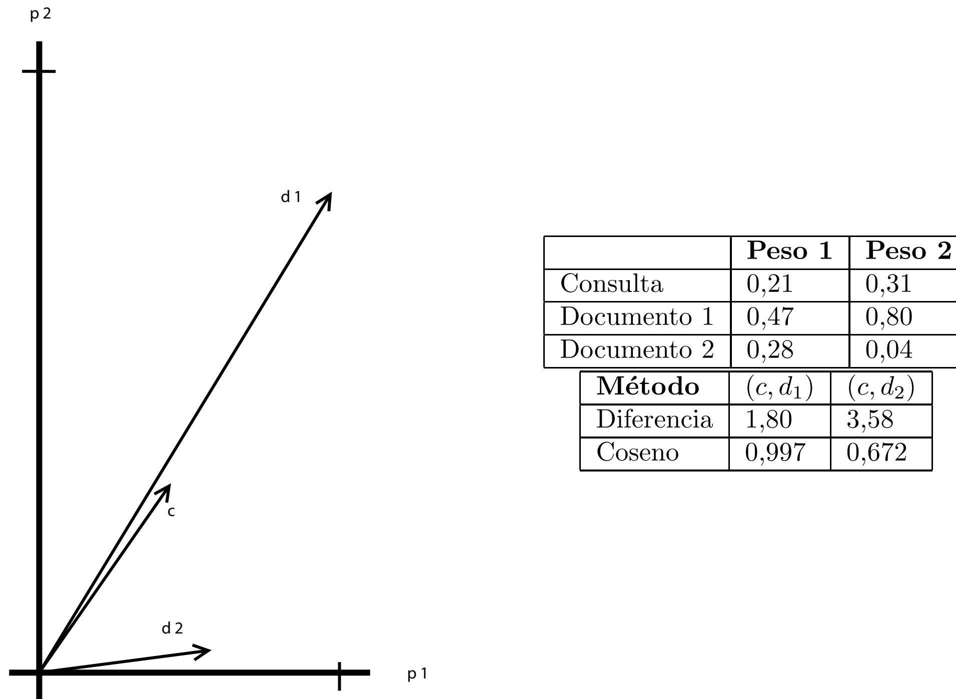
En la figura 2.14 se muestra un ejemplo simplificado de esta aproximación. En ella aparecen representados dos documentos, d_1 y d_2, cuyos vectores son, respectivamente, $\vec{d_1} = (0{,}47, 0{,}80)$ y $\vec{d_2} = (0{,}28, 0{,}04)$, y una consulta $\vec{c} = (0{,}21, 0{,}31)$. En la tabla adjunta se muestran los resultados del cálculo de la similitud utilizando, en primer lugar, la inversa del módulo de la diferencia:

$$sim_{dif}\left(\vec{d}, \vec{c}\right) = \frac{1}{\left|\vec{d} - \vec{c}\right|}$$

y, a continuación, el coseno:

$$sim_{cos}\left(\vec{d}, \vec{c}\right) = \frac{\vec{d} \cdot \vec{c}}{\left|\vec{d}\right| \left|\vec{c}\right|}$$

Se observa que la similitud basada en el coseno del ángulo que forman otorga un valor mayor de semejanza a d_1, mientras que la que utiliza la diferencia lo hace con d_2.



	Peso 1	**Peso 2**
Consulta	0,21	0,31
Documento 1	0,47	0,80
Documento 2	0,28	0,04

Método	(c, d_1)	(c, d_2)
Diferencia	1,80	3,58
Coseno	0,997	0,672

Figura 2.14: Representación vectorial de documentos y consultas

Otros modelos utilizados habitualmente en Recuperación de Información son el modelo booleano, más simple e intuitivo que el del espacio vectorial, se basa en expresiones booleanas para la realización de consultas, y el modelo probabilístico o de independencia binaria, que se basa en la estimación de la probabilidad de que cada uno de los términos aparezca en un documento relevante [36].

Cálculo de pesos de los términos en un documento

- Puntuación por zonas y pesos (*weighted zone scoring*): los documentos se dividen en zonas y se asignan pesos a cada una de ellas. El peso de un término en el documento es la suma de los pesos de las zonas en las que aparece.

- Frecuencia del término ($frec\,(t_i, d_j)$): el peso de un término t_i en un documento d_j es igual al número de ocurrencias del término en el documento. El problema de este método es que se considera que todos los términos son igualmente relevantes.

- Frecuencia relativa: es la normalización del valor anterior con respecto a la máxima frecuencia que puede tener un término en el documento:

$$f\,(t_i, d_j) = \frac{frec\,(t_i, d_j)}{max_l\,\{frec\,(t_l, d_j)\}}$$

  De esta forma se pretende evitar que documentos más largos tengan mayores frecuencias por que repitan las mismas palabras. Por ejemplo, dado un documento d, si se construye un documento d' que es igual a la concatenación de d con sigo mismo, los pesos para d y d' deberían de ser los mismos.

- Frecuencia del término-inversa de la frecuencia del documento: esta aproximación viene dada por la siguiente fórmula:

$$peso\,(t_i, d_j) = f\,(t_i, d_j) \cdot \log \frac{D}{D\,(t_i)}$$

  donde D es el número total de documentos y $D\,(t_i)$ es el número de documentos que contienen el término t_i, de manera que, si $\frac{D(t_i)}{D}$ es la proporción de documentos que contienen el término t_i, en este caso se está multiplicando por su inversa. La idea es que los términos que se repiten en la mayoría de los documentos no son muy útiles para distinguir si un documento es relevante, por lo que se les debe asignar un peso más bajo que a los términos que aparecen en unos pocos documentos. De esta manera, un término que aparece pocas veces en el documento en cuestión (su frecuencia es baja) o que aparece en la mayoría de los documentos (su frecuencia de documento es alta, por lo que la inversa es baja) tendrá un peso bajo, mientras que si aparece muchas veces (su frecuencia es alta) y, además, aparece en pocos documentos (su frecuencia de documento es baja) tendrá un peso más alto.

- Proximidad de los términos de la consulta (*query-term proximity*): en este caso se tiene en cuenta la distancia mínima dentro del documento entre los términos de la consulta, medida en palabras. Cuanto más pequeña sea esta mayor relevancia tiene el documento.

2.4. Aprendizaje y juegos

Aprender desde una posición pasiva es, en general, poco efectivo. Suele ser más beneficioso utilizar ejercicios prácticos como método de aprendizaje, en especial en áreas en las que el aprendizaje pasa por adquirir destrezas, habilidades o creatividad.

Learning-by-doing o "aprender haciendo" es un método de aprendizaje que se basa en la experiencia y que intenta solventar este problema utilizando ejercicios como método de aprendizaje [25]. Para que esta aproximación sea efectiva, la elección de los ejercicios debe seguir un orden lógico de dificultad creciente, que no sobrepase las capacidades del alumno pero que tampoco le resulte tan sencillo que acabe por ser aburrido.

El ciclo que siguen los sistemas basados en learning-by-doing consta de cinco pasos:

1. Selección de los conceptos a practicar: en esta fase se eligen los conceptos que el alumno debe practicar en la iteración actual en función de los conocimientos del alumno.

2. Selección del ejercicio: a partir de los conceptos seleccionados anteriormente se proporciona un ejercicio al estudiante.

3. Resolución del ejercicio: en esta fase el alumno resuelve el ejercicio propuesto en la anterior.

4. Comprobación de la solución: se analiza el trabajo realizado por el alumno.

5. Feedback: en función de los resultados obtenidos en la fase anterior, se explica al alumno qué errores ha cometido o se le indica qué conceptos debe reforzar.

Estas fases no tienen porqué estar separadas, sino que, en muchas ocasiones, su aplicación se solapa, especialmente la de las tres últimas, donde, al mismo tiempo que el estudiante resuelve el ejercicio, se pueden analizar los pasos que está dando para obtener la solución

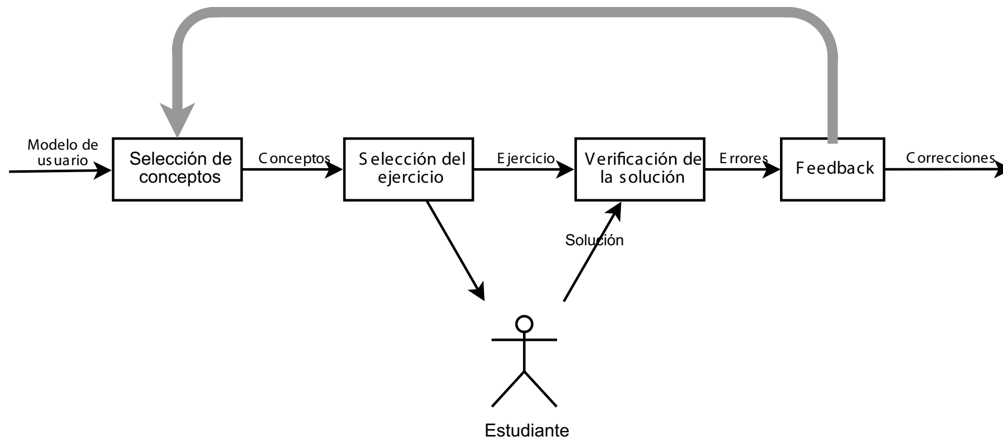


Figura 2.15: Ciclo learning-by-doing

y ofrecerle algún consejo u orientación sobre su actuación. Este es el caso, por ejemplo, de JV^{2}M, un juego para la enseñanza de la máquina virtual de Java al que se ha aplicado el editor de comportamientos desarrollado, y que se tratará en mayor profundidad en la sección siguiente.

En los sistemas de enseñanza inteligentes se manejan 3 tipos de conocimiento [25]:

- Conocimiento del dominio: información relativa al área de conocimiento que se está enseñando. Dentro de este se pueden distinguir cuatro aspectos:
  - Conocimiento general del dominio: formado por un conjunto de conceptos del dominio, se utilizan en la primera fase para indicar qué conceptos hay que poner en práctica en cada iteración. También pueden servir en determinados sistemas para marcar qué partes del dominio domina el alumno y cuáles no. Pueden incluir información sobre los prerrequisitos que se consideran necesarios para un concepto del dominio o su nivel de dificultad. Esta información puede resultar de gran utilidad a la hora de tomar decisiones pedagógicas.
  - Ejercicios: se utilizan en la segunda fase del ciclo. Se seleccionan en función de los conceptos a practicar escogidos en la primera. Pueden estar creados de antemano, o ser construidos bajo demanda.
  - Soluciones: se emplean durante la cuarta etapa, comparándolas con la solución proporcionada por el usuario. Al igual que los ejercicios, también pueden estar creadas de antemano o construirse bajo demanda.
  - Explicaciones: se utilizan durante la quinta etapa para informar al usuario qué ha hecho mal o qué puede mejorar. De la misma forma que en los dos casos anteriores, pueden haberse creado previamente o construirse automáticamente.
- Conocimiento sobre el usuario o modelo de usuario: contiene toda la información sobre el alumno. Suele incluir información sobre qué partes del conocimiento del dominio ya posee, cuáles no y cuáles ha aprendido de manera incorrecta.
- Conocimiento pedagógico: se utiliza para decidir qué ejercicio debe realizar el alumno en cada momento.

El uso de los juegos para la enseñanza puede encuadrarse dentro del “learning by doing”, ya que por su interactividad, los juegos son actividades que implican “acción” [42].

Un juego puede definirse, de manera cualitativa, mediante 6 propiedades [32]:

- Está basado en un conjunto de reglas prefijadas.
- El resultado es variable y cuantificable.
- Los diferentes resultados potenciales tienen asociados diferentes valores, positivos o negativos.
- El jugador invierte un esfuerzo para poder influenciar el resultado.
- El jugador está comprometido con el resultado del juego, en el sentido de que se sentirá bien si el resultado es positivo y gana, y se sentirá mal si el resultado es negativo y pierde.
- El mismo juego puede jugarse con o sin consecuencias para la vida real. Por ejemplo, se puede jugar a la ruleta en un casino, en donde hay que apostar dinero, o en un ordenador, donde sólo se apuestan puntos.

En juegos en los que participa más de un jugador, la interacción social es otro factor importante a tener en cuenta.

En [37] se apuntan algunos de los beneficios de usar videojuegos en la enseñanza. En primer lugar, los juegos despiertan interés. Motivan a través de la diversión, del desafío que suponen y de la respuesta instantánea y visual dentro de un entorno de juego interactivo. Se mantiene que pueden incorporar hasta 36 principios educativos importantes, e inducen al jugador a probar diferentes formas de pensamiento y aprendizaje.

Los juegos de simulación pueden fomentar la visualización, la experimentación y la creatividad a la hora de encontrar nuevas formas de resolver un problema. Además, son una forma de aprendizaje activo, puesto que los jugadores experimentan el dominio o la situación de la simulación de diferentes maneras.

También facilitan la exploración de las relaciones interpersonales, ya sea a través de comportamientos competitivos o cooperativos. Esto puede propiciar el debate sobre los temas tratados, una vez acabada la experiencia de juego.

Gracias a las nuevas tecnologías los jugadores pueden experimentar diferentes roles en entornos muy próximos a la realidad, y aprender sobre el propio entorno, desarrollando su intuición para desenvolverse en él. El uso cada vez más extendido de dispositivos móviles (teléfonos, PDAs e incluso consolas portátiles) ofrecen nuevas oportunidades para el *blended learning*[7], relacionando, por ejemplo, el aprendizaje en el aula con aprendizaje on-line y actividades fuera del aula, como visitas a museos.

Existen, no obstante, algunas consideraciones a tener en cuenta. Por ejemplo, si los objetivos de aprendizaje no son congruentes con los del juego, este puede distraer al estudiante, concentrado únicamente un ganar el juego. Además, lo que para alguien es un juego, a otra persona le puede resultar un trabajo, perdiendo esta aproximación gran parte de su eficacia.

2.4.1. JV^{2}M

JV^{2}M es un juego interactivo para enseñar el proceso de compilación en lenguajes de alto nivel, esto es, cómo se traduce el código fuente a código objeto. Para conseguir aprender

[7]El blended learning es una modalidad enseñanza que combina la enseñanza presencial con la tecnología no presencial [10].

estas tareas, el estudiante necesita conocer tanto el lenguaje fuente como el lenguaje objeto. En el caso de JV^{2}M, el lenguaje fuente es Java y el lenguaje objeto es el código interpretado por la JVM (Máquina Virtual de Java). Se parte de la suposición de que el alumno conoce el lenguaje Java, pero no necesariamente el código objeto de la JVM, que debe aprenderlo con el sistema. El sistema, por lo tanto, enseña dos dominios simultáneamente: conceptos de alto nivel relacionados con los conceptos de compilación, y conceptos de bajo nivel sobre la estructura de la JVM y sus instrucciones [24].

La aproximación que se emplea en JV^{2}M es "learning-by-doing" y sigue el ciclo expuesto en la sección anterior. En JV^{2}M existe una jerarquía que incluye los conceptos del dominio que enseña. Los conceptos que se deben practicar se escogen de esta jerarquía, paliando un modelo de usuario que da una idea sobre los conocimientos que posee el alumno. Por otro lado, existe una base de ejercicios, indexados por el concepto con el que están relacionados. Durante la segunda fase, se escoge de esta base de ejercicios el que será presentado al usuario. El alumno resuelve, entonces, el ejercicio, dentro del entorno virtual. Al realizarse esta tarea dentro del sistema, este puede analizar cada acción que realiza el alumno y proporcionarle información sobre su ejecución. Se observa que las fases 3, 4 y 5 se realizan de manera conjunta.

Los problemas propuestos son programas Java, escogidos en orden creciente de dificultad. En las primeras interacciones, el usuario recibe el código fuente y el código objeto completo que debe ejecutar. Posteriormente, se le van proponiendo ejercicios más complejos, en los que el código compilado no está completo. El estudiante debe completar el código compilado ejecutando instrucciones de código objeto que ha aprendido en interacciones anteriores.

Para resolver los ejercicios planteados, el estudiante deberá compilarlos. La compilación se realiza llevando a cabo determinadas tareas en un entorno virtual tridimensional que es una metáfora de la JVM. En este entorno, el estudiante puede moverse libremente e interactuar con distintos dispositivos que representan las diferentes estructuras de la JVM, variables, operandos de instrucciones, etc [25].

El conocimiento del dominio, tanto del de alto nivel (los conceptos de compilación) como el de bajo nivel (la estructura de la JVM), está incluido en una jerarquía conceptual, y se utiliza, fundamentalmente, en la primera fase del ciclo. Esta jerarquía está organizada por niveles. En la parte superior se encuentran los conceptos acerca de los procesos de compilación. Bajando por la jerarquía se encuentran los conceptos relativos a las instrucciones de la máquina virtual (bytecodes) y, debajo, las microinstrucciones o microoperaciones, las acciones mínimas que puede realizar la máquina virtual. Estas microinstrucciones se relacionan con los conceptos relativos a la metáfora mediante la que se representa la máquina virtual en el entorno de simulación.

JV^{2}M se encuadra, como juego, dentro del género de los juegos de acción en tercera persona, y, como herramienta pedagógica, dentro del *"edutainment"*, género que incluye las formas de entretenimiento diseñadas para educar y divertir a la vez. Al potenciar los aspectos puramente lúdicos del sistema se intenta que resulte más atractivo a los ojos del público al que está destinado. De esta manera, es más fácil lograr los objetivos didácticos planteados [21].

En el juego, el alumno/jugador controla un avatar[8] que está atrapado en una estación espacial de la que tiene que escapar. Esta estación espacial es la metáfora de la máquina virtual, y en ella se encuentran distintas localizaciones y personajes controlados por el sistema que representan los distintos componentes o aspectos característicos de la JVM. Cada ejercicio se corresponde en el juego con un escenario, en el que se reta al jugador a completar la

[8]En el ámbito de los juegos y de internet, un avatar es la representación gráfica de una persona, ya sea, como en este caso, mediante un modelo tridimensional, o con un icono o imagen bidimensional o con una imagen formada por texto.

Figura 2.16: Captura de pantalla de JV^{2}M

ejecución de un código compilado, correspondiente a un determinado código fuente Java, que se proporciona como referencia. El objetivo del jugador es ejecutar el bytecode correspondiente al código fuente del ejercicio antes que su oponente. Si logra ejecutar un determinado número de instrucciones de una misma categoría podrá, a partir de entonces, ejecutar el resto de instrucciones de la misma categoría simplemente introduciendo sus mnemónicos y operandos.

Por ejemplo, en la figura 2.16 se muestra al avatar del jugador a punto de realizar una operación de multiplicación sobre la pila de operandos situada a su izquierda. Cada uno de los operandos se representa mediante una caja con una letra I. Esta letra indica que, en este caso, los operandos son enteros. Sobreimpresa en la interfaz gráfica, puede verse la interfaz de entrada de instrucciones de la máquina virtual, donde se escriben los bytecodes.

Para conseguir los rasgos dinámicos propios de los juegos de acción, se plantea el uso de tres mecanismos:

- Acción: JV^{2}M incluye características propias de los juegos de acción. Para poder completar un escenario, no sólo será necesario dominar los conceptos de la máquina virtual Java, sino que además habrá que demostrar habilidad y reflejos controlando el avatar del jugador.

- Competición: el jugador deberá ejecutar el ejercicio que le corresponde antes que su oponente, que puede ser otro estudiante o un personaje controlado por la inteligencia artificial del sistema.

- Escritura de bytecode: cuando el sistema considera que el jugador ha asimilado una instrucción de bytecode, este puede escribir directamente el mnemónico de la instrucción y sus operandos. De esta manera se evita que tenga que realizar tareas rutinarias

frecuentemente, como sucedía en anteriores versiones [9]

2.5. Herramientas para la edición de comportamientos

2.5.1. BrainFrame

BrainFrame [18] es una herramienta gráfica de autoría de comportamientos inteligentes orientada al ámbito de los juegos y la simulación. Ha sido desarrollada por la empresa Stottler Henke Associates.

Esta herramienta permite la creación, mediante un editor gráfico, de descripciones de comportamientos que pueden ser posteriormente interpretadas por un motor de ejecución que debe integrarse dentro del juego.

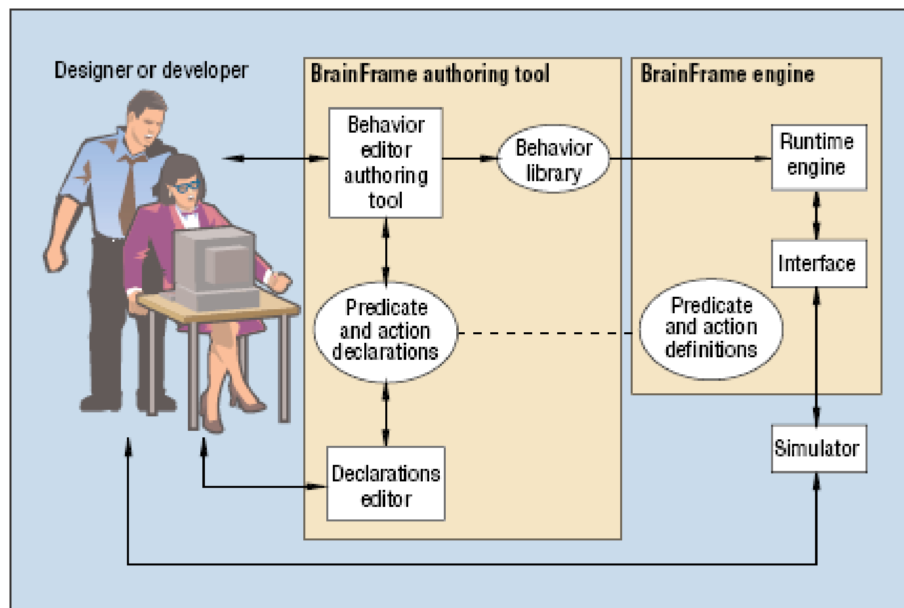


Figura 2.17: Estructura de BrainFrame

El editor gráfico

El editor gráfico está formado por dos componentes, como se muestra en la figura 2.17. En primer lugar, mediante el *editor de declaraciones* se proporciona un vocabulario básico de predicados o condiciones y acciones que serán utilizados como bloques de construcción por el editor de comportamientos [19].

El *editor de comportamientos* utiliza los predicados y las acciones antes mencionadas para construir comportamientos mediante máquinas de estado finito. Las acciones se corresponden con estados de la máquina de estados y los predicados se usan para determinar las transiciones válidas entre estados [20].

Dentro de un comportamiento se pueden referenciar otros comportamientos permitiendo así que estos sean jerárquicos y recursivos.

[9] En las versiones anteriores de JV^{2}M el juego era más similar a una aventura gráfica que a un juego de acción. Puede encontrarse más información en [26].

Los comportamientos se agrupan en una *biblioteca de comportamientos* que será utilizada por el motor de ejecución.

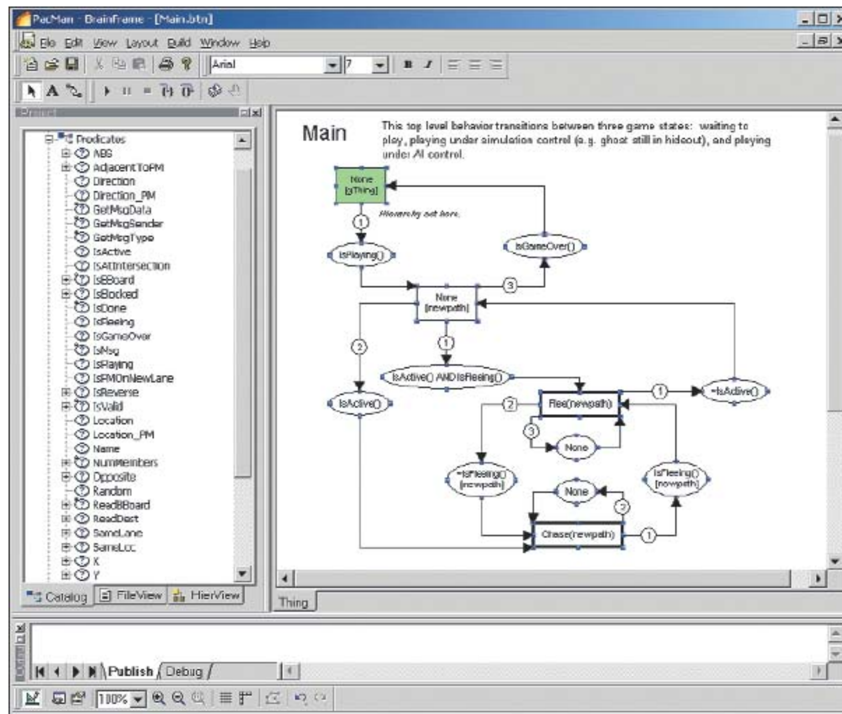


Figura 2.18: El editor gráfico de BrainFrame

El motor de ejecución

El editor gráfico sólo describe el comportamiento, pero no lo implementa. Es el motor de ejecución quien toma esta descripción y la hace operacional en el juego.

El *motor de ejecución* dirige el comportamiento de las entidades en el juego, comunicándose mediante un módulo de interfaz entre el motor y el juego. Un desarrollador debe escribir el código de la interfaz de manera que asigne operaciones en el juego a cada acción y condición.

Para la ejecución de los comportamientos se utiliza una *pila de ejecución*, que mantiene un conjunto de *marcos de ejecución*. Cada entidad en el juego tiene su propia pila de ejecución. Cada marco de ejecución almacena el estado de un comportamiento (nombre, estado actual y valores de las variables). Para la ejecución del comportamiento se crea un marco de ejecución que se introduce en la pila. Cuando se alcanza un estado que referencia a otro comportamiento, se crea un nuevo marco con ese comportamiento y se añade a la cima de la pila.

El motor de inteligencia artificial procesa las pilas en dos pasos. En primer lugar se ejecuta la acción del estado actual del marco que se encuentra en la cima de la pila. En segundo lugar, se determina cuál es la próxima acción. Para ello, se examinan todas las transiciones que salen del nodo actual de cada uno de los marcos, empezando por la parte de abajo de la pila. Cuando se encuentra una transición con una condición que se evalúa a cierto se descartan los marcos que están por encima de ella y se realiza la transición,

añadiendo si es necesario nuevos marcos a la pila. De esta manera, tiene precedencia un comportamiento más “externo” sobre los que están contenidos dentro de él.

2.5.2. Simbionic

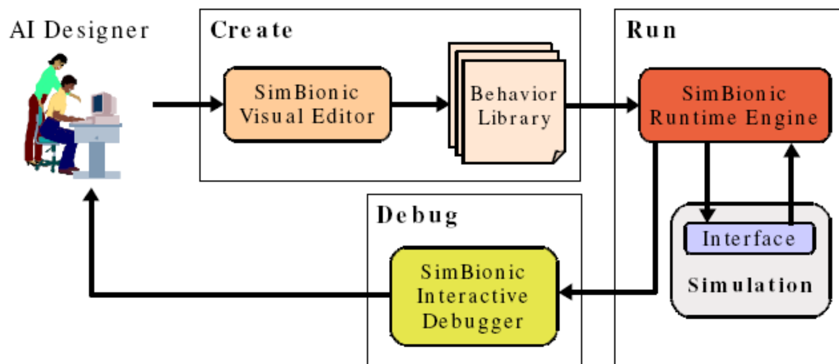


Figura 2.19: Diagrama de bloques de Simbionic

Simbionic [28, 14] surge como evolución de BrainFrame (ver sección 2.5.1), y aporta algunas mejoras sobre ella. A las capacidades ya existentes se añade una implementación del motor de ejecución en java, además de la existente en C++.

Además, se añade un entorno de depuración interactivo que permite encontrar fallos en la lógica de los comportamientos.

También es capaz de generar documentación sobre los comportamientos en HTML.

2.5.3. MindEditor y Replicant Toolkit

MindEditor [3] es un entorno de desarrollo integrado para la edición y simulación de comportamientos con Replicant Toolkit.

El Replicant Toolkit es un API de desarrollo para la simulación de humanos (llamados *replicantes* en el toolkit) y objetos controlados por humanos, tales como coches o camiones. Está formado por dos partes: Replicant Mind, que controla el comportamiento y la inteligencia de los replicantes, y Replicant Body, que está compuesto por una serie de modelos tridimensionales y una biblioteca de animaciones y se usa para visualizar las acciones.

Replicant Mind

El mundo en el que habitan los replicantes es un entorno inteligente. Está poblado por *emisores* (emitters). Los emisores poseen suficiente información como para hacer que la IA entienda lo que son.

Los replicantes son emisores móviles e inteligentes. Su estado está representado por una serie de *atributos mentales* (mind attributes) de los que se definen el estado actual, el estado óptimo, la tolerancia y un peso que representa la importancia del atributo. Los atributos se organizan en grupos jerárquicos denominados *dominios mentales* (mind domains).

Además, cada replicante posee una serie de *generadores de señales* (signal generators). Cada generador de señales recibe la información de los emisores cercanos y, si está capacitado para tratarla, emite una señal específica a todos los procesos del replicante. Una señal sólo es un portador de información acerca del emisor. Cada *proceso* representa un impulso, deseo

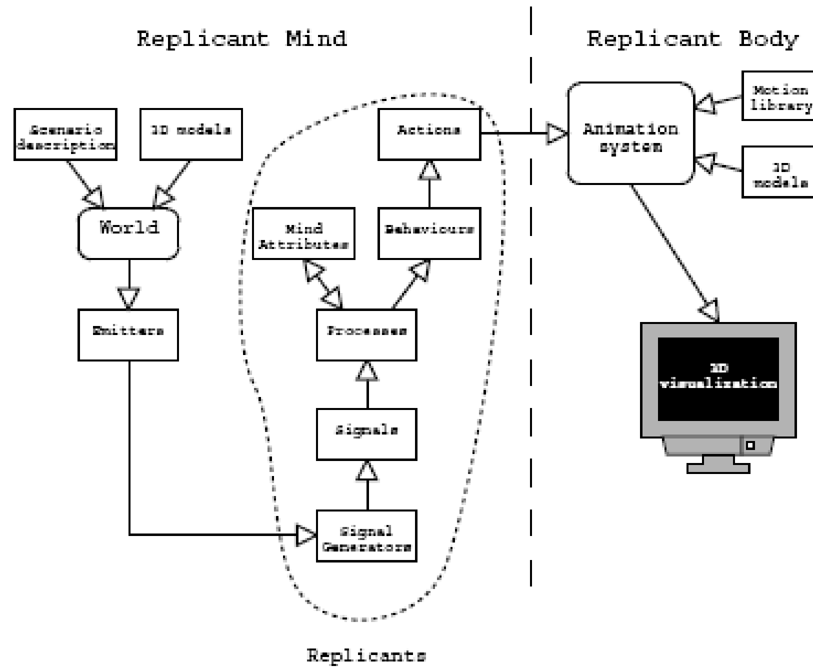


Figura 2.20: Estructura de Replicant Toolkit

o necesidad. Los procesos reaccionan ante las señales modificando los atributos mentales. Si el valor de alguno de estos atributos queda fuera de su rango óptimo el proceso debe tomar medidas para devolverlo a este rango. Esto se hace delegando en los *comportamientos*. Los comportamientos tienen objetivos muy específicos, que intentan lograr en la medida de lo posible. Para alcanzar sus objetivos los comportamientos pueden nominar *acciones*. El sistema de decisión permite que los comportamientos voten por las diferentes acciones nominadas y elige que acción debe realizarse. Las acciones son la salida del sistema de decisión y son suministradas al Replicant Body para que las ejecute.

MindEditor

MindEditor es un entorno de desarrollo integrado para el Replicant Toolkit. Se utiliza para diseñar y probar comportamientos para los replicantes. Tiene un entorno de programación integrado que permite el uso del lenguaje de `scripts` Lua para definir los procesos y los comportamientos. Además, permite visualizar simulaciones de escenarios en tres dimensiones.

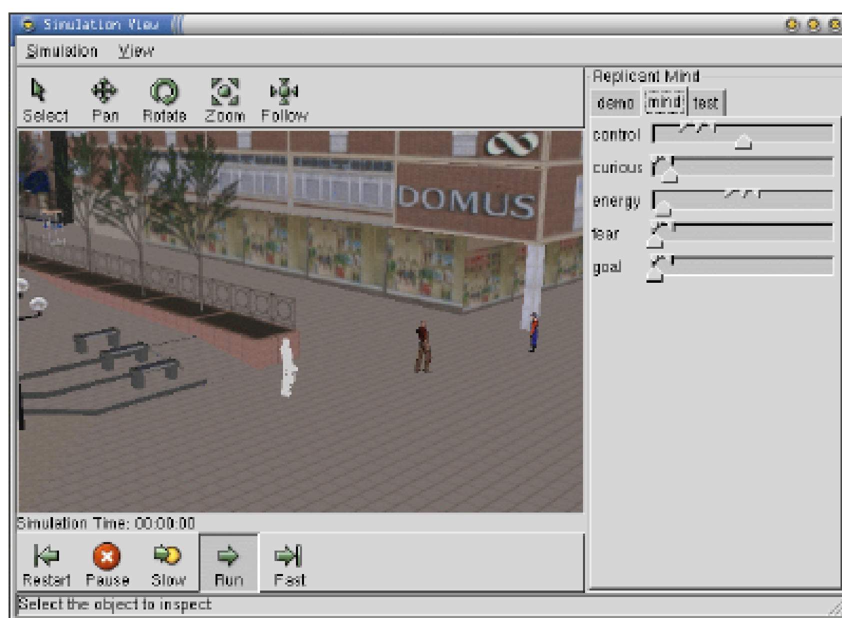


Figura 2.21: Captura de pantalla de MindEditor

Capítulo 3

Edición y generación semiautomática de comportamientos

La parte central del presente trabajo ha consistido en el desarrollo de una herramienta gráfica de edición de comportamientos inteligentes. Esta herramienta permite la creación y edición de comportamientos basados en máquinas de estado finito aplicables a, prácticamente, cualquier juego o entorno de simulación. De hecho, no es necesario restringir su uso únicamente al ámbito de los juegos, sino que se pueden emplear para cualquier tarea que requiera el procesamiento de eventos.

El editor consta de dos partes, íntimamente ligadas entre sí. Por un lado, (1) un editor gráfico de comportamientos, con el que se pueden crear y editar comportamientos basados en máquinas de estado finito. Este editor utiliza (2) un sistema de razonamiento basado en casos para realizar consultas contra una base de casos de comportamientos. Esta base de casos está formada por comportamientos creados previamente por el editor. Por lo tanto, existe una doble dependencia de un sistema sobre el otro. Mediante el editor gráfico se pueden crear nuevos comportamientos desde cero, o utilizando como punto de partida comportamientos recuperados mediante el sistema CBR; y el sistema CBR utiliza en en las consultas una base de casos compuesta de comportamientos editados previamente mediante el editor gráfico.

Como se mencionó en el capítulo 1, los objetivos principales que se han perseguido durante el desarrollo de esta aplicación son tres: sencillez de uso, aplicabilidad y asistencia al usuario.

La sencillez de uso implica que no sea necesario un conocimiento técnico amplio para poder utilizar el editor. Se han escogido máquinas de estado jerárquicas para representar los comportamientos porque son intuitivas y simples de diseñar y editar. No obstante, queda abierto el camino para añadir en futuras versiones otros paradigmas de representación de comportamientos, como los tratados en la sección 2.2.

La aplicabilidad supone poder utilizar el editor para diferentes juegos y entornos de simulación. Para lograrla se utilizan distintos ficheros, llamados modelos de juego, que describen cada juego, y diferentes generadores de código, que generan el código fuente correspondiente al juego y al comportamiento editado.

La asistencia al usuario se logra mediante un sistema de búsqueda de comportamientos basado en CBR.

En las siguientes secciones se tratan en detalle estas características y cómo se integran

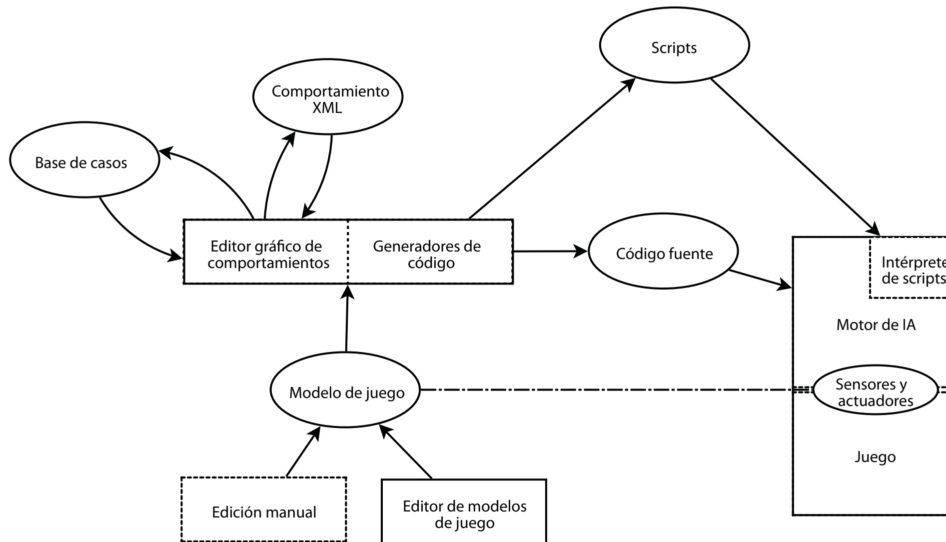


Figura 3.1: Diagrama de bloques del editor de comportamientos

en el editor, tratando en primer lugar el editor gráfico y, a continuación, el sistema CBR.

3.1. El editor gráfico de comportamientos

En el editor gráfico se pueden distinguir tres partes que cumplen tres funciones diferenciadas:

1. Los modelos de juego, que especifican cómo es la interfaz entre el juego y la inteligencia artificial.
2. El editor de comportamientos propiamente dicho, que permite crear nuevos comportamientos y modificar los existentes. Además, ofrece asistencia al usuario en la realización de estas tareas.
3. Los generadores de código, que transforman el comportamiento creado en el editor en código fuente de un lenguaje de programación o en un `script`.

En la figura 3.1 se muestra un esquema de la arquitectura del editor de comportamientos y las relaciones entre cada una de sus partes.

3.1.1. Modelos de juego

Para poder crear los comportamientos es necesario conocer determinados detalles sobre la arquitectura del juego, en concreto, el conjunto de sensores y actuadores (véase la sección 2.1 en la página 13) que están disponibles. Para que el editor sea consciente de estos elementos es necesario indicarle un *modelo de juego*.

Un modelo de juego es un fichero en el que se explicita el conjunto de sensores y de actuadores que ofrece la interfaz del juego. Este fichero debe ser creado por alguien con conocimientos sobre la arquitectura y la estructura del juego, generalmente un programador.

El fichero que define cada modelo de juego es un xml (figura 3.2). A continuación se esboza de manera informal su estructura, que se detallará más adelante, en el apéndice A:

- Modelo de juego: queda descrito por un atributo nombre. Contiene el modelo de la interfaz de juego y una lista de propiedades.
  - Interfaz de juego: la interfaz de juego es el conjunto de sensores y actuadores que permite la comunicación entre la inteligencia artificial y el juego. Se define mediante una lista de sensores y otra de actuadores.
    - Sensores: los sensores quedan definidos por el nombre del sensor, que es el texto que se mostrará al usuario en la interfaz de usuario, el tipo de datos del valor que devuelve el sensor, su categoría y una descripción más detallada de su función.
    - Actuadores: los datos necesarios para cada actuador son el nombre mostrado en la interfaz de usuario, el número de parámetros que recibe, la categoría y una descripción textual detallada.
  - Lista de propiedades: el modelo de juego se completa mediante una lista en la que se declaran una serie de propiedades que se utilizarán para realizar las consultas a la base de casos durante el proceso de búsqueda basada en CBR. Para cada una de estas propiedades se detalla su nombre y una serie de valores, el mínimo, el máximo y un valor por defecto. En la sección 3.2.1 se explica en mayor detalle la función de estas propiedades y cómo participan en las búsquedas.

Tanto en los sensores como en los actuadores se distinguen tres categorías diferentes:

- Naturales: son sensores o actuadores que vienen definidos en la interfaz que ofrece el juego para comunicarse con el módulo de control de comportamientos.
- Sintéticos: se obtienen realizando alguna operación sobre uno o varios sensores o actuadores naturales. Por ejemplo, el sensor `getBallX`, mostrado en el listado 3.1 es el resultado de realizar la operación `abstract_robot.getBall(timestamp).x`.
- Por defecto: esta categoría incluye sensores y actuadores comunes a todos los modelos de juego. El actuador `MACRO`, mediante el que se puede definir un fragmento de código fuente para que sea ejecutado directamente en el comportamiento, es un ejemplo de actuadores por defecto.

Mediante esta categorización se pretende dar una orientación al desarrollador del generador de código sobre qué sensores o actuadores se deben incluir directamente en el código generado (naturales) y cuáles requieren realizar algún tipo de operación adicional (sintéticos y por defecto).

Para poder generar un comportamiento para un determinado juego hay que utilizar el modelo de juego adecuado al juego en cuestión. De esta manera, puede utilizarse el editor para crear comportamientos para distintos juegos con interfaces de juego diferentes.

Por ejemplo, para crear un comportamiento para Soccerbots[1] debe utilizarse el modelo de juego de soccerbots, que incluye las declaraciones de los sensores existentes, tales como, `canKick` o `getBallX` y `getBallY`, y los actuadores permitidos, como `kick` o `setSpeed`. En el listado 3.1 se muestran algunos ejemplos de sensores y actuadores.

3.1.2. El editor de comportamientos

La pieza central del prototipo desarrollado es el editor de comportamientos. Se trata de un editor gráfico que permite la visualización y edición de comportamientos inteligentes,

[1]Soccerbots es un entorno de simulación de partidos de fútbol entre robots. En la sección 4.2 se incluye una descripción más detallada del entorno, así como un ejemplo de su integración con el editor.

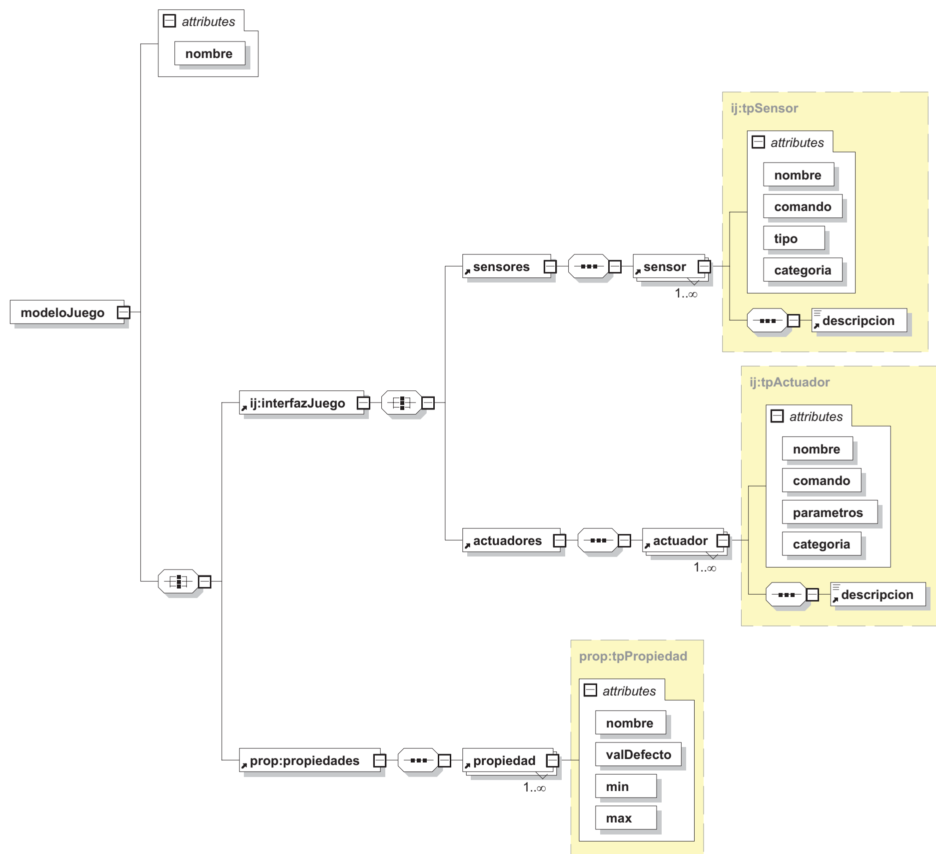


Figura 3.2: Estructura de un modelo de juego

utilizando para ello máquinas de estado finito jerárquicas.

El editor permite crear comportamientos desde cero que utilizan los sensores y actuadores de un modelo de juego creado previamente. También permite cargar comportamientos creados anteriormente para modificarlos o salvar comportamientos en disco.

Para la edición de comportamientos se dispone de dos herramientas básicas: los nodos y los conectores.

Nodos

Los nodos representan los estados de la máquina de estados y, gráficamente, se representan mediante rectángulos con las esquinas redondeadas. El estado inicial de un comportamiento se aparece dibujado con un doble borde.

De la misma manera que en las HFSMs había dos tipos de estado (los estados clásicos y los superestados), existen dos tipos de nodos: los que contienen secuencias de acciones atómicas (invocaciones a actuadores), a los que llamaremos nodos atómicos, y los que contienen otro comportamiento anidado, los nodos compuestos. Los primeros muestran en su interior una lista que contiene las llamadas a los actuadores, mientras que en los segundos se muestra el nombre del comportamiento al que hacen referencia.

```
...
<ij:sensor nombre="canKick" categoria="NATURAL" tipo="BOOLEAN">
  <ij:descripcion>
    Es cierto si el jugador puede golpear la pelota
  </ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getBallX" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>
    Devuelve la coordenada X de la posición de la pelota
  </ij:descripcion>
</ij:sensor>
...
<ij:actuador nombre="Kick" categoria="NATURAL" parametros="0">
  <ij:descripcion>
    Da una patada a la pelota si esta se encuentra a la
       distancia adecuada
  </ij:descripcion>
</ij:actuador>
<ij:actuador nombre="SetSpeed" categoria="NATURAL"
   parametros="1">
  <ij:descripcion>
    Establece la velocidad a la que se mueve el jugador
  </ij:descripcion>
</ij:actuador>
...
```

Listado 3.1: Ejemplo de declaración de sensores y actuadores

Para establecer estos valores se utilizan las páginas de propiedades de los nodos, las cuales están divididas en tres paneles.

1. Tipo de nodo: el panel situado en la parte superior muestra un selector en el que se permite escoger entre el tipo de nodo: atómico, compuesto o sin comportamiento. En el último caso elimina el contenido del nodo.

2. Propiedades del nodo: es el panel situado en la parte central de la página de propiedades. El contenido de este panel cambia dependiendo del tipo de nodo seleccionado:

   - Nodo atómico: este es el caso mostrado en la figura 3.4. En este caso, en la parte superior aparece el nombre que se asigna al comportamiento; debajo se pueden ver dos listas: en la de la derecha se muestran todos los actuadores disponibles en el modelo de juego actual. La de la izquierda contiene las llamadas a los actuadores que forman el comportamiento atómico, junto con los valores asignados a los parámetros del actuador si son necesarios.
   - Nodo compuesto: para los nodos compuestos se muestra únicamente el nombre del comportamiento contenido en el nodo (ver figura 3.5). Para asignar un comportamiento al nodo se muestra un formulario que permite añadir un comportamiento

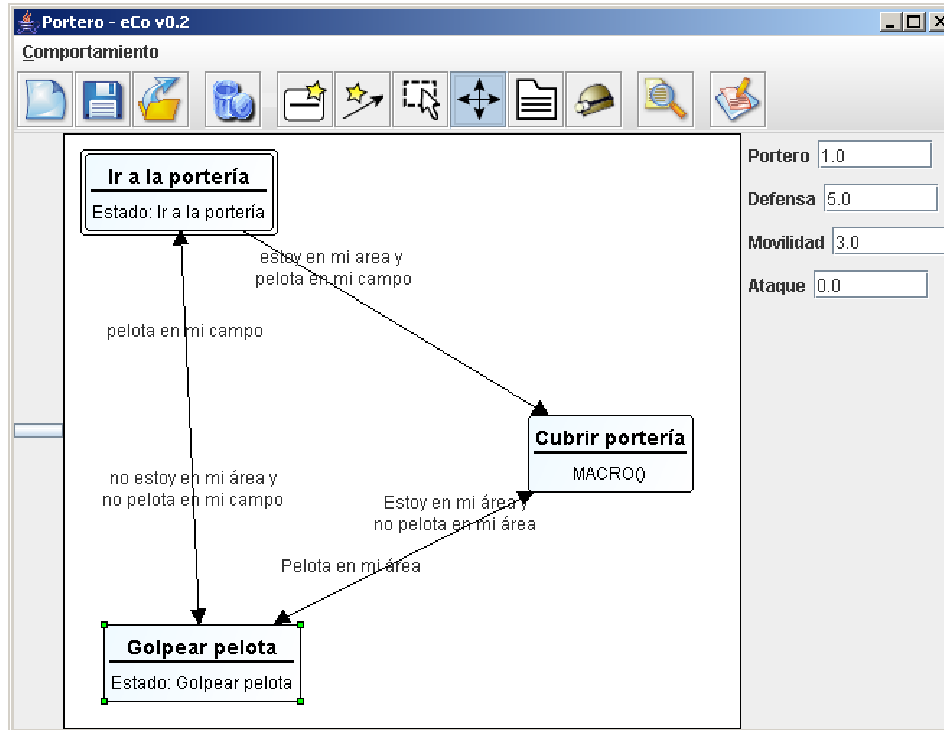


Figura 3.3: Captura de pantalla del editor de comportamientos

nuevo, añadir un comportamiento existente en otro punto de la jerarquía de comportamientos o cargar un comportamiento de disco.

3. Propiedades del comportamiento: mediante estas propiedades se definen las características más importantes del comportamiento. Los valores aquí asignados serán utilizados por el motor de búsqueda de CBR.

   No todas las propiedades son igualmente adecuadas para distintos juegos. Por ejemplo, mientras que en un juego de fútbol como puede ser Soccerbot podría existir una propiedad llamada "portero", que indique la destreza que posee un comportamiento para defender la portería, esta propiedad carecería de sentido en un juego de acción en primera persona (First Person Shooter o FPS). Por ello, para cada modelo de juego, se puede definir un conjunto de propiedades diferente. En las figuras 3.4 y 3.5 se muestra como ejemplo un conjunto de propiedades utilizado con Soccerbots.

Conectores

Los conectores representan las aristas que unen los estados, es decir, las transiciones entre estados. Gráficamente, aparecen en el editor como una flecha que apunta desde el estado de partida al estado destino.

Los conectores pueden tener asociadas fórmulas para controlar cuándo se producen las transiciones. Cada fórmula tiene la forma de una disyunción de conjunciones de condiciones del tipo *sensor <comparador> valor*. Si existe una fórmula asociada al conector, este se mostrará en el editor, dibujado en color negro. En caso contrario, aparecerá en color rojo.

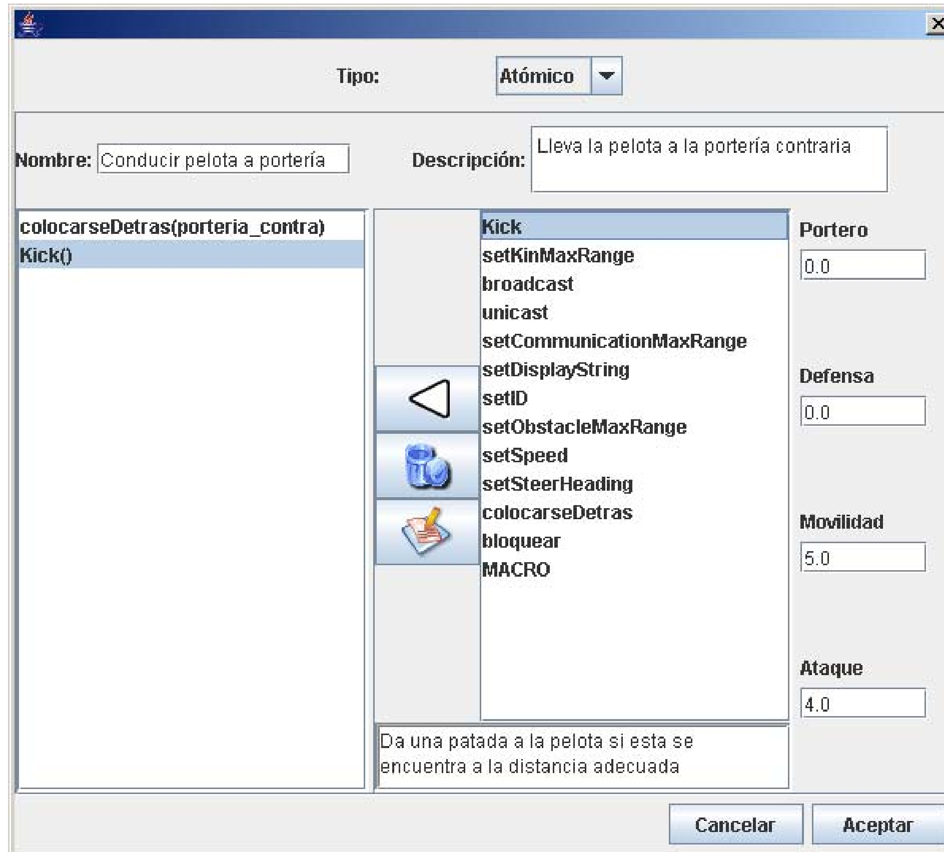


Figura 3.4: Página de propiedades de un nodo atómico

Para asociar las condiciones a los conectores se utiliza la página de propiedades de los conectores, que se muestra en la figura 3.6. Se trata de un formulario dividido en tres paneles. La funcionalidad de los paneles, de derecha a izquierda, es:

1. En el tercer panel, el que se encuentra más a la derecha, se muestra una lista de los sensores declarados en el modelo de juego. Encima de la lista existe un selector que permite especificar el comparador y el valor. Los comparadores dependen del tipo de datos del sensor, como se indica en la tabla 3.1.

2. En el panel central se añaden las condiciones que forman una conjunción.

3. Finalmente, el panel de la izquierda contiene todas las conjunciones que forman la disyunción, es decir, la fórmula asociada al conector.

El editor no obliga a que las fórmulas asociadas a las aristas que salen de un nodo sean mutuamente excluyentes, es decir, que puede haber dos aristas cuyas condiciones se hagan ciertas a la vez. Esto introduce en las máquinas de estado generadas por el editor la noción de no determinismo. Este no determinismo puede ser resuelto de diferentes formas. La responsabilidad de resolverlo recae en el generador de código. En la sección 3.1.3 se trata este tema con más detalle.

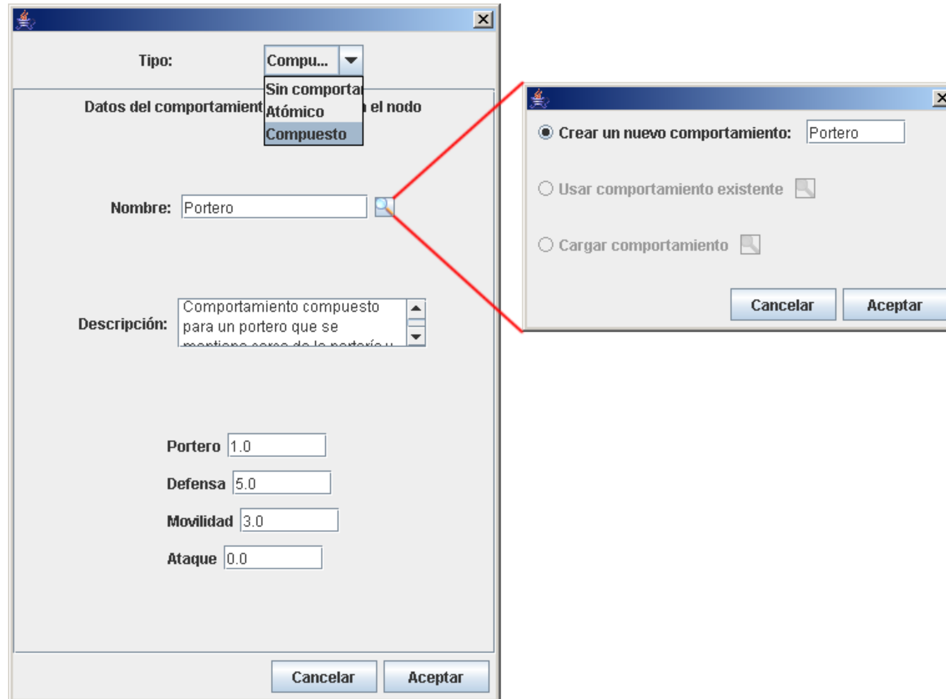


Figura 3.5: Página de propiedades de un nodo compuesto

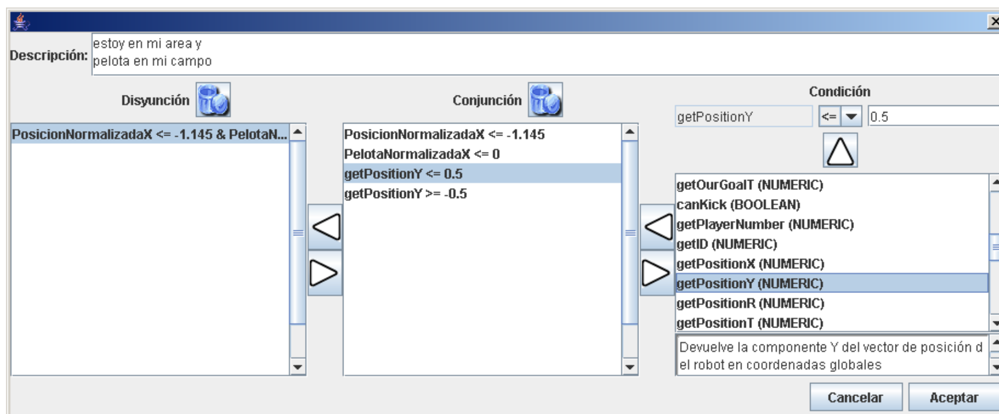


Figura 3.6: Página de propiedades de un conector

Otras funcionalidades

Además de añadir nodos y conectores el editor permite realizar otras acciones para la edición y gestión de los comportamientos:

- Crear un **nuevo** comportamiento usando el mismo modelo de juego u otro diferente.
- **Guardar** el comportamiento actual. Mediante esta orden, el editor almacena el comportamiento actual en disco en formato xml para su posterior edición. Mediante esta

Tipo	Comparadores
BOOLEAN	$=, \neq$
NUMERIC	$=, \neq, >, <, \geq, \leq$
STRING	$=, \neq$

Tabla 3.1: Tipos de datos de los sensores y comparadores asociados

Figura 3.7: Herramientas disponibles en el editor

orden también es posible almacenar cualquier comportamiento de la jerarquía de comportamientos actual en la base de casos.

- **Cargar** un comportamiento de disco. Esta opción carga un comportamiento y el modelo de juego asociado a este.
- **Borrar** estados o conectores.
- Crear nuevos **estados** y **conectores**.
- **Seleccionar** un conjunto de estados y conectores para realizar acciones sobre ellos (moverlos o borrarlos).
- **Mover** un estado o conjunto de estados.
- Mostrar las **propiedades** de un estado o conector.
- **Examinar** un estado compuesto. Mediante esta orden se puede descender en la jerarquía de comportamientos para ver y editar el contenido de un estado compuesto.
- **Buscar** comportamientos en la base de casos. Mediante esta orden se invoca el módulo de CBR para realizar búsquedas de comportamientos. En la sección 3.2 se trata este tema en más profundidad.

3.1.3. Generadores de código

Los generadores de código entran en juego en la última fase del desarrollo de un comportamiento y su función, como su nombre indica, consiste en obtener un fichero de código compilable, interpretable o ejecutable a partir del comportamiento editado.

Cada generador es una clase que hereda de la clase abstracta `Generador`. Cualquier clase derivada de esta debe implementar un método `generar`, que recibe como parámetro un comportamiento compuesto y debe devolver una cadena de texto que contenga la implementación del comportamiento.

Además, por derivar de la clase `Generador`, todo generador de código poseerá una estructura llamada parámetros, en la que se almacenarán los posibles parámetros que pueda necesitar el generador de código para su ejecución. Existen dos métodos que dan acceso a esta estructura: `getParametros` y `setParametros`.

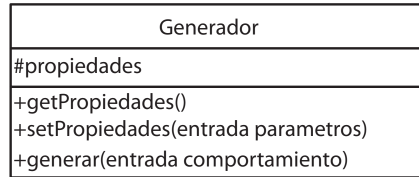


Figura 3.8: Clase abstracta generador

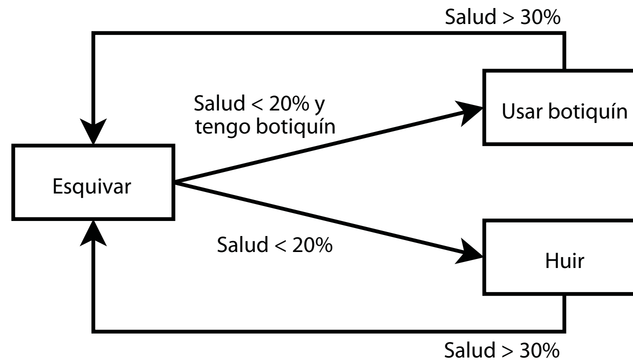


Figura 3.9: Comportamiento con dos transiciones para la misma entrada

Por regla general, un generador de código recorrerá todos los nodos y aristas del comportamiento, traduciendo su contenido, ya sean llamadas a sensores o a actuadores, al lenguaje de destino. Por este motivo, los generadores están muy ligados al modelo de juego y, normalmente, un generador será válido únicamente para un modelo de juego determinado.

Dentro del método `generar` se admite cualquier implementación de máquinas de estado finito. Incluso es posible transformar la máquina de estado a otra representación, como un sistema de reglas, e implementarla.

Resolución del no determinismo

Una característica de la implementación actual del editor es que se puede definir un comportamiento que no sea determinista. En efecto, en un nodo determinado puede haber dos transiciones que se hagan ciertas al mismo tiempo. Es en la implementación realizada por el generador de código donde se decide qué transición se realiza. Por ejemplo, se puede dar un orden de precedencia basado en el orden en que se han definido las aristas, en los sensores asociados a las aristas (determinados sensores tienen precedencia sobre otros) o escoger aleatoriamente una transición de entre todas las posibles. Al editar un comportamiento es importante tener en cuenta cómo se trata el no determinismo en el generador. De lo contrario, la ejecución del comportamiento generado podría no mostrar los resultados deseados en el diseño.

Supongamos que deseamos implementar un comportamiento para un personaje de un juego de acción. Este personaje debe provocar al jugador y esquivarle, evitando ser atrapado. Cuando la salud del personaje es muy baja, este tiene dos opciones: si tiene un botiquín puede utilizarlo para curarse; si no, debe huir y esperar a curarse de manera natural. Cuando el personaje recupera la salud, vuelve a provocar al jugador. Este comportamiento podría quedar especificado mediante la máquina de estado de la figura 3.9.

Dependiendo de la resolución del no determinismo en la implementación del generador de código, este mismo comportamiento puede dar un resultado u otro. Por ejemplo, si se evalúan las condiciones en el orden en que se han declarado, y se declara primero la arista que va de `Esquivar` a `Usar botiquín`, el personaje se curaría siempre que tuviera un botiquín en su poder; si se declarara antes la otra arista, el personaje nunca usaría su botiquín, ya que primero se comprueba si $salud > 20\,\%$ y, si no es así, se comprobaría si $salud > 20\,\%$ `y tengo botiquín`.

Otra posibilidad sería evaluar todas las condiciones que se verifican y elegir una de ellas aleatoriamente. En ese caso, el comportamiento representado en la figura huiría del jugador si no tiene en su poder un botiquín y, en caso contrario, podría huir o utilizar el botiquín indistintamente.

3.2. El sistema de razonamiento basado en casos

Como se apuntó en el capítulo 1, una característica común a muchos de los comportamientos de los personajes dentro de los juegos es la modularidad. El usuario del editor puede aprovechar esta característica a su favor utilizando comportamientos creados previamente para modificarlos o para incluirlos dentro de comportamientos más complejos. Para ello, tendría que buscarlos dentro de su disco y abrirlos o importarlos[2] en la aplicación. Existen diversos escenarios en los que esto puede ser, cuando menos, una tarea bastante tediosa. Por ejemplo, en el caso en que los comportamientos hayan sido diseñados por otra persona, el usuario debería abrir uno por uno todos ellos, y analizarlos para ver si se adecúan a sus necesidades, hasta dar con el más apropiado.

Mediante el sistema CBR se pretende dar asistencia al usuario en la realización de estas tareas, permitiendo realizar consultas a una base de casos, en la que cada uno de los casos representa un comportamiento creado previamente. Mediante estas consultas se puede realizar una recuperación aproximada de comportamientos editados con anterioridad con características parecidas a las propuestas en la consulta. Los comportamientos recuperados pueden ser editados para crear un nuevo comportamiento diferente, o incluidos dentro de un comportamiento más abstracto.

Se permiten dos tipos de consultas contra la base de casos: consultas por funcionalidad y consultas por estructura. En las primeras se aportan determinados parámetros que indican la funcionalidad deseada del comportamiento, mientras que las segundas tratan de recuperar comportamientos cuya composición de nodos y aristas sea similar a la indicada.

3.2.1. Consultas por funcionalidad

Mediante este tipo de consultas se busca un comportamiento cuyas características se aproximen a unos parámetros introducidos por el usuario, que expresan las funcionalidades deseadas en el comportamiento. Los parámetros que conforman la consulta se utilizan para describir el comportamiento, y están muy relacionados con el modelo de juego. Cuanto más diferentes entre sí sean dos juegos, mayores diferencias habrá entre los comportamientos y, por lo tanto, entre los atributos utilizados para describirlos.

Los comportamientos así recuperados pueden, por ejemplo, ser modificados en el editor para obtener un nuevo comportamiento, o ser añadidos a un comportamiento de más alto nivel.

[2] Distinguimos entre abrir un comportamiento, cuando se carga directamente en la aplicación, e importar, cuando el comportamiento se añade al que está siendo editado.

A continuación se analizan cada una de las fases del proceso de CBR, comenzando por la representación de los casos, la recuperación, la adaptación y el aprendizaje. Por último, se incluye una breve exposición sobre cómo se integran este tipo de consultas en el editor.

Representación de los casos

Los casos se representan usando varios parámetros que se pueden agrupar en tres categorías: atributos, descripción y comportamientos incluidos.

Atributos

Se trata de un conjunto de atributos numéricos que describen distintas propiedades del comportamiento.

Es evidente que no se pueden caracterizar de la misma forma, por ejemplo, un comportamiento para un juego de fútbol o para un juego de acción. Cada uno de ellos necesitará un conjunto de atributos diferente. El conjunto de atributos se especifica en el modelo de juego (esbozado en la sección 3.1.1), junto con los sensores y los actuadores disponibles. Para cada uno de los atributos hay que indicar:

- El nombre.
- Los valores mínimo y máximo que pueden admitir.
- Un valor por defecto para el atributo.

Se pueden definir tantos atributos como sea necesario. Los atributos escogidos deben ser suficientes y lo bastante representativos como para poder describir los comportamientos que puedan darse, pero no deben ser demasiados ya que deben definirse explícitamente para cada comportamiento incluido en la base de casos.

Aunque en principio puede parecer que los atributos numéricos son suficientes para caracterizar los posibles comportamientos, existen casos en los que puede ser necesario utilizar atributos simbólicos. Por ejemplo, en el caso de Soccerbots, se podría emplear un atributo rol, que indique el rol que desempeña el comportamiento en el equipo (delantero, defensa, portero, etc.). Este atributo no se podría expresar con un atributo numérico, ya que esto influiría en el cálculo de la similitud.

Por ello, se propone añadir, en versiones posteriores, la posibilidad de incluir atributos simbólicos y multivaluados (es decir, que admiten varios valores simultáneamente) a la descripción de los casos.

En el listado 3.2 se muestra la definición de atributos para Soccerbots.

Descripción

Este parámetro contiene una descripción textual del comportamiento. La descripción cumple una doble función durante el proceso de razonamiento basado en casos. Por un lado, permite recuperar comportamientos aplicando CBR textual. Además, al recuperar varios casos, el usuario puede ver las descripciones, lo que facilita la elección del caso más adecuado.

Comportamientos incluidos

Mediante este parámetro el usuario puede especificar qué comportamientos o qué actuadores estarán subordinados jerárquicamente al comportamiento recuperado. Por ejemplo, en el caso de Soccerbots se podría realizar una consulta con un valor de 5 en

```
...
<prop:propiedades>
  <prop:propiedad nombre="Movilidad" min="0" max="5"
     valDefecto="4"/>
  <prop:propiedad nombre="Ataque" min="0" max="5"
     valDefecto="3"/>
  <prop:propiedad nombre="Defensa" min="0" max="5"
     valDefecto="2"/>
  <prop:propiedad nombre="Portero" min="0" max="1"
     valDefecto="0"/>
</prop:propiedades>
...
```

Listado 3.2: Ejemplo de definición de atributos

el atributo "ataque" y que incluyera el comportamiento "delantero", para garantizar que el comportamiento recuperado fuera el adecuado para un delantero.

En cuanto a la estructura de la base de casos, el editor de comportamiento permite utilizar distintos formatos. Para ello se vale de la clase `Cargador`, una clase abstracta extendida por cada uno de los cargadores de base de casos. La misión de un cargador es cargar y guardar los casos que conforman la base de casos, así como otras estructuras auxiliares utilizadas por el sistema.

Durante el desarrollo de la aplicación se han utilizado dos cargadores diferentes. En primer lugar, se optó, por su simplicidad, por un cargador que utilizaba ficheros de texto plano. Los ficheros manejados por este cargador sólo admitían atributos numéricos en la descripción de los casos, pero no la descripción textual ni los comportamientos incluidos.

Más adelante, se ha implementado un cargador capaz de leer y escribir ficheros xml que incluyen los tres tipos de parámetros descritos anteriormente. La estructura de estos ficheros se muestra en el esquema de la figura 3.10.

En primer lugar se incluye una descripción del modelo al que se ajustan los casos de la base de casos. A continuación hay una lista de casos, en la que se asignan valores para cada uno de los parámetros descritos en el modelo. Por último, se incluyen una serie de estructuras auxiliares utilizadas en distintos puntos del proceso de razonamiento basado en casos: el diccionario y los dominios de los atributos simbólicos. El primero es un diccionario inverso, que asocia a cada palabra que aparece en las descripciones de los casos el nombre de los casos en que aparece y la frecuencia. Mediante este diccionario se acelera el cálculo de similitud para el parámetro descripción. La lista de dominios contiene los posibles valores que pueden aparecer en cada atributo simbólico. Este atributo se utiliza para presentar al usuario los diferentes valores posibles al realizar la consulta.

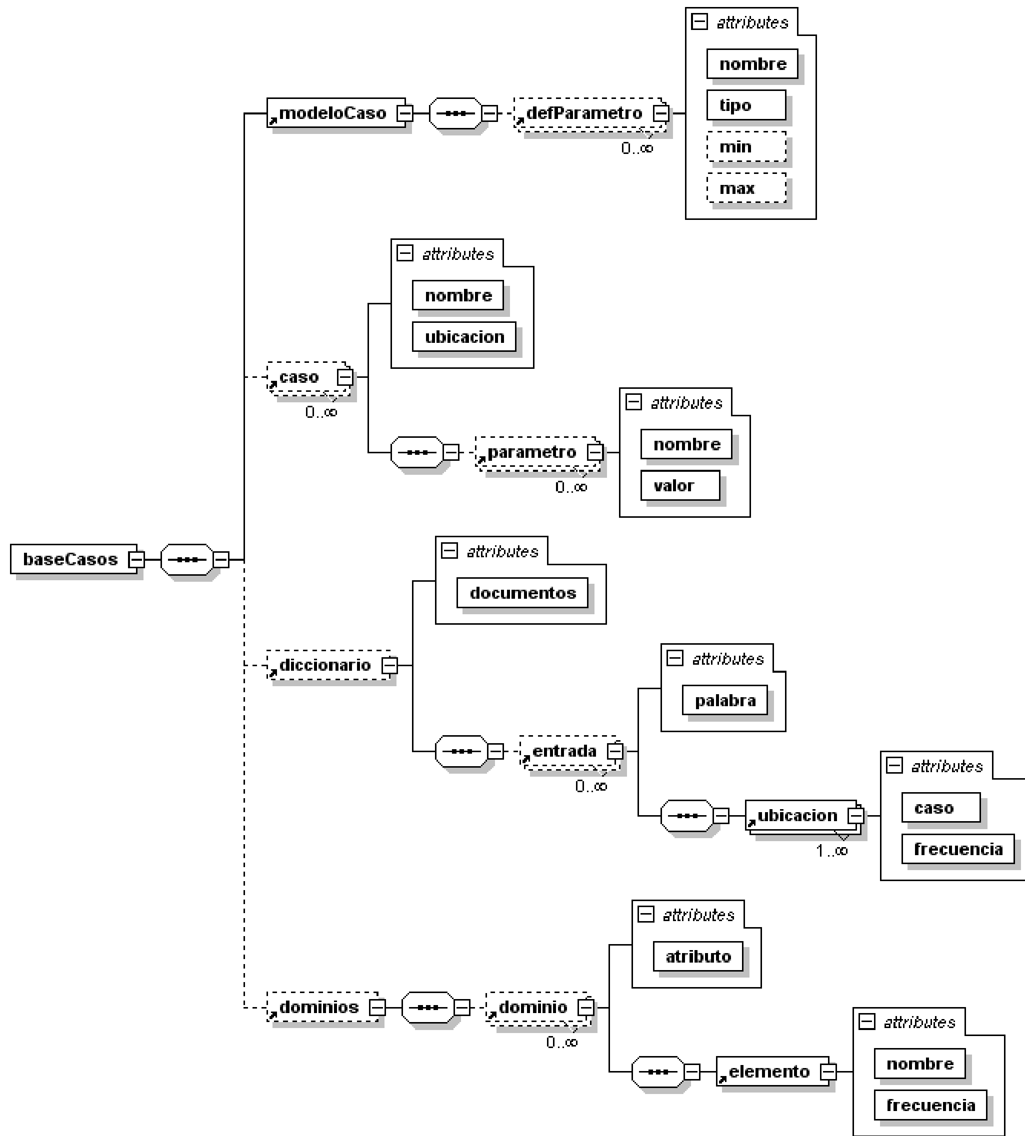


Figura 3.10: Estructura de la base de casos

Recuperación

En la fase de recuperación se obtienen de la base de casos los comportamientos que guardan mayor similitud con la consulta realizada por el usuario.

Para realizar la consulta se presenta al usuario una interfaz gráfica con los diferentes parámetros, a los que debe asignar una serie valores. Mediante esta interfaz también es posible seleccionar el modelo que se aplicará para el cálculo de la similitud, y los parámetros de este modelo (por ejemplo, si se trata de la media ponderada, los pesos que deben aplicarse a cada valor).

La base de casos

En la base de casos se encuentran las descripciones de todos los casos que se ajustan a un modelo de juego concreto. Esto es así porque cada modelo de juego impone su propia lista de sensores, actuadores y atributos. En futuras versiones se contempla la posibilidad de implementar un mecanismo de importación de casos, que permita traducir comportamientos de un modelo de juego a otro declarando equivalencias entre los sensores, actuadores y atributos de los distintos modelos.

La estructura de la base de casos se corresponde con una representación plana, es decir, en la que todos los casos tienen los mismos atributos, y está organizada de manera lineal: los casos se encuentran uno a continuación de otro, sin ninguna clasificación. Se ha escogido esta organización porque resulta simple de gestionar y mantener, pero no se descarta emplear una organización estructurada si se observa que las bases de casos tienden a crecer mucho.

Además de la información de los casos descrita en la sección anterior, en la base de casos se incluye un índice inverso para facilitar el cálculo de similitud cuando se aplica el CBR textual. Este índice asocia cada palabra aparecida en las descripciones de los casos con los casos en los que aparece y la frecuencia de apariciones en cada uno de ellos. De esta forma, es más eficiente el cálculo de la similitud. La desventaja de este método es que, cada vez que se añade o se modifica un caso de la base de casos, es necesario volver a generarla.

Similitud local

Diferentes modelos de cálculo de similitud son aplicables para los distintos grupos de parámetros que aparecen en la consulta.

- Atributos: cada uno de los atributos es un parámetro numérico univaluado, y es aplicable, por ejemplo, la función mostrada en la tabla 2.4.
- Descripción: al ser un texto plano, no estructurado, lo más adecuado es aplicar alguna función de similitud para CBR textual. En este caso, se opta por el modelo del espacio vectorial. Según este modelo, cada texto se interpreta como un vector, cuyas componentes son las frecuencias de aparición de las palabras. El valor de similitud se obtiene calculando el ángulo que forman los vectores de la consulta y del caso, como se explica en la sección 2.3.2.
- Comportamientos: se trata de un parámetro simbólico multivaluado. Para calcular la similitud local se puede aplicar cualquiera de las fómulas mostradas en la tabla 2.4.

Se propone, en futuras versiones del editor, incluir en la consulta y en el cálculo de similitud el propio nombre del comportamiento, ya que suelen ser nombres descriptivos, que aportan información sobre su funcionalidad. Para el cálculo de similitud entre el nombre en la consulta y el nombre de los casos se podrían aplicar los métodos de CBR textual, incluyendo la extracción de raíces y el uso de diccionarios y tesauros para encontrar relaciones lingüísticas entre términos.

Similitud global

Para el cálculo de la similitud global se toma como entrada la lista de valores de similitud local de cada uno de los parámetros anteriores y se aplica alguno de los operadores de cálculo mencionados en la sección 2.3.1. El resultado es un grado de similitud entre el caso propuesto en la consulta y el caso de la base de casos.

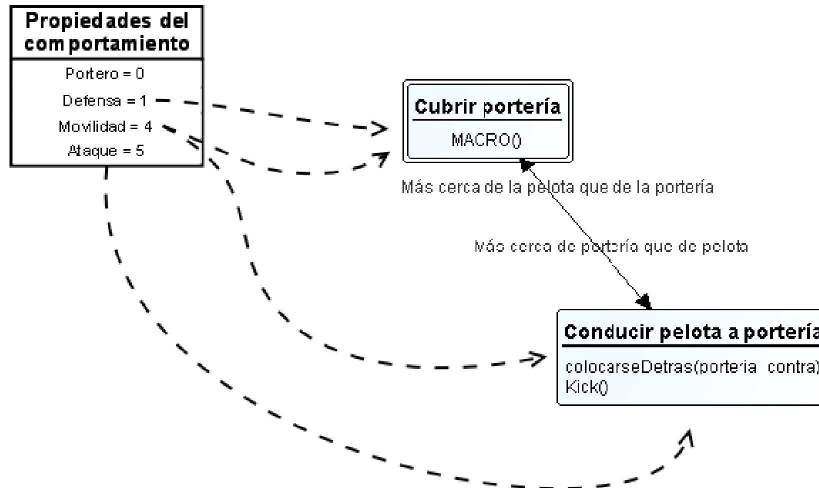


Figura 3.11: Relación entre los atributos y los nodos de un comportamiento

Durante la fase de recuperación se obtienen los k casos más similares a la consulta proporcionada por el usuario. De entre ellos, el usuario puede elegir el que considere más adecuado.

Adaptación

El proceso de adaptación es manual. Partiendo del caso recuperado, el usuario debe modificarlo para conseguir el resultado deseado.

El sistema no puede realizar la adaptación de manera automática, pero puede ofrecer soporte al usuario para llevarla a cabo, señalando los estados que puede ser conveniente modificar. Para ello, es necesario conocer la relación entre los atributos y los nodos del comportamiento que justifican su valor. Esto se consigue haciendo explícita la relación atributos–nodos. Una vez recuperado el caso, el sistema evalúa qué atributos son diferentes en la consulta y el caso y señala al usuario los nodos relacionados con ellos para que sean modificados.

En la figura 3.11 se puede ver un ejemplo de este proceso. El nodo `Cubrir portería` está relacionado con los atributos `Defensa` y `Movilidad`, y el nodo `Conducir pelota a portería` con `Movilidad` y `Ataque`. Esto quiere decir que si, por ejemplo, este comportamiento se ha recuperado con motivo de una consulta en la que el atributo `Ataque` vale 3, el sistema indicará que el nodo que se debe modificar es `Conducir pelota a portería`, ya que es el responsable de este valor. Esta asociación entre los descriptores del comportamiento (atributos) y los nodos debe ser realizada por el usuario, al guardar en disco el comportamiento.

El tipo de adaptación empleado es transformacional, ya que se transforma el comportamiento recuperado para que se ajuste a las características del comportamiento deseado, indicadas en la consulta. En concreto, se realiza una adaptación por ajuste de parámetros, que consiste en sustituir determinados valores de la solución (nodos y aristas) en base a sus diferencias con la consulta, aplicando determinadas reglas de ajuste.

Aprendizaje

El usuario es, en gran medida, responsable del aprendizaje del sistema. Por un lado, el aprendizaje está dirigido por el usuario: es el usuario quien decide qué comportamientos son lo bastante relevantes o reutilizables como para que sean almacenados en la base de casos. Además, también es responsabilidad del usuario asignar unos valores a los atributos y a la descripción lo bastante representativos como para definir la funcionalidad del comportamiento.

Dada la organización lineal de la base de casos, es sencillo añadir un nuevo caso. Solamente hay que colocarlo a continuación de los existentes. Otra tarea necesaria y algo más costosa es actualizar el índice de palabras que utiliza el CBR textual. Para ello hay que buscar cada palabra que forma la descripción del caso en el índice y asociarla con el caso.

3.2.2. Integración en el editor

En esta sección se detalla cómo se integra en el editor de comportamientos el sistema CBR descrito anteriormente.

A grandes rasgos, existen dos escenarios para los que se utilizan las consultas por funcionalidad. En el primero de ellos se parte de un comportamiento que está siendo editado, y se consulta la base de casos en busca de un estado que cumpla determinadas propiedades, para completar el comportamiento editado. Otra posibilidad es utilizar la búsqueda para, partiendo de un comportamiento en blanco, encontrar un comportamiento inicial y editarlo hasta conseguir el resultado deseado.

Por supuesto, ambos escenarios pueden combinarse, buscando en primer lugar un comportamiento inicial y luego modificándolo o añadiendo estados también contenidos en la base de casos.

Para invocar al motor de búsqueda basada en CBR se utiliza el botón correspondiente de la barra de herramientas mostrada en la figura 3.7 de la página 59.

Esto lanza la interfaz gráfica de usuario que permite introducir los parámetros de la búsqueda. En la figura 3.12 se muestra una captura de pantalla correspondiente a la primera versión del editor.

Para completar la consulta es necesario, en primer lugar, seleccionar una base de casos, utilizando el botón que se encuentra en la primera línea. La base de casos seleccionada en el ejemplo de la figura es `Soccerbots.bc.xml`, que se corresponde con el modelo de juego de Soccerbots. A continuación se muestra el número de casos que forman la base de casos, 21 en este ejemplo. El siguiente apartado es la consulta en sí. Aquí se pueden rellenar los valores para cada uno de las propiedades y escoger la función de similitud global. En la versión actual del editor, el cálculo de la similitud global se realiza utilizando un agregador, que resume los valores de similitud de cada uno de las propiedades en un solo valor representativo. En el ejemplo antes mencionado, se ha seleccionado la media aritmética. A continuación se introducen los valores de las propiedades de la consulta y se seleccionan las funciones de similitud local asociadas. Para cada grupo de propiedades se permite seleccionar una función de similitud adecuada su tipo, como las mencionadas en la sección 2.3. En esta versión, se ha implementado una función para cada grupo. En el caso de los atributos numéricos se utiliza la distancia normalizada, para la descripción textual, el modelo del espacio vectorial, y para los atribútos simbólicos multivaluados el cociente entre la intersección y la unión.

Por último, el usuario puede introducir el número de casos que desea recuperar de la base de casos. En el ejemplo se ha introducido un 5, de manera que se recuperarán los 5 casos de mayor similitud, ordenados por su similitud con la consulta.

Al aceptar los valores de la consulta, el sistema realiza la recuperación de los casos. En

Base de casos: \owner\Mis documentos\eCo\Bases de casos\Soccerbots.bc.xml

Propiedades de la base de casos

Número de registros 21

Consulta

Agregador: Media aritmética

Atributos — Distancia normalizada

Portero 1.0

Defensa 4.0

Movilidad 2.5

Ataque 0.5

Descripción — Espacio vectorial

Comportamiento de portero capaz de subir hasta el centro del campo

Comportamientos — Intersección-Unión

Disponibles: Golpear pelota, Kick, Bloquear portero, Portero, bloquear, MACRO, Centro, Ir a centro

Asignados: Cubrir portería

Registros a devolver

Número de registros: 5

Cancelar Aceptar

Figura 3.12: Parámetros de una consulta a la base de casos

Nombre	Puntuación
Ir a la portería	0,489
Chafaris	0,535
Bloquear delantero	0,55
Cubrir portería	0,623
Portero	0,724

Propiedad	Valor
Portero	1.0
Defensa	5.0
Movilidad	3.0
Ataque	0.0
descripcion	Comportamiento compuesto para un portero que se mantiene cerca de la...
comportamientos	[Cubrir portería, Golpear pelota, Ir a la portería]

Cancelar Aceptar

Figura 3.13: Resultados de la recuperación de los casos

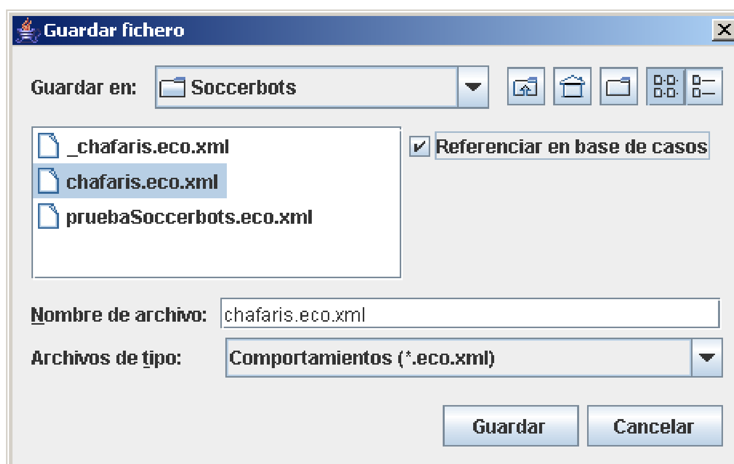


Figura 3.14: Formulario para salvar un comportamiento

la figura 3.13 se muestra, como ejemplo, el resultado de la consulta anterior. Los diferentes comportamientos recuperados, junto con su valor de similitud, se muestran en la tabla de la izquierda, mientras que a la derecha aparecen las propiedades del comportamiento seleccionado.

El aprendizaje se realiza de manera independiente a la recuperación. Es decir, que no todos los casos recuperados y adaptados posteriormente pasarán a formar parte de la base de casos, ni todos los casos de la base de casos provienen necesariamente del ciclo de CBR.

Cuando el usuario guarda en disco un comportamiento se le ofrece también la opción de almacenarlo en una base de casos. Para ello debe marcar la casilla "Referenciar en base de casos" en el formulario de salvado (figura 3.14). Hecho esto, se muestra el formulario para almacenar un caso en la base de casos.

Este formulario, como se ve en la figura 3.15, está dividido en dos partes. En primer lugar, a la izquierda, se muestra la pantalla de selección de ficheros habitual. En ella se puede elegir el fichero de base de casos de destino. En este caso, se ha escogido el fichero "Soccerbots.bc.xml". Si se introduce un nombre que no corresponde a ningún fichero existente se creará una nueva base de casos. A la derecha se muestra un árbol en el que aparece representado jerárquicamente el comportamiento que se va a almacenar en la base de casos. Seleccionando cada uno de los comportamientos y subcomportamientos se permite al usuario que se almacenen como casos independientes. En el ejemplo de la figura, se añadirán cuatro casos a la base de casos: el caso *Chafaris*, un comportamiento compuesto y raíz de la jerarquía, el caso *Golpear pelota*, también un comportamiento compuesto, y los casos *Chutar* y *Cubrir portería*, ambos, comportamientos atómicos.

3.2.3. Consultas por estructura

En determinadas situaciones, puede no ser suficiente especificar la funcionalidad del comportamiento deseado, sino que se desea, además, recuperar un comportamiento que se ajuste, en la medida de lo posible, a una estructura de nodos y aristas concreta. Las consultas por estructura tratan de resolver este problema incluyendo dentro de la propia consulta un patrón de comportamiento. Un patrón de comportamiento es un comportamiento en el que se ha dejado sin definir el contenido asociado a los nodos y las aristas. En la figura 3.16 se muestra un ejemplo de patrón de comportamiento formado por tres nodos conectados entre

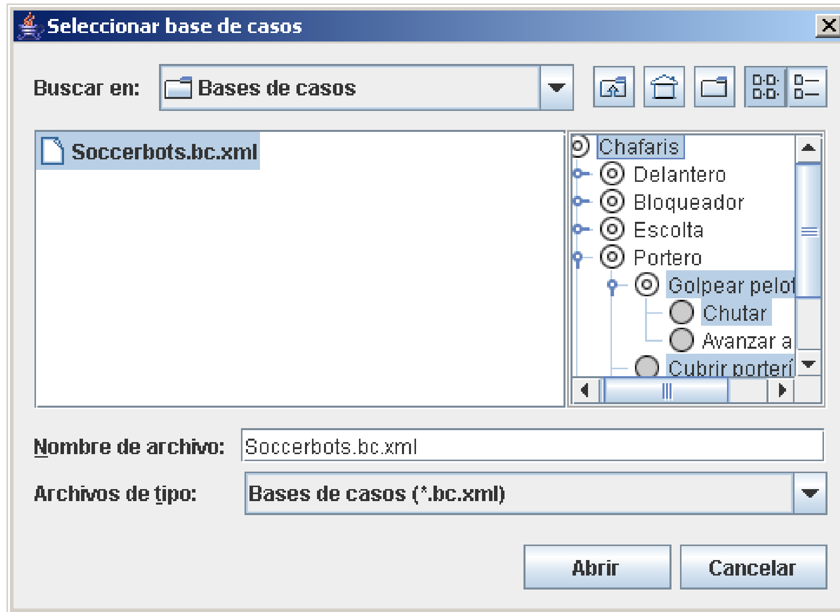


Figura 3.15: Guardar un caso en la base de casos

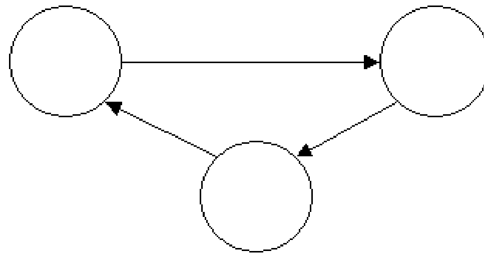


Figura 3.16: Ejemplo de un patrón de comportamiento

sí por tres aristas.

Las consultas para realizar este tipo de búsquedas estarán compuestas por tres elementos:

- Un patrón de comportamiento: como se explica en el párrafo anterior, mediante este elemento se indica la estructura deseada que se desea recuperar de la base de casos.

- Especificación de la funcionalidad del comportamiento: mediante este elemento se puede ajustar mejor la búsqueda de los comportamientos indicando la funcionalidad que se desea que cumpla el comportamiento recuperado. Para especificarla se utiliza un consulta por funcionalidad como las vistas en la sección 3.2.1.

- Especificación de la funcionalidad de los nodos incluidos en el patrón: para cada uno de los nodos incluidos en el patrón se puede indicar una consulta por funcionalidad que refine aún más el conjunto de resultados obtenidos en la recuperación aproximada. Mediante estas consultas se puede especificar la funcionalidad que se desea que cumplan los comportamientos asociados a cada nodo del patrón.

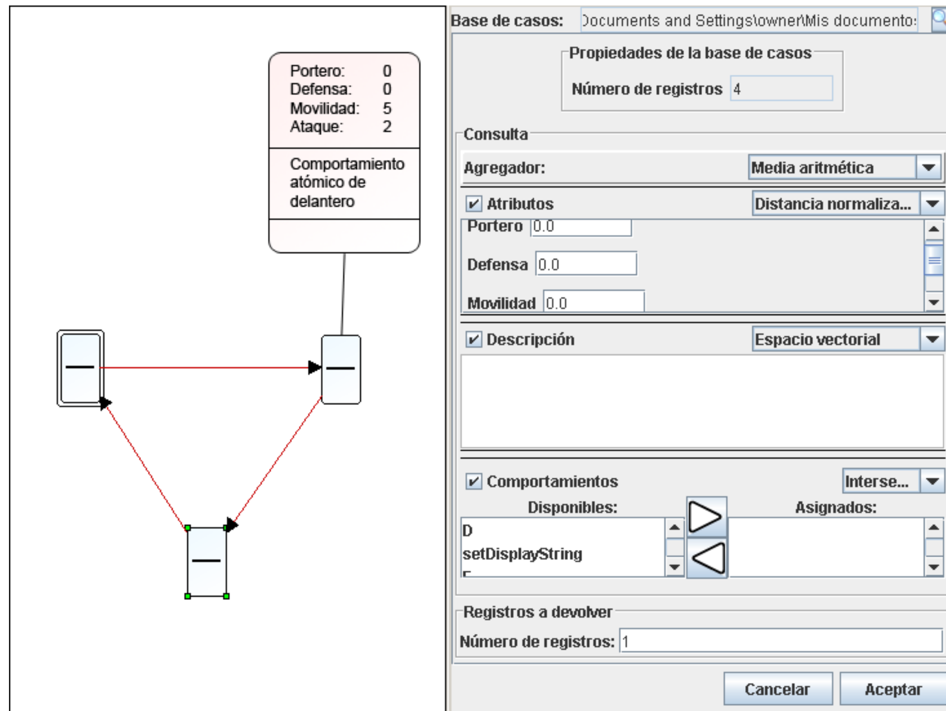


Figura 3.17: Interfaz gráfica de usuario para realizar consultas por estructura

Cada uno de estos elementos es opcional. Si no se especifica un valor para alguno de estos elementos, no se tiene en cuenta para el cálculo de la similitud.

La base de casos deberá incluir información referente a la estructura de los comportamientos asociados a cada uno de los casos. Esta información dependerá en gran medida de la función de similitud empleada para comparar los patrones con los comportamientos. La función de similitud será una agregación de las funciones de similitud de cada uno de los elementos antes mencionados.

Para la funcionalidad del comportamiento se emplea una función de similitud como las utilizadas para las consultas por funcionalidad. Para comparar el patrón de comportamiento se puede utilizar alguna función de comparación de grafos [49, 6, 7]. Por último, si se especifican funcionalidades para los nodos del patrón, se utilizará de nuevo una función de similitud como la de las consultas por funcionalidad.

En la figura 3.17 se muestra un ejemplo de consulta por estructura. A la izquierda se define el patrón de comportamiento mientras que a la derecha se muestra la funcionalidad deseada para el comportamiento recuperado. Uno de los nodos del patrón se ha asociado con una consulta, que se muestra en el recuadro rojo.

Una vez se han recuperado los casos, se presentan al usuario y se le da la posibilidad de realizar la adaptación. El usuario puede seleccionar cada uno de los nodos para adaptarlo. En ese caso, se buscan en la base de casos comportamientos similares a la consulta asociada al nodo, como se muestra en la figura 3.18. Esto se realiza como si se tratara de una consulta por funcionalidad. El comportamiento del nodo se sustituye por el comportamiento recuperado.

Al sustituir el comportamiento en el patrón recuperado, las fórmulas asociadas a las aristas que salen o entran en él pueden dejar de tener sentido. La información que aportan las fórmulas de cada arista es muy ambigua con respecto al significado global del compor-

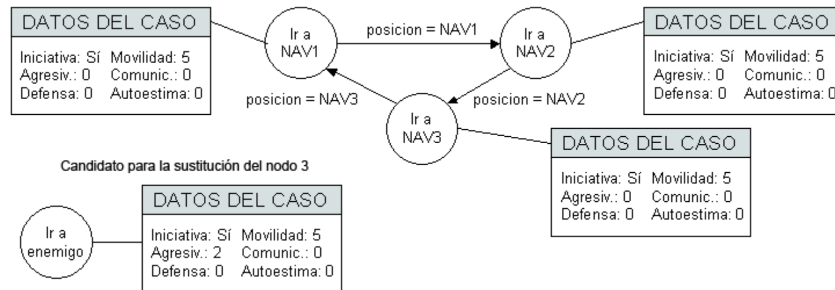


Figura 3.18: Adaptación de un nodo en un comportamiento recuperado

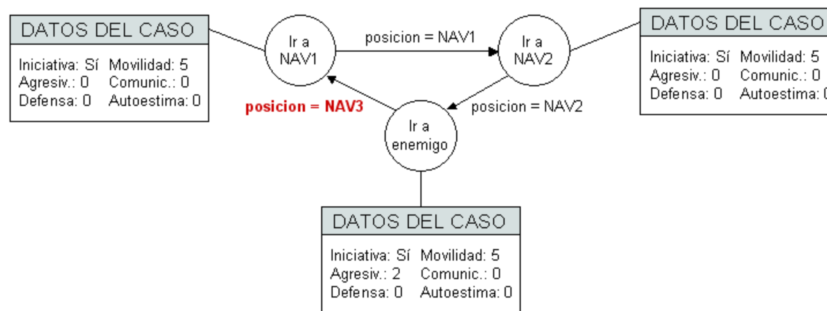


Figura 3.19: Perdida de consistencia en las aristas al sustituir un nodo

tamiento. Por ejemplo, una arista con la fórmula `distancia_enemigo <5`, puede aparecer en un comportamiento agresivo (si la distancia al enemigo es menor que 5 atacar) o huidizo (si la distancia al enemigo es menor que 5 huir) indistintamente. Por eso, el usuario debe prestar especial atención a las aristas asociadas a los nodos sustituidos durante la adaptación para asegurar la consistencia de las fórmulas con los nuevos comportamientos añadidos. En la figura 3.19 se muestra un ejemplo de cómo puede perderse la consistencia al sustituir un nodo.

3.2.4. El sistema CBR fuera del editor

El conjunto de clases java que forman el sistema CBR constituyen un framework que puede ser utilizado fuera del editor. De esta manera, las ventajas que ofrece el razonamiento basado en casos en la búsqueda de comportamientos, pueden integrarse en otras aplicaciones.

Mediante el framework se pueden crear, cargar o guardar bases de casos, utilizando para ello cualquier cargador de bases de casos que esté implementado (ver sección 3.2.1), y, por supuesto, realizar consultas de recuperación aproximada contra una base de casos cargada. Es esta última funcionalidad la que presenta más ventajas para ser utilizada fuera del editor de comportamientos.

En la figura 3.20 se muestra un diagrama de clases con las clases principales de este framework. A continuación se describen brevemente algunas de ellas.

Existen dos formas equivalentes de realizar una consulta a una base de casos. Se puede utilizar el método estático `evaluar` de la clase `Motor` o el método `evaluar` de la clase `BaseCasos`. Ambos devuelven una lista de casos evaluados (`ListaCasosEvaluados`). Esta

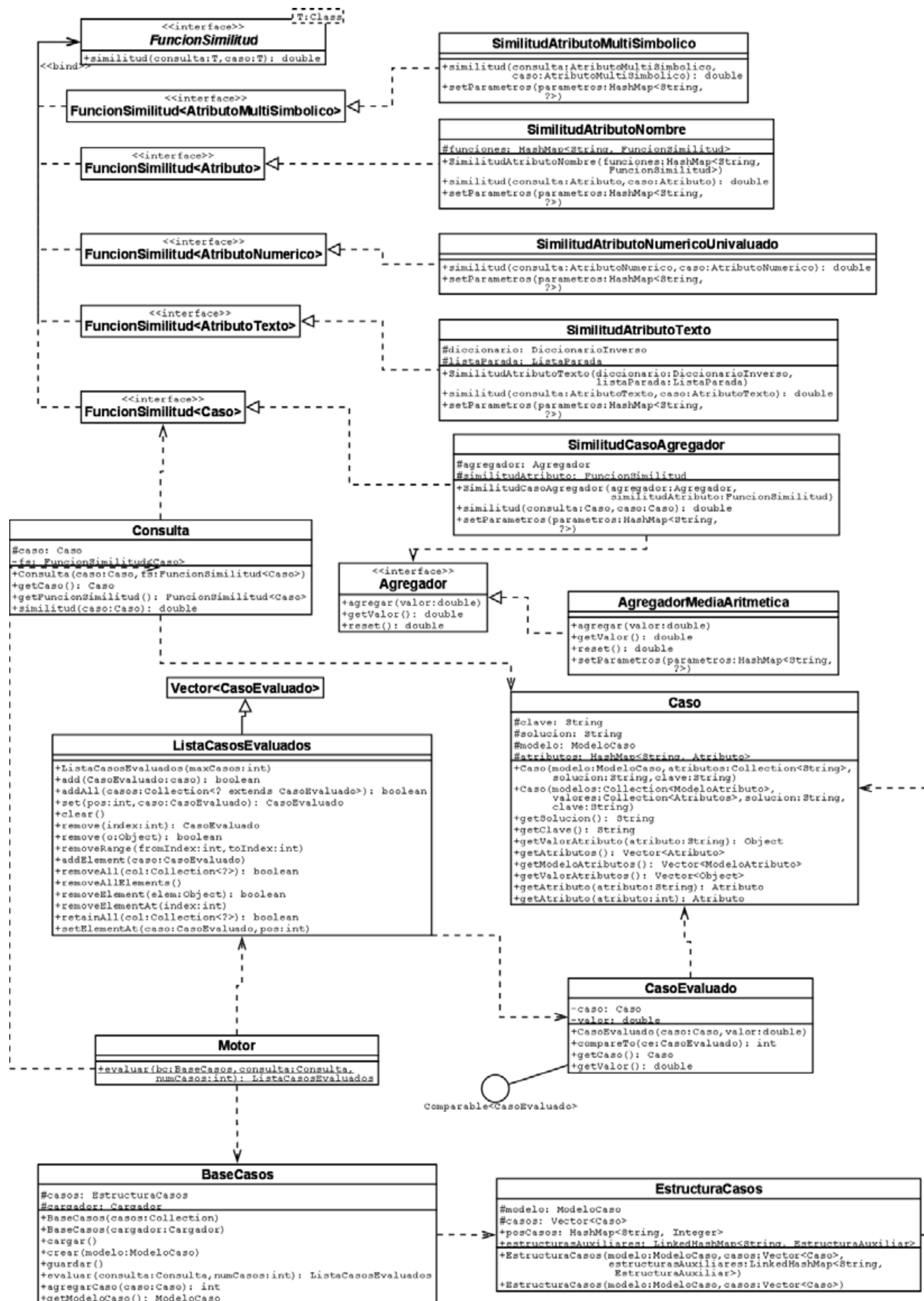


Figura 3.20: Diagrama de clases del sistema CBR

lista es un vector cuyos elementos son de la clase `CasoEvaluado`. Esta clase únicamente relaciona un caso con un valor de tipo `double` que almacena el valor de similitud con la consulta. La lista de casos evaluados se mantiene ordenada en orden creciente de similitud.

El método evaluar necesita una base de casos y una consulta. La base de casos contiene los casos sobre los que se realiza la consulta. Generalmente se cargará de un fichero en disco utilizando un cargador.

En la consulta se especifican una serie de valores para los atributos del caso, que se dan como un caso, y la función de similitud que se emplea para compararla con los casos de la base de casos. Es importante que coincidan los atributos de la consulta con los que aparecen en los casos de la base de casos. Para ello, se puede obtener el modelo de los casos de la base de casos, mediante el método `getModeloCaso`, y utilizarlo para crear el caso que se incluye en la consulta.

El sistema de razonamiento basado en casos descrito anteriormente puede ser utilizado en diversos escenarios. Por ejemplo, en lugar de especificar comportamientos concretos para las entidades de un juego en tiempo de diseño, se pueden dar una serie de atributos que describan los comportamientos deseados. Posteriormente, durante la ejecución, se realizará una consulta a una base de casos, con los comportamientos disponibles, de manera que se recuperen los más adecuados. De entre los comportamientos más similares a la consulta realizada se puede elegir el que presente mayor similitud o, por ejemplo, uno aleatorio, de manera que el comportamiento asociado a la entidad pueda variar durante el juego, pero siempre presentando unas características concretas (que vienen dadas por la consulta). Esto también permite tener un repositorio de comportamientos, que puede ir creciendo, con nuevos comportamientos añadidos, o disminuyendo, si, por ejemplo, se observa algún comportamiento que no es adecuado, sin que por ello sea necesario volver a compilar el juego o modificar un `script`.

3.3. Procedimiento de aplicación

En esta sección se explica, mediante un ejemplo práctico, cómo crear un comportamiento utilizando el editor, para ilustrar así algunas de sus funciones básicas.

Aunque no es estrictamente necesario, el paso inicial es diseñar el comportamiento “con lapiz y papel”, para hacernos una idea aproximada de su estructura. En este caso utilizaremos el comportamiento de la figura 3.21.

Al arrancar la aplicación se muestra la pantalla de inicio (figura 3.22) en la que se puede elegir entre crear un nuevo comportamiento o cargar un comportamiento ya existente. Si se elige crear un nuevo comportamiento, se mostrará un diálogo para escoger el modelo de juego que se va a emplear, como se muestra en la figura 3.23. El modelo de juego debe haberse creado antes de editar los comportamientos. El comportamiento que se va a diseñar es para el entorno de simulación de Soccerbots, por lo que en esta pantalla se escogerá el modelo de juego correspondiente: `soccerbots.mj.xml`

A partir de este punto se puede empezar a editar el comportamiento con las herramientas de creación de nodos y conectores.

El editor está concebido para seguir una filosofía de construcción de comportamientos de arriba hacia abajo, es decir, editar primero los comportamientos situados a un nivel jerárquico más alto, para después ir creando los subcomportamientos que estos incluyen. Esto no impide que pueda ser utilizada a la inversa, aunque es, quizás, algo más engorroso. Es necesario crear los comportamientos que se encuentran a un nivel más bajo en la jerarquía y almacenarlos en disco. Después, se crea el comportamiento de nivel superior y se recuperan en él mediante el sistema CBR o simplemente se cargan de disco.

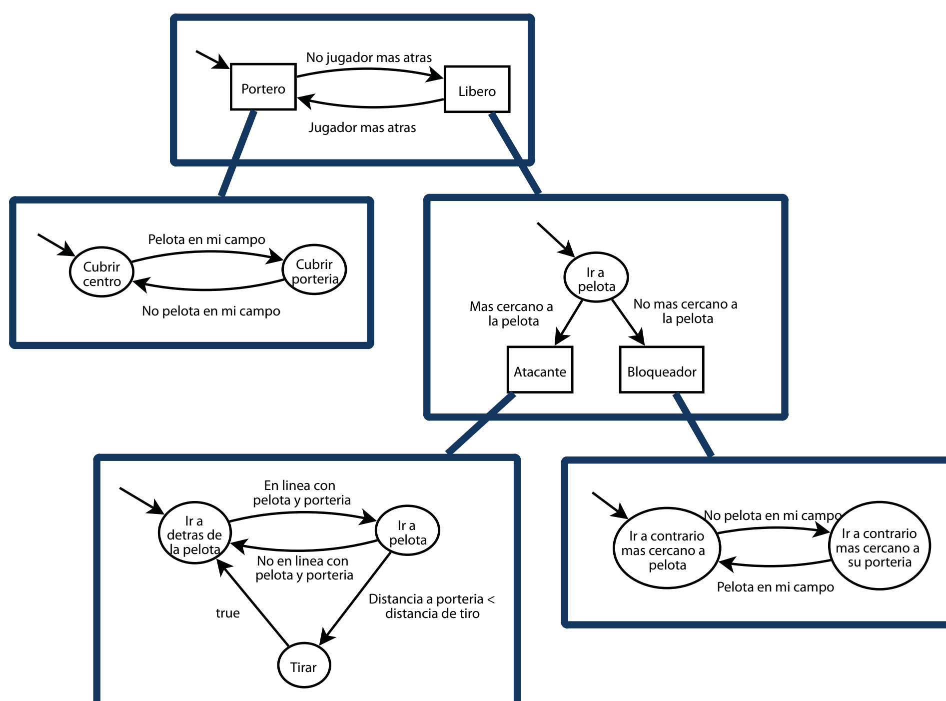


Figura 3.21: Comportamiento de ejemplo

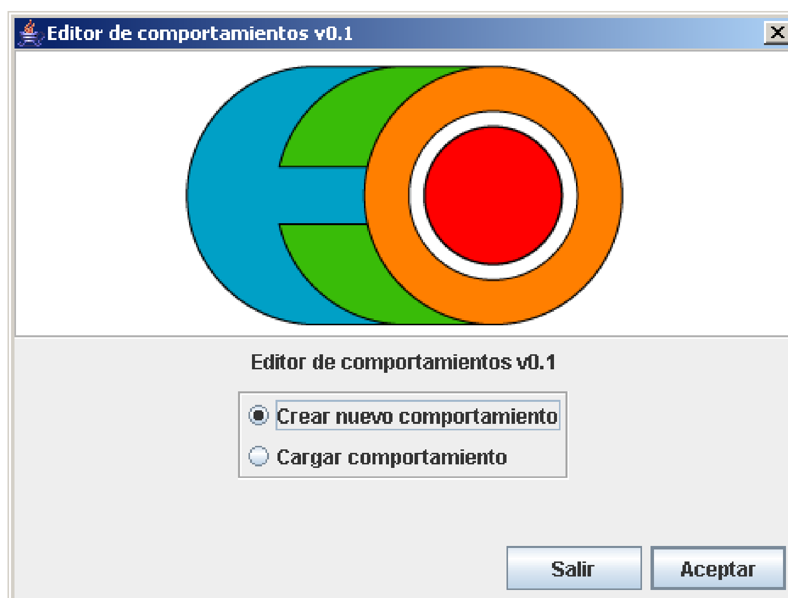


Figura 3.22: Pantalla de inicio

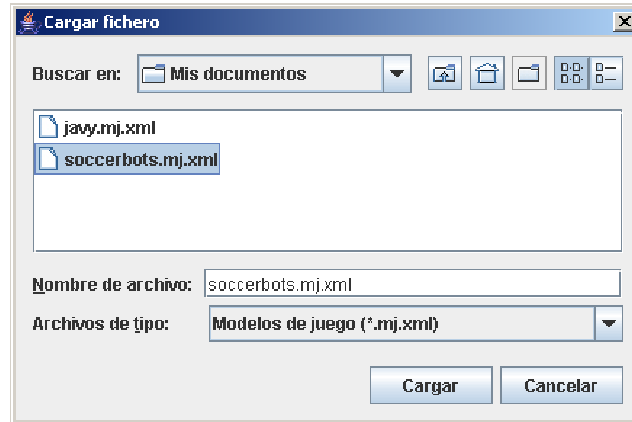


Figura 3.23: Carga del modelo de juego

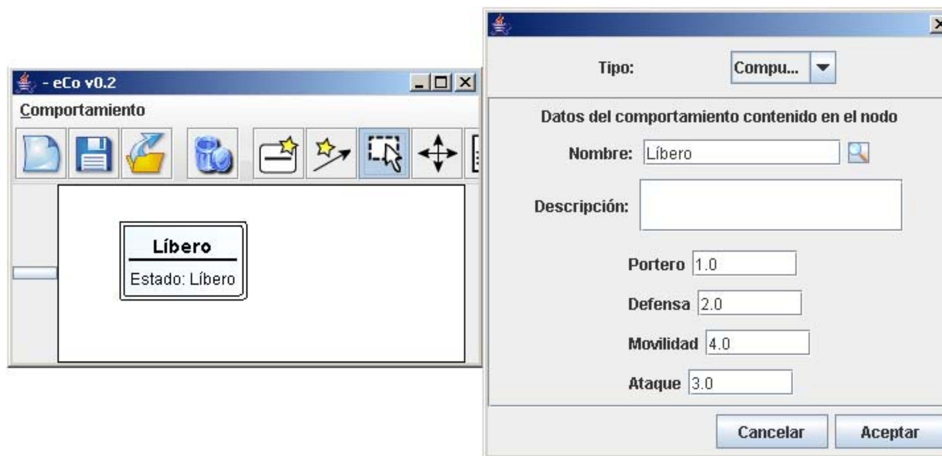


Figura 3.24: Configuración de los nodos

En este caso, se comienza creando uno de los nodos, para el comportamiento compuesto `Líbero`.El siguiente paso es fijar las propiedades del nodo. Este nodo representa un comportamiento compuesto, y así se selecciona en el desplegable correspondiente. La otra propiedad necesaria para completar el nodo compuesto es el nombre del comportamiento contenido en él, que se solicita al usuario en el momento en que selecciona el desplegable. Las restantes propiedades atañen al sistema CBR y no entraremos en ellas por el momento. El resultado de esta tarea se muestra en la figura 3.24.

El diseño continúa descendiendo por la jerarquía de comportamientos, con la herramienta inspeccionar, con el comportamiento `Líbero`, hasta completar todos los comportamientos atómicos. El comportamiento `Líbero` esta formado por un comportamiento atómico, `Ir a pelota`, y dos comportamientos compuestos, `Atacante` y `Ayuda`. El comportamiento atómico necesita que el robot gire hasta apuntar en dirección a la pelota, y se mueve hacia ella. En la versión actual, no es posible asignar a un actuador el resultado de un sensor, por lo que es necesario utilizar el actuador `MACRO`. En la figura 3.25 se muestran las propiedades del comportamiento y el contenido de la macro.

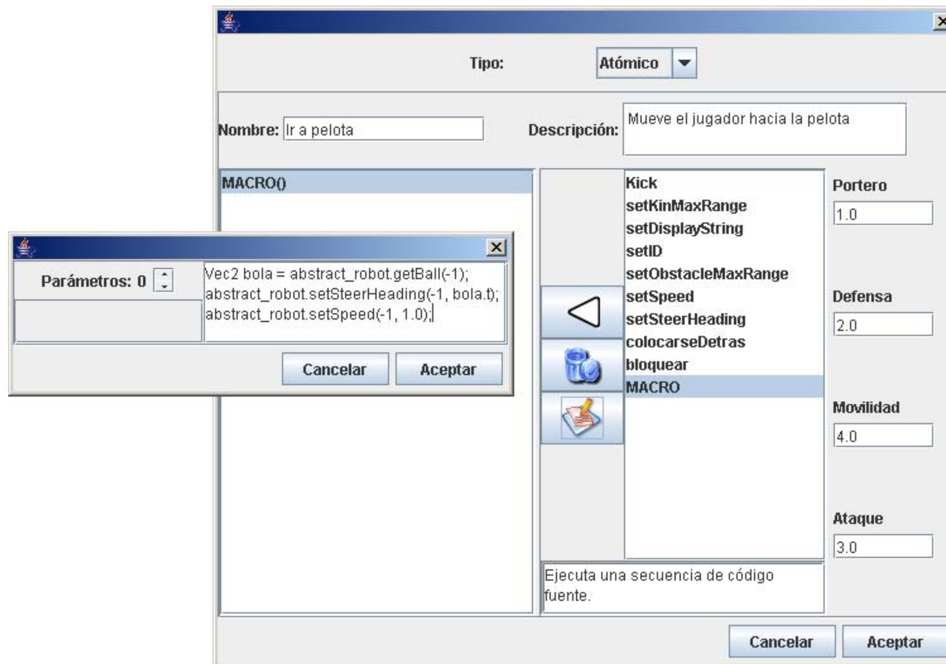


Figura 3.25: Configuración del comportamiento `Ir a pelota`

Los restantes estados y aristas se definen de manera similar, utilizando los comportamientos compuestos, los sensores, los actuadores o el actuador `MACRO` según sea necesario.

Para mostrar el funcionamiento del sistema de razonamiento basado en casos, en lugar de introducir manualmente el comportamiento `Portero`, haremos una búsqueda en la base de casos.

Al invocar el asistente de búsquedas se muestra el formulario correspondiente. En primer lugar, se debe abrir una base de casos. A continuación, se introducen los parámetros de la búsqueda y se seleccionan las medidas de similitud. En este caso, se introducen los parámetros mostrados en la figura 3.26. De esta manera se busca un comportamiento con funciones de portero ($Portero = 1$), defensivo ($Defensa = 4$), con cierta movilidad ($Movilidad = 2$) y que no se preocupe por el ataque ($Ataque = 0$). Mediante la descripción se ajusta aún más esta búsqueda, incluyendo algunas características no contempladas en los atributos (que se mantenga cerca de la portería). Por último, se da mayor peso a los casos que incluyan el comportamiento (en este caso, el actuador) `bloquear`. En la figura 3.27 se muestran los resultados de esta búsqueda. El resultado de mayor similitud es un comportamiento compuesto que "se mantiene cerca de la portería", aunque no incluye el comportamiento `bloquear`. Si seleccionamos este caso, se añadirá al comportamiento actual.

En cuanto a las aristas, a la que va del portero al líbero, hay que asignarle la condición "`MasAtras = false`" y a la que va en sentido contrario, "`MasAtras = true`". En la figura 3.28 se muestra el formulario de configuración de una de las aristas. El resultado final puede verse en la figura 3.29.

Una vez que se ha creado un comportamiento existen varias opciones:

- Guardar el comportamiento en disco. De esta manera se puede tener acceso al comportamiento posteriormente para su edición.

Figura 3.26: Consulta CBR

- Guardar el comportamiento en la base de casos para poder utilizarlo en el ciclo CBR.
- Generar el código fuente correspondiente al comportamiento utilizando un generador de código adecuado.

Las dos primeras van muy ligadas, ya que, en la base de casos se almacena la ruta en disco del comportamiento. Antes de añadir un comportamiento a la base de casos, es necesario asignar valores a sus atributos, y estos deben describir lo más exactamente posible el comportamiento añadido.

Nombre	Puntuación
Ir a la portería	0,527
Chafaris	0,529
Bloquear delantero	0,633
Cubrir portería	0,737
Portero	0,767

Propiedad	Valor
Portero	1.0
Defensa	5.0
Movilidad	3.0
Ataque	0.0
descripcion	Comportamiento compuesto para un portero que se mantiene cerca de la portería y va a por la pelota ...
comportamientos	[Cubrir portería, Golpear pelota, Ir a la portería]

Cancelar Aceptar

Figura 3.27: Resultados de la búsqueda

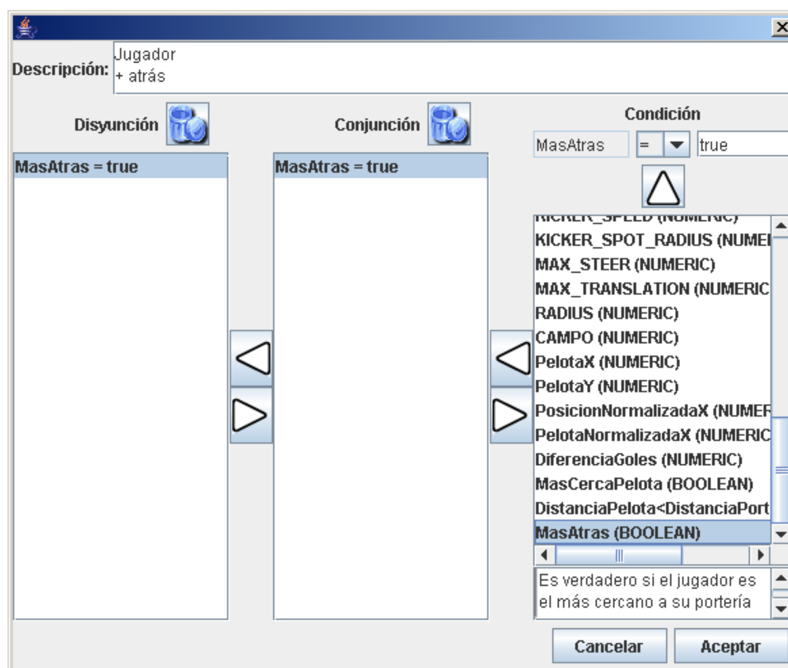


Figura 3.28: Configuración de las aristas

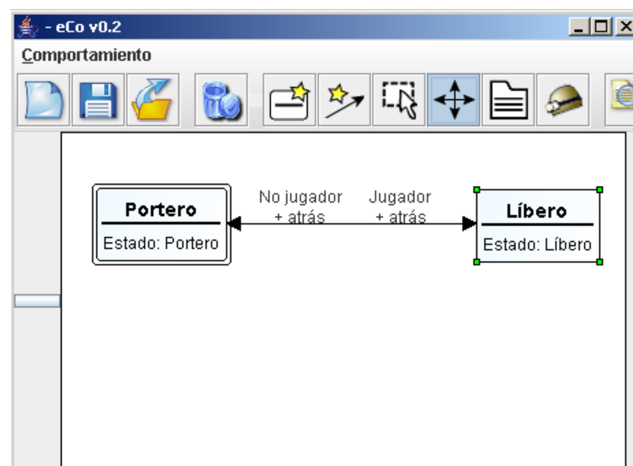


Figura 3.29: Resultado de la edición del comportamiento

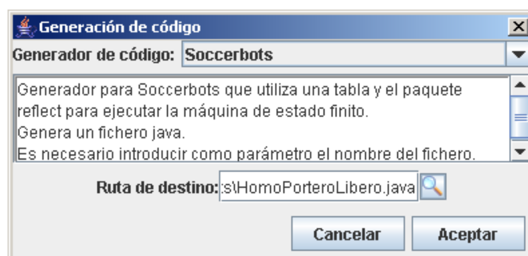


Figura 3.30: Generador de código

Para la generación de código fuente se necesita invocar a un `generador`. Para ello se utiliza el botón `Generar` de la interfaz y se selecciona el generador adecuado en el desplegable. Al seleccionar el generador, se muestra su descripción en la caja de texto situada debajo. Para proceder a la generación de código, se deben rellenar antes los parámetros que necesite el generador. En el ejemplo de la figura 3.30, el parámetro requerido es la ruta de la clase Java generada.

Capítulo 4

Evaluación del generador de comportamientos

En este capítulo se muestran algunos de los resultados obtenidos durante las pruebas del prototipo del editor de comportamientos. En primer lugar se examinan los resultados del sistema de búsqueda aproximada basado en CBR. A continuación, se muestran los pasos seguidos para integrar el editor con tres entornos diferentes y de muy distinta naturaleza: Soccerbots, un entorno de simulación de partidos de fútbol entre robots, JV^{2}M un juego de acción en primera persona con fines pedagógicos, y Neverwinter Nights, que es un juego de rol. Para finalizar, se plantea de manera breve cómo se integrarían en el editor otros juegos y entornos de simulación.

4.1. Resultados del sistema de CBR

En las primeras interacciones con el editor, al utilizar un nuevo modelo de juego, lo más habitual es partir de una base de casos vacía. El problema que esto conlleva es que, evidentemente, para editar los primeros comportamientos no se puede utilizar el sistema CBR. Además, por cómo se ha construido el editor, los parámetros en las llamadas a los actuadores sólo admiten valores constantes. No se permite, por ejemplo, una llamada del tipo *ir_a(posición_del_balón)*. Para solventar parcialmente esta carencia, se incluye un actuador por defecto llamado *MACRO*, que permite incorporar código fuente del lenguaje de destino directamente en el comportamiento, como se detalla en el capítulo 5. El código incluido dentro del actuador *MACRO* depende en gran medida de la interfaz de control que ofrece el juego y del generador de código que se empleará, por lo que es necesario un conocimiento de la arquitectura de estos elementos. Por estos motivos, es conveniente partir de una base de casos que contenga algunos casos predefinidos, que sean utilizados para componer los comportamientos más complejos. En esta base de casos inicial se pueden incluir también comportamientos de uso general que requieran del uso del actuador *MACRO*, especialmente en las situaciones en las que los diseñadores de los comportamientos no tienen un conocimiento extendido de la arquitectura del juego o del generador de código.

La base de casos inicial se puede crear mediante el editor de comportamientos, creando comportamientos simples y almacenándolos en una base de casos nueva, o editando el fichero xml de base de casos con un editor adecuado. Esta segunda opción no es muy recomendable, ya que, además de los atributos de los comportamientos, es necesario incluir otras estructuras como el diccionario de términos incluidos en las descripciones y la lista de

Nombre	Puntu...
Avanzar a pelota	0,48
Ir a centro	0,561
Conducir pelota a portería	0,705

Propiedad	Valor
Portero	0.0
Defensa	0.0
Movilidad	5.0
Ataque	4.0
descripcion	Lleva la pelota a la portería contraria
comportamientos	[colocarseDetras, Kick, setDisplayString]

Cancelar Aceptar

Figura 4.1: Resultados de la búsqueda de un delantero

subcomportamientos que aparecen en la base de casos. La estructura de este fichero viene dada por el esquema *BaseCasos.xsd*, que se incluye en el apéndice A.

Para realizar las pruebas del editor se ha utilizado una base de casos poblada por comportamientos diseñados por alumnos de la asignatura ISBC del curso 2006/2007. Se puede encontrar más información sobre los equipos participantes en el torneo de Soccerbots celebrado este año, en la página web del grupo GAIA[1].

Es importante que el número de casos en la base de casos sea suficiente y que estos sean suficientemente heterogéneos. De lo contrario, los resultados de la recuperación podrían no ser los esperados.

Por ejemplo, partimos de una base de casos con 5 casos como la mostrada en la tabla 4.1, y deseamos crear un comportamiento para un delantero que mantenga su posición en el campo contrario. Realizamos una consulta contra la base de casos con los siguientes valores de parámetros:

Consulta	
Portero	0,0
Defensa	0,0
Movilidad	1,0
Ataque	4,0
descripcion	`Delantero que no baja a defender`
comportamientos	`NO ESPECIFICADO`

Los casos recuperados se muestran en la figura 4.1. Se comprueba que el comoportamiento de mayor similitud recuperado es `Conducir pelota a portería`, con un valor de similitud de 0,705. Si abrimos el comportamiento en el editor o nos guiamos por su descripción, observaremos que su función es colocarse detrás de la pelota, mirando en dirección a la portería contraria, y darle una patada. Este comportamiento no es el que se buscaba en un principio pero, es el más adecuado de los incluidos en la base de casos. Al ser esta tan pequeña (tan solo 5 casos) el sistema no puede encontrar un comportamiento más parecido a la consulta.

Independientemente de su tamaño, la base de casos debe contener comportamientos bastante heterogéneos, si se desea que la recuperación funcione bien en todos las situaciones. Siguiendo con el ejemplo anterior, si se desea recuperar un comportamiento para un portero que permanezca cerca de la portería se podría utilizar la siguiente consulta:

[1] `http://gaia.fdi.ucm.es/grupo/projects/soccerBots/template.htm`

Conducir pelota a portería	
Portero	0.0
Defensa	0.0
Movilidad	5.0
Ataque	4.0
descripcion	Lleva la pelota a la portería contraria
comportamientos	colocarseDetras·Kick
Cubrir portería	
Portero	0.5
Defensa	5.0
Movilidad	3.0
Ataque	0.0
descripcion	Comportamiento atómico que mantiene la posición entre la pelota y la portería
comportamientos	MACRO
Avanzar a pelota	
Portero	0.0
Defensa	2.5
Movilidad	5.0
Ataque	2.5
descripcion	Se mueve hacia la pelota en línea recta
comportamientos	MACRO
Avanzar a portería	
Portero	0.5
Defensa	1.0
Movilidad	4.0
Ataque	0.0
descripcion	Se mueve hacia la portería
comportamientos	MACRO
Ir a centro	
Portero	0.0
Defensa	1.0
Movilidad	3.0
Ataque	0.0
descripcion	Comportamiento atómico que mueve al jugador hasta el centro del campo
comportamientos	MACRO

Tabla 4.1: Base de casos inicial

Consulta	
Portero	1,0
Defensa	3,0
Movilidad	1,0
Ataque	0,0
descripcion	`Portero que se mantiene cerca de la portería`
comportamientos	`NO ESPECIFICADO`

Observamos que en la base de casos no existe ningún comportamiento con movilidad baja, por lo que es imposible que mediante la recuperación se obtenga el comportamiento

Nombre	Puntuación
Ir a centro	0,499
Avanzar a portería	0,568
Cubrir portería	0,656

Propiedad	Valor
Portero	0.5
Defensa	5.0
Movilidad	3.0
Ataque	0.0
descripcion	Comportamiento atómico que mantiene la posición entre la pelota y la portería
comportamientos	[MACRO, setDisplayString]

Cancelar Aceptar

Figura 4.2: Resultados de la búsqueda de un portero

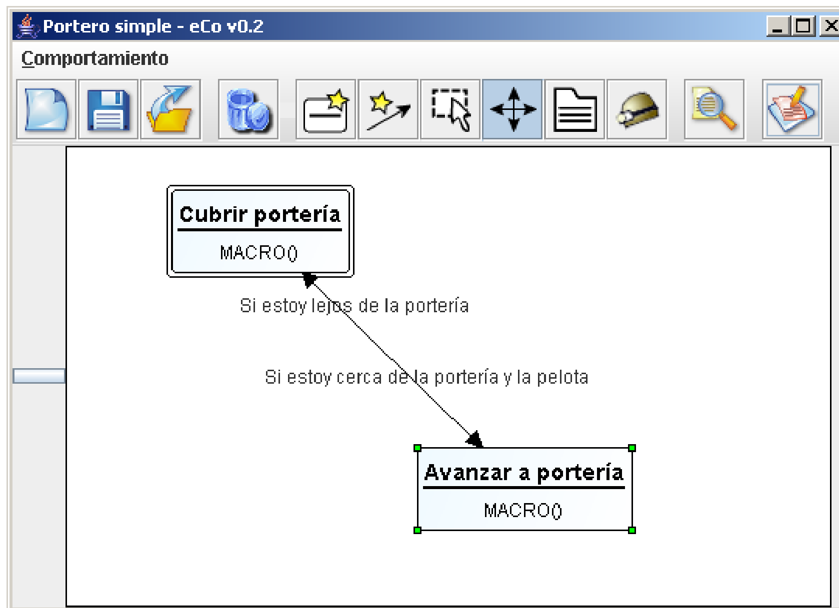


Figura 4.3: Adaptación de los comportamientos recuperados

deseado. En la figura 4.2 se muestran los resultados de esta consulta.

Los dos comportamientos de mayor similitud que aparecen en ella, `Cubrir portería` y `Avanzar a portería`, cumplen parcialmente los requisitos de la consulta. El primero da la funcionalidad de un portero, cubriendo la portería, y el segundo se puede utilizar para hacer que el jugador vuelva a la portería cuando se aleje demasiado. Ambos parecen buenos candidatos para realizar la adaptación.

Como se mencionó en la sección 3.2.1, la adaptación se realiza manualmente. En este caso, se recuperan los dos comportamientos dentro del comportamiento actual y se unen con dos aristas. El resultado se muestra en la figura 4.3. Cuando el jugador se encuentra lejos de la portería avanza hacia ella, y cuando se encuentra cerca de la portería y de la pelota, trata de cubrirla.

Como hemos visto, el tamaño adecuado para la base de casos viene influenciado por la cantidad de casos necesarios para tener una muestra representativa de los comportamientos que se pueden generar para el juego, que, a su vez, depende de la complejidad de modelo de juego. En el caso de Soccerbots, los resultados obtenidos con una base de casos de alrededor de 25 entradas son bastante buenos.

En la mayoría de sistemas basados en conocimiento, la fase de adquisición del conocimiento suele ser, en mayor o menor medida, un cuello de botella. En el caso del editor de comportamientos, parte de la adquisición del conocimiento se realiza automáticamente; en concreto, la generación del diccionario de términos incluidos en las descripciones y los subcomportamientos. Otra parte debe ser realizada por el usuario antes de añadir los comportamientos a la base de casos. Su tarea es asignar valores a cada una de las propiedades que describen el comportamiento, de manera que la descripción sea adecuada al contenido del caso. Este paso es fundamental para el buen funcionamiento del CBR, pues si un caso no está descrito adecuadamente no será recuperado en las consultas para las que es adecuado o, por el contrario, será recuperado como respuesta a consultas para las que no es idóneo.

En estas circunstancias, el concepto de descripción adecuada es bastante relativo, y puede ser diferente para cada persona. Por ejemplo, en el caso de Soccerbots, un comportamiento que mantenga a jugador en su lado del campo, cerca de la portería, cubriéndola, será considerado como altamente defensivo, pero también puede considerarse en cierto grado como portero. De la misma forma, las descripciones textuales pueden ser ambiguas o dar lugar a confusión. Una posible solución a este problema es tratar de obtener estos datos directamente de los comportamientos. Esta es una de las líneas de investigación abiertas, como se explica en el capítulo 5.

4.2. Integración con Soccerbots

Soccerbots es un entorno de simulación que simula la dinámica y las dimensiones de un partido reglamentario de la Robocup, la copa mundial de fútbol para robots de pequeño tamaño. Las reglas son sencillas: dos equipos, formados por cinco robots, compiten en un campo de juego del tamaño de una mesa de ping-pong, delimitado por una pared. El objetivo es introducir una pelota de golf en la portería del contrario. Para ello, los robots pueden empujar la pelota o darle una "patada", un empujón más fuerte que la lanza a través del campo. La pelota se mantiene en juego durante 60 segundos. Si transcurrido ese tiempo no se ha metido un gol, se devuelve al centro del campo. Cuando alguno de los equipos mete un gol, tanto la pelota como los equipos vuelven a sus posiciones iniciales antes de reanudarse el juego.

En la figura 4.4 se muestra un esquema del campo con las dimensiones del mismo.

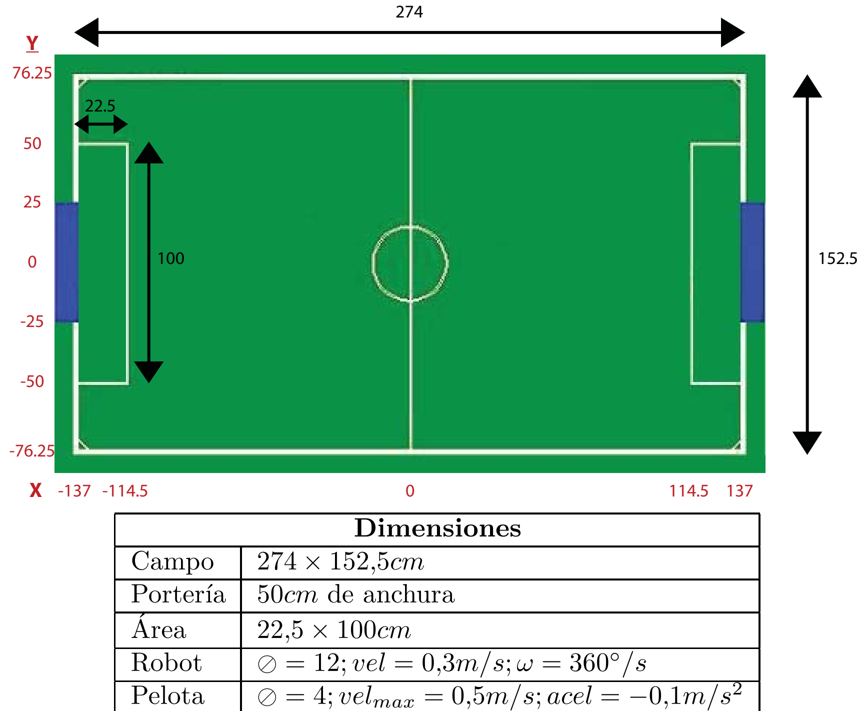
Para poder utilizar el editor de comportamientos con Soccerbots, es necesario definir el modelo de juego y crear, al menos, un generador de código.

El modelo de juego está formado por dos partes: la interfaz de juego, en la que se enumera el conjunto de sensores y actuadores que forman la interfaz de comunicación entre el simulador y la implementación del comportamiento, y las propiedades utilizadas para describir la funcionalidad de los comportamientos en la base de casos.

El conjunto de sensores y actuadores que forman la interfaz de juego se obtiene de la especificación de Soccerbots[2]. A continuación se describe la interfaz de juego implementada para Soccerbots:

- Sensores:
  - Naturales:
    - **double getBallX, getBallY, getBallR, getBallT**: obtener las componentes de un vector que apunta desde el jugador hacia la pelota. getBallX

[2] http://www.cs.cmu.edu/ trb/TeamBots/Docs/EDU/gatech/cc/is/abstractrobot/SocSmall.html



Dimensiones	
Campo	$274 \times 152{,}5cm$
Portería	$50cm$ de anchura
Área	$22{,}5 \times 100cm$
Robot	$\oslash = 12; vel = 0{,}3m/s; \omega = 360°/s$
Pelota	$\oslash = 4; vel_{max} = 0{,}5m/s; acel = -0{,}1m/s^2$

Figura 4.4: Dimensiones del terreno de juego de Soccerbots

y getBallY devuelve las componentes cartesianas del vector, mientras que getBallR y getBallT devuelve el módulo y el ángulo.

- **boolean getJustScored**: indica si se ha producido un evento de gol. (1 si es un gol a favor, -1 en contra y 0 en otro caso).
- **double getOpponentsGoalX, getOpponentsGoalY, getOpponentsGoalR, getOpponentsGoalT**: obtener las componentes de un vector que apunta desde el jugador hacia la portería contraria.
- **double getOurGoalX, getOurGoalY, getOurGoalR, getOurGoalT**: obtener las componentes de un vector que apunta desde el jugador hacia la propia portería.
- **boolean canKick**: indica si el jugador está a una distancia adecuada para chutar la pelota.
- **int getPlayerNumber**: obtener el número de jugador.
- **int getID**: obtener el identificador único del robot.
- **double getPositionX, getPositionY, getPositionR, getPositionT**: obtener la posición del robot en coordenadas absolutas.
- **double getSteerHeading**: obtener la dirección del robot.
- **long getTime**: obtener el tiempo transcurrido desde que se instanció el robot (timestamp).
- **double KICKER_SPEED**: velocidad a la que se chuta el balón.
- **double KICKER_SPOT_RADIUS**: distancia de la pelota para que se pueda chutar.

- **double MAX_STEER**: máxima velocidad de giro de un robot.
- **double MAX_TRANSLATION**: máxima velocidad de movimiento de un robot.
- **double RADIUS**: radio del robot.

- Sintéticos:
  - **int CAMPO**: valor que indica en qué campo tiene que marcar nuestro equipo. Vale 1 si tiene que marcar en el campo de la derecha y -1 si marca a la izquierda.
  - **double PosicionNormalizadaX**: posición del jugador en coordenadas absolutas suponiendo que el eje X del campo se hace positivo hacia la portería contraria. Es el resultado de multiplicar $CAMPO \cdot getPositionX$.
  - **double PelotaX, PelotaY**: Devuelve las componentes X e Y del vector de posición de la pelota en coordenadas globales.
  - **double PelotaNormalizadaX**: posición de la pelota en coordenadas globales suponiendo que el eje X del campo se hace positivo hacia la portería contraria. Es el resultado de multiplicar $CAMPO \cdot PelotaX$.
  - **int DiferenciaGoles**: goles a favor menos goles en contra.
  - **boolean MasCercaPelota**: devuelve el valor `true` si el jugador se encuentra más cerca de la pelota que los demás.
  - **boolean DistanciaPelota<DistanciaPorteria**: es verdadero si el jugador se encuentra más cerca de la pelota que de la portería.

- Actuadores:
  - Naturales:
    - **kick**: da una patada a la pelota si se encuentra a la distancia apropiada.
    - **setKinMaxRange(double)**: establecer la distancia a la que se pueden percibir los restantes jugadores.
    - **setDisplayString(string)**: establecer el mensaje de estado del robot.
    - **setID(int)**: establecer el identificador del robot.
    - **setObstacleMaxRange(double)**: establecer la distancia máxima a la que se detectan los obstáculos.
    - **setSpeed(double)**: establecer la velocidad del robot (0 - 1).
    - **setSteerHeading(double)**: establecer el ángulo (en radianes) hacia el que mira el robot.
  - Sinténticos
    - **colocarseDetras(string)**: Modifica la dirección hacia la que mira el jugador y su velocidad para colocarse así detrás de la pelota, apuntando a la posición indicada por el parámetro. Los parámetros posibles son "porteria_contra" y "porteria_propia".
    - **bloquear(string)**: Bloquea a un jugador del equipo contrario para evitar que se mueva. El jugador bloqueado se puede indicar mediante un parámetro, que puede ser: "portero" (bloquea al jugador más atrasado), "delantero" (bloquea al jugador más adelantado) o "jugador" (bloquea al jugador más cercano).

Para describir los comportamientos se han escogidos cuatro propiedades:

- **Movilidad [0, 5]**: mediante esta propiedad se mide la capacidad del robot para moverse por todo el campo. Un comportamiento que hace que el robot se mantenga en una posición fija, por ejemplo un portero que se mantiene cerca de la portería, tendrá un valor bajo en esta propiedad.
- **Ataque [0, 5]**: mide la capacidad del robot para jugar al ataque. Si el robot puede jugar de delantero tendrá un valor alto en esta propiedad.
- **Defensa [0, 5]**: esta propiedad mide la capacidad defensiva del robot.
- **Portero [0, 1]**: un valor de 1 en esta propiedad indica que el robot juega cubriendo la portería.

El generador de código debe generar código Java y utilizará los sensores y actuadores definidos anteriormente.

El generador desarrollado utiliza una implementación de las máquinas de estado finito basada en tablas. En esta implementación existe una tabla que almacena, para cada estado, una estructura con la siguiente información:

- Un puntero a la función que ejecuta la acción asociada al nodo.
- Una lista con las aristas que salen del nodo. Por cada transición se almacena:
  - Un puntero a la función que evalúa la condición asociada a la arista.
  - El nuevo estado al que se llega cuando la condición se hace cierta.

En Java no se permite almacenar punteros a funciones directamente, pero existen métodos para sortear este problema. En este caso, se ha optado por almacenar el nombre de la función en lugar del puntero. Para invocarla, se utilizan los métodos y las clases del paquete `java.lang.reflect`. Otra posibilidad sería definir dos interfaces, uno para las condiciones y otro para las acciones, de manera que se almacene una referencia a una clase que implemente la interfaz adecuada.

Para implementar la jerarquía en las máquinas de estado se incluye una pila, que contendrá, en cada instante de tiempo, el nodo actual de cada uno de los comportamientos y subcomportamientos activos. Además de las operaciones habituales en una pila (añadir y extraer objetos de la cima), esta estructura debe permitir acceder a todos los elementos contenidos en ella. De esta manera se pueden evaluar las condiciones de todos los nodos activos.

En el listado 4.1 se muestra el código correspondiente a la ejecución de un comportamiento.

En el método `configure` se inicializa el sistema, generando la tabla de estados e introduciendo en la pila el estado inicial. Durante la generación de la tabla de estados, además de crear esta estructura, se generan los métodos de condiciones y acciones. Para implementar la jerarquía de estados, la acción que realizan los nodos que contienen comportamientos subordinados es añadir a la pila el nodo subordinado.

El método `takeStep` es llamado por el entorno de simulación cada iteración del bucle de simulación. En este método se ejecuta, en primer lugar, la acción asociada al nodo actual, que es el que se encuentra en la cima de la pila. A continuación se evalúan todas las condiciones asociadas a las transiciones de los nodos de la pila. Esta evaluación comienza por la parte más baja de la pila y acaba por la cima. De esta manera, se da mayor precedencia a los comportamientos que se encuentran a un nivel jerárquico más alto. Si alguna de las condiciones se evalúa a cierto, se sacan de la pila todos los estados que se encuentran por encima de ella, y se añade el estado de destino de la transición.

```
// Inicialización
public void configure() {
  // Generación de la tabla de estados y las funciones
     asociadas
  tabla = generarTabla();
  pila = new Vector<Estado>();
  // Añadir el estado inicial
  pila.add(tabla.get(0));
}

// Ejecución del comportamiento
public int takeStep() {
  // Ejecución de la acción del nodo
  this.getClass().getMethod(pila.lastElement().acciones,
     null).invoke(this, null);
  int posPila = 0;
  while (posPila < pila.size()) {
    Estado e = pila.get(posPila);
    if (e.transiciones != null) {
      // Evaluación de las condiciones
      for (int j = 0; j < e.transiciones.length; j++) {
        if ((Boolean)
            this.getClass().getMethod(e.transiciones[j].condicion,
            null).invoke(this, null)) {
          // Eliminar de la pila los estados que están por
             encima
          for (int k = pila.size(); k >= posPila; --k) {
            pila.remove(k);
          }
          // Añadir nuevo estado actual
          pila.add(tabla.get(e.transiciones[j].estado));
        }
      }
    }
  }
}
```

Listado 4.1: Ejecución de un comportamiento

El no determinismo se resuelve en el bucle de evaluación de las condiciones. En este caso, se toma la primera arista cuya condición se evalúa a cierto. Otra posible aproximación sería almacenar en una estructura temporal todas las condiciones que se evalúen a cierto, y escoger una de ellas aleatoriamente.

4.3. Integración con JV^{2}M

El desarrollo de JV^{2}M (ver sección 2.4.1) está aún en marcha y en la fase actual no existe un modelo de sensores y actuadores para la comunicación entre los comportamientos externos y el juego.

El desarrollo completo de este sistema de comunicación es una tarea laboriosa y en absoluto sencilla, y, aunque está relacionada, queda fuera del alcance de este trabajo. No obstante es una tarea que deberá realizarse antes o después. Se incluye a continuación, como punto de partida, un pequeño esbozo de un sistema de sensores y actuadores inicial.

Para obtener el modelo propuesto se ha partido del modelo utilizado por *Far Cry*[3]. Dos han sido los factores principales que nos han impulsado ha tomar esta decisión: por un lado, el género de ambos juegos (Far Cry y JV^{2}M) es bastante similar; además, Far Cry está construido de manera que permite personalizar gran parte de sus elementos, por ejemplo, permite describir comportamientos nuevos para ser utilizados por su inteligencia artificial. Por otro lado, existe gran cantidad de documentación disponible sobre este tema.

4.3.1. El modelo de Far Cry

La construcción de comportamientos en Far Cry se basa en la utilización de `scripts` *Lua*[4].

Los comportamientos se definen como una secuencia de instrucciones en respuesta a determinados eventos. Los eventos se organizan en dos niveles jerárquicos:

- Eventos de juego: cualquier cosa que pueda ocurrir durante el juego es un evento de juego.
- Eventos tácticos: agrupan eventos de juego. Una secuencia de eventos de juego, combinada opcionalmente con determinadas propiedades del entorno, puede generar un evento táctico.

Una de las características más importantes de Far Cry es que, mediante los `scripts` *Lua*, se pueden personalizar prácticamente todos los elementos que aparecen en el juego o añadir nuevos elementos. Los eventos no se utilizan únicamente como elementos dentro de los comportamientos inteligentes de los personajes del juego, sino que participan en esta personalización. Este hecho hace que el número de eventos sea muy elevado y que muchos de ellos sean propios de un sólo elemento o grupo de elementos y no aplicables a los restantes. Por ejemplo, los elementos de tipo puerta tienen un evento `Opened` que indica que la puerta está completamente abierta.

Otro elemento importante es que la construcción de comportamientos se apoya en una serie de subsistemas capaces de transformar información de bajo nivel en datos más complejos y abstractos. Los dos más importantes son el modelo sensorial, utilizado principalmente para simular los sentidos de vista y oído de los personajes, y la búsqueda de caminos. Esta separación cobra mayor sentido en la aproximación seguida por el editor de comportamientos, ya que este tipo de tareas, que pueden ser realizadas eficientemente mediante algoritmos, son bastante difíciles de implementar utilizando máquinas de estado.

A grandes rasgos, la búsqueda de caminos se realiza en tres fases. En primer lugar, se divide el terreno mediante un algoritmo de triangulación, creando las zonas prohibidas

---

[3] Far Cry es un juego de acción en primera persona. Se puede encontrar más información en la siguiente dirección: http://www.ubi.com/UK/Games/Info.aspx?pId=547

[4] Lua es un lenguaje de `scripts` imperativo, procedural y reflexivo que ofrece bastante flexibilidad. Puede encontrarse más información en `http://www.lua.org`.

(zonas en las que no pueden entrar los personajes) y las conexiones entre zonas (espacios por donde se puede pasar de una zona a otra). Con esta información se genera un grafo, en el que los nodos representan los triángulos obtenidos en el paso anterior, y las aristas representan los lados comunes entre dos triángulos adyacentes. En cada arista se incluye la información sobre las conexiones entre zonas. Los dos pasos iniciales se realizan cuando se diseña el mapa del nivel. El último paso se produce en tiempo de ejecución, y consiste en la búsqueda, si existe, de un camino que conecte dos nodos concretos dentro del grafo anterior. Para realizar esta búsqueda se utiliza el algoritmo A* [43].

El modelo sensorial gestiona, básicamente, el sentido de la vista de los personajes y, en algunos casos el oído. Para llevar a cabo esta simulación, en cada ciclo de juego y por cada personaje, el sistema realiza una comprobación de visibilidad, desde el punto de vista del personaje contra todos los objetivos potenciales. La comprobación de visibilidad se realiza en tres pasos:

1. Comprueba si el objetivo está dentro del rango de visión del personaje.

2. Si se supera la prueba anterior, se comprueba si el objetivo está dentro del FOV (campo de visión, ángulo que determina la amplitud del cono de visión).

3. La última comprobación, y la más costosa, consiste en un trazado de rayos desde el personaje hasta el objetivo, para verificar que no existe ningún obstáculo físico entre ellos.

Si se supera esta comprobación, se añade el objetivo a una lista de visibilidad. Cuando se pierde la visibilidad de un objetivo, sale de la lista de visibilidad y entra en una lista de objetivos en memoria. De esta manera, aunque se pierda de vista al objetivo, es posible perseguirlo. Dado que el proceso de comprobación de visibilidad es bastante costoso y que se realiza en cada actualización, se introducen algunas restricciones sobre los objetivos potenciales del mismo. Para que un objeto sea candidato a ser visible, debe estar activo y además debe ser de una especie diferente al personaje que realiza la visión. La especie es una propiedad de los personajes para identificar los distintos posibles bandos existentes en el juego.

Con lo dicho hasta ahora, el concepto de visibilidad consistiría en asignar un valor de cierto o falso a cada objetivo potencial. Para conseguir mayor realismo, se aplica lo que se denomina el coeficiente de visibilidad. Al añadir un objetivo a la lista de visibilidad, se añade también un valor de percepción que describe la certeza que tiene el personaje de haber visto a su objetivo. Cuando este coeficiente alcanza un valor máximo (10 en la implementación final de Far Cry) se considera que el personaje ha visto a ese objetivo. Cada personaje en el juego tiene un coeficiente de visibilidad independiente para cada objetivo que procesa. La velocidad a la que aumenta el coeficiente de visibilidad depende de varios factores, como la distancia, la altura desde el suelo (para distinguir si el objetivos está de pie, agachado o tumbado), si se está moviendo o determinados modificadores artificiales. Por otra parte, si se pierde la visibilidad, el coeficiente irá descendiendo en cada actualización hasta llegar a cero. En la implementación de Far Cry, el coeficiente de visibilidad sólo se aplica cuando el objetivo potencial es el jugador. En los demás casos, se considera que el objetivo es visible si supera la comprobación de visibilidad.

4.3.2. El modelo de JV^{2}M

Utilizando el modelo propuesto en Far Cry como base, se propone construir un modelo similar para utilizar en JV^{2}M.

Constante	Descripción
ULTIMA_OP	Objeto de la última operación
CENTRO_AT	Centro de atención actual
GRUPO_RANGO	Las entidades de mi grupo en un rango
GRUPO_NIVEL	Las entidades de mi grupo en todo el nivel
ESPECIE_RANGO	Las entidades de mi especie en un rango
ESPECIE_NIVEL	Las entidades de mi especie en todo el nivel
TODO_RANGO	Todas las entidades en un rango
TODO_NIVEL	Todas las entidades del nivel
JUGADOR	El jugador
<Objeto>	Localiza un objeto por su nombre

Tabla 4.2: Filtros de objetivos

Constante	Descripción
MAS_CERCANO	Elemento más cercano
MAS_CERCANO_ATENCION	Elemento más cercano al centro de atención actual
MAS_LEJANO_ATENCION	Elemento más lejano del centro de atención actual
MAS_CERCANO_OPERACION	Elemento más cercano al objeto de la última operación realizada
MAS_LEJANO_OPERACION	Elemento más lejano al objeto de la última operación realizada
IZQUIERDA_ATENCION	Elemento más a la izquierda del centro de atención actual
DERECHA_ATENCION	Elemento más a la derecha del centro de atención actual

Tabla 4.3: Búsqueda de objetivos

La construcción de este modelo será un proceso iterativo, que comenzará con la definición de un conjunto pequeño de sensores y actuadores. Este conjunto deberá ser integrado en el juego, junto con los subsistemas necesarios para hacerlos funcionar, y deberá ser probado implementando diferentes comportamientos. Por último, se eliminarán los sensores y actuadores que se consideren innecesarios para la edición de comportamientos, o cuya implementación sea contraproducente para el funcionamiento de la aplicación (por ejemplo, si ralentizan demasiado la velocidad del juego), y se añadirán nuevos sensores o actuadores que se hayan echado en falta para la edición de los comportamientos. Este ciclo se sucede hasta encontrar un conjunto de sensores y actuadores estable y lo bastante expresivo como para implementar los comportamientos necesarios.

En la elaboración de esta lista de sensores y actuadores se parte de la suposición de que existen un sistema para el cálculo de caminos y otro para calcular la visibilidad. Además, por cada entidad del juego se mantiene un conjunto de variables cuyos valores se conservan entre diferentes iteraciones del bucle de juego. Estas variables son:

- El camino actual: el cálculo de caminos es un proceso costoso, por lo que se realiza en dos fases. En la primera fase se obtiene el camino que se va a seguir. Esta fase podría ser asíncrona para no ralentizar el desarrollo del juego. Ese camino se almacena en esta variable y luego puede ser recorrido por el personaje.

- La lista de visibilidad: es una lista en la que se mantienen todas las entidades que pueden ser vistas por el personaje.

- Memoria de visibilidad: en esta lista se incluyen las entidades que no pueden ser vistas pero que lo han sido recientemente.

- Objeto de la última operación: en esta variable se almacena el elemento que ha sido objetivo de la última operación realizada.
- Centro de atención: en esta variable se almacena una entidad que sea de especial interés para el personaje.

La lista de sensores y actuadores es la siguiente:

- Sensores:
  - **double GetRangoDisparo**: obtiene la distancia máxima a la que puede disparar el personaje.
  - **boolean GetMensajesEnCola**: devuelve cierto si existe algún mensaje en la cola de mensajes pendientes.
  - **string GetMensaje**: devuelve el contenido del primer mensaje de la cola de mensajes. Si la cola está vacía devuelve un mensaje nulo. Este mensaje es diferente a la cadena vacía.
  - **GetPosicion**: devuelve la posición en coordenadas cartesianas del personaje.
  - **GetDistancia(objetivo)**: devuelve la distancia a un determinado objetivo, que será indicado por una constante de las que aparecen en la tabla 4.2. Si la constante se refiere a un grupo de objetivos, se obtiene la distancia al más cercano del grupo.
  - **double GetSalud**: devuelve la cantidad de vida del personaje.
  - **boolean GetDaño(entero)**: es verdadero si se ha recibido daño durante el tiempo indicado por el parámetro. Si el valor de tiempo es -1, devuelve verdadero si se ha recibido daño desde la última llamada a la función.
  - **string GetCentroDeAtencion**: devuelve el centro de atención del personaje.
  - **string GetUltimaOperacion**: devuelve el objetivo de la última operación realizada.
  - **boolean EsVisible(string)**: devuelve verdadero si el elemento está en la lista de visibilidad.
  - **boolean EsVisibleClase(string)**: devuelve verdadero si es visible algún elemento de la clase indicada por el parámetro. Las clases describen tipos de elementos. Un ejemplo de distintas clases puede ser: JUGADOR, GRANADA, EXPLORADOR, BOTIQUIN, MUNICION, etc.
  - **boolean EsVisibleEnemigo**: devuelve verdadero si es visible alguna entidad enemiga, es decir, cuya especie sea diferente a la del personaje.
  - **boolean EstaEnMemoria(string)**: devuelve verdadero si el elemento está en la memoria, es decir, si ha sido visto recientemente.
  - **boolean EsVisibleCentroDeAtencion**: es verdadero si el centro de atención es visible. Si el personaje no tiene centro de atención, o si este no es visible, devolverá falso. Equivale a realizar la llamada `EsVisible(GetCentroDeAtencion)`.
- Actuadores:
  - **SetCentroAtencion(objetivo)**: cambia el centro de atención actual por el indicado mediante el parámetro. El parámetro es un filtro de objetivos (tabla 4.2). Si se refiere a un grupo de elementos, se escogerá uno entre ellos. Sería conveniente que el método para escogerlo fuera aleatorio. Es posible que fuera más conveniente que este actuador sólo funcionara cuando el objetivo sea visible.

- **Localizar(objetivo)**: actualiza el valor del resultado de la última operación al valor indicado por el parámetro objetivo. Este valor debe ser una constante de las indicadas en la tabla 4.2.
- **Acercarse, Alejarse(double)**: se aleja o se acerca del blanco de atención actual una cierta distancia, indicada mediante un parámetro.
- **SetPostura(string)**: cambia la postura del personaje. El cambio en la postura no sólo influye en la representación visual, sino que también afecta a la velocidad de movimiento, la animación empleada para desplazarse o si puede correr.
- **Dispara(boolean)**: indica al personaje que puede disparar. No significa que realice un disparo en el momento en que se activa el actuador, sino que, si tiene algún enemigo a la vista y dentro del rango de disparo, le disparará. El rango de disparo vendrá dado por la propiedad RangoDisparo, aunque está limitado por el alcance máximo del arma activa del personaje.
- **Corre(boolean)**: indica al personaje que los movimientos que haga a partir de entonces debe hacerlos corriendo.
- **Salta(destino, distancia)**: salta detrás de un objeto para cubrirse. El parámetro destino indica la zona donde el personaje busca cobertura (tabla 4.3), y la distancia, el rango máximo de la búsqueda.
- **SetTemporizador(nombre, duración)**: inicializa un temporizador asignándole un nombre y una duración en segundos. Si el nombre ya existía, lo reinicia con la nueva duración.
- **TrazarCamino(objetivo)**: comienza el proceso de generación de un camino desde la posición actual hasta el objetivo indicado mediante el parámetro (tabla 4.2). Este actuador genera el camino, pero no se recorre hasta invocar al actuador `Recorrer`, ya que la generación del camino puede tomar cierto tiempo.
- **Recorrer(boolean)**: recorre el último camino generado por una llamada del personaje a TrazarCamino. Una vez que el agente ha alcanzado el final del camino, puede esperar allí o recorrerlo en sentido contrario, dependiendo del valor del parámetro booleano.
- **Enviar(string, objetivo)**: envía un mensaje a uno o varios objetivos. El mensaje es una cadena de texto y los objetivos se seleccionan mediante las constantes de la tabla 4.2. Si alguno de los objetivos están dentro de la distancia de comunicación, el mensaje entra en una cola de mensajes en los destinatarios, activando el sensor GetMensajesEnCola. Para recuperar los mensajes de la cola se utiliza el sensor GetMensaje.
- **MovimientoLateral(distancia)**: mueve al personaje alrededor de su centro de atención. Si la distancia indicada es positiva se moverá hacia la derecha, mientras que si es negativa lo hará hacia la izquierda.
- **MirarA(ángulo)**: cambia la dirección hacia la que mira el personaje.
- **Cubrirse(destino, distancia)**: busca cobertura detrás de un elemento. El parámetro destino indica la zona donde el personaje busca cobertura (tabla 4.3), y la distancia, el rango máximo de la búsqueda. Este es un actuador de alto nivel, que deberá apoyarse en los subsistemas antes mencionados.
- **SetRangoVisión(distancia)**: especifica un nuevo rango de visión.
- **SetFOV(ángulo)**: especifica un nuevo ángulo para el campo de visión.

- **MostrarConsola(string)**: muestra una cadena en la consola. Se usa principalmente para depuración de comportamientos.
- **SetRangoDisparo(double)**: establece la distancia máxima a partir de la cual el personaje puede comenzar a disparar. Si esta distancia es mayor que el alcance del arma, el personaje disparará aunque no alcance a su objetivo. Puede ser útil para realizar disparos de aviso o para intimidar a los enemigos.
- **SetRangoComunicación()**: establece la distancia máxima a la que el personaje puede enviar mensajes a otras entidades. Esta distancia se utiliza, además, como rango para los filtros `GRUPO_RANGO`, `ESPECIE_RANGO` y `TODO_RANGO`.
- **Olvidar**: olvida los objetivos que están en la memoria del personaje.
- **Limpiar**: devuelve al personaje a un estado inicial, con la memoria, el centro de atención, la lista de visibilidad y la última operación vacíos.
- **PuedeVer(boolean)**: especifica si el personaje puede ver. Si no puede ver, no es necesario realizar las comprobaciones de visibilidad, lo que puede ahorrar tiempo de proceso.
- **Invisible(boolean)**: si el parámetro es verdadero, el personaje no participará en las comprobaciones de visibilidad del resto de entidades, siendo, a todos los efectos, invisible para ellas. No obstante, seguirá mostrándose en pantalla.
- **SetEspecie(string)**: asigna un nombre de especie al personaje. Los individuos de especies diferentes se ven como enemigos.

Otros sensores y actuadores que pueden ser añadidos en futuras iteraciones incluyen aquellos referentes al tratamiento de sonidos, tanto la capacidad auditiva de los personajes como la posibilidad de que determinadas acciones puedan asociarse a la reproducción de un sonido, la posibilidad de que los personajes se agrupen formando unidades organizadas, con algún tipo de formación, y el cálculo de algunas estadísticas referentes a la partida en curso que puedan ser utilizadas para ajustar la dificultad del juego mientras se está desarrollando.

Para definir por completo la interfaz de juego, es necesario especificar el conjunto de propiedades que se utilizarán para describir los comportamientos. A continuación describiremos informalmente algunos comportamientos representativos, que suelen darse en este tipo de juegos, para obtener de ellos una lista de propiedades capaz de describirlos a todos:

- Soldado: es un comportamiento muy genérico. Suele estar guardando una posición o haciendo una ronda entre varios puntos. Dispara al ver a su enemigo. Suele llevar armas ligeras.

- Explorador: es bastante rápido y silencioso. Se mueve dentro de una zona sin un destino fijo hasta que detecta a un enemigo. Sus capacidades sensoriales son más altas que lo habitual. Lleva armas ligeras y prefiere flanquear al enemigo antes que un ataque directo.

- Francotirador: se mantiene en una posición fija, desde la que ataca a sus enemigos con un arma de larga distancia.

- Bruto: personaje con una condición física por encima de lo normal. Resiste mejor el daño enemigo. Tiene preferencia por el combate cuerpo a cuerpo.

- Demoledor: tiene preferencia por las armas pesadas (granadas, lanzacohetes, etc.) e intenta guardar una distancia prudencial con sus enemigos.

- Suicida: cuando ve a un enemigo, corre hacia él atacándole con el arma más pesada que tenga. No le importa recibir daño del enemigo o de sus propias armas.

A partir de estas descripciones se elabora la siguiente lista:

- Movilidad: indica si el personaje tiende a moverse o mantiene fija su posición.
- Sigilo: indica si el personaje es sigiloso o utiliza algún medio para no ser detectado (camuflaje, caminar agachado, etc.).
- Agresividad: un valor alto en esta propiedad indica que el personaje ataca a sus enemigos en cuanto los descubre. Valores más bajos pueden indicar que ataca cuando es provocado o, incluso, que no ataca, sino que huye de sus enemigos.
- Protección: indica que el personaje tiene algún tipo de protección que hace que resista mejor el daño, o que utiliza su entorno para protegerse.
- Autoestima: indica el grado en que el personaje cuida de su salud. Un comportamiento suicida tendrá una autoestima muy baja, mientras que un comportamiento en el que el personaje busque un botiquín para curarse cuando esté herido tendrá una autoestima alta.
- Potencia de fuego: indica el tipo de arma por el que se tiene preferencia en el comportamiento. Si se utilizan armas pesadas, este valor deberá ser alto.
- Comunicatividad: indica si el personaje puede comunicarse de alguna forma con el jugador.

De la misma forma que la lista de sensores y actuadores, en futuras iteraciones esta lista puede ser objeto de cambios para ajustar los parámetros y obtener una descripción más adecuada de los comportamientos.

4.4. Integración con Neverwinter Nights

Neverwinter Nights es un videojuego de rol basado en el universo de Dungeons & Dragons, un juego de rol de mesa que se encuadra dentro del subgénero de fantasía medieval. Antes de exponer las ideas sobre la integración del juego con el editor de comportamientos, haremos una breve descripción del mismo, con objeto de dar una idea de la gran cantidad de parámetros que afectan a la toma de decisiones por parte de la inteligencia artificial.

En Neverwinter Nights (NWN) el jugador controla a un personaje que habita un entorno poblado por otros personajes no jugadores y criaturas que le encomiendan misiones que el jugador debe resolver.

Como en la mayoría de juegos de rol, las características y conocimientos del personaje vienen definidas por una serie de atributos que determinan sus habilidades y su competencia en ellas. Es lo que se denomina la ficha del personaje.

Cuando el personaje derrota a un enemigo o realiza una misión, recibe una recompensa en forma de oro, objetos que puede utilizar, como espadas, armaduras o hechizos, y puntos de experiencia. Según gana experiencia, el personaje puede subir de nivel, y mejorar así sus habilidades o adquirir otras nuevas.

Otra característica común a la mayoría de juegos de rol son las clases. Al crear un nuevo personaje, el jugador debe elegir la clase a la que pertenece (guerrero, mago, ladrón, druida, explorador...). La clase puede determinar el conjunto de habilidades básicas que posee y restricciones sobre otras habilidades o características.

En NWN existen once clases y siete razas diferentes. Cada combinación de raza y clase tiene una serie de modificadores y restricciones sobre las habilidades del personaje.

Existen seis habilidades principales (fuerza, destreza, constitución, inteligencia, sabiduría y carisma), comunes a todos los personajes, cuyos valores definen cómo interactúa el personaje con el mundo que le rodea. Por ejemplo, un personaje con fuerza 20 (un valor alto) es capaz de realizar tareas que requieran mucha fuerza, como mover un carro con sus propias manos; de la misma forma, si tiene inteligencia 3, no será capaz de realizar tareas que requieran el uso de la inteligencia, como realizar conjuros mágicos.

A estas habilidades principales hay que añadir un conjunto de habilidades secundarias o aptitudes, que el jugador debe elegir de entre las 23 aptitudes predefinidas por el juego. También se pueden considerar como pertenecientes a este grupo las capacidades de algunas clases para invocar hechizos mágicos. El conjunto de habilidades secundarias está influenciado por la clase del personaje y puede ir creciendo y mejorando al subir de nivel.

La última característica que define cómo interactúa el personaje con su entorno, común a la mayoría de juegos de rol, es el equipo. El equipo es el conjunto de objetos que el personaje lleva puestos (ropas, armas, armadura, etc.) o que transporta en su inventario. Los objetos equipados pueden modificar los valores de las habilidades y las aptitudes.

La distribución comercial de NWN incluye, además, el Aurora Neverwinter Toolset, un conjunto de herramientas integradas para la edición de mundos, campañas, personajes, etc. Este editor permite, en concreto, especificar comportamientos para los personajes no jugadores[5] mediante un lenguaje de `scripts`.

El lenguaje de `scripts` de NWN es muy amplio y permite acceder a gran cantidad de propiedades, no solo de los personajes que habitan el mundo de juego, sino también de los objetos y del propio entorno, como el clima o la hora (día o noche) en que se desarrolla la acción.

Los comportamientos en NWN están basados en eventos. Cada personaje tiene un conjunto de eventos que se disparan al realizar o recibir determinadas acciones (al ser atacado o al recibir un hechizo). Mediante el editor Aurora pueden asociarse con `scripts`, de manera que se ejecuten al dispararse el evento.

Una de las posibilidades estudiadas para la integración es utilizar RCEI.

RCEI (Remote Controlled Environments Interface) es un protocolo y un lenguaje de comunicación entre aplicaciones ideados para conectar de manera genérica, un entorno virtual, de tipo narrativo, con una aplicación de control remoto, manual o automática [41].

El protocolo RCEI se ha implementado en sockets ASCII. La comunicación es síncrona bloqueante. La aplicación remota es quien comienza la comunicación, que se desarrolla de una forma estrictamente alterna, y la que la finaliza, mediante un mensaje de finalización.

El significado de los mensajes depende de su emisor. Si el mensaje es enviado desde la aplicación de control hacia el entorno virtual, es una orden que debe realizar el entorno virtual. Si es enviado en sentido opuesto, se considera que es la comunicación de un hecho que ha tenido lugar en el entorno virtual.

En la siguiente lista se enumeran los distintos tipos de mensajes incluidos en RCEI:

- Inicio de la comunicación.
- Identificación del entorno virtual al que se está conectado.
- Finalización de la comunicación.
- Valores de fecha y hora del entorno virtual.

[5]Los personajes no jugadores o PNJs son otros personajes participantes en el juego, manejados por la inteligencia artificial.

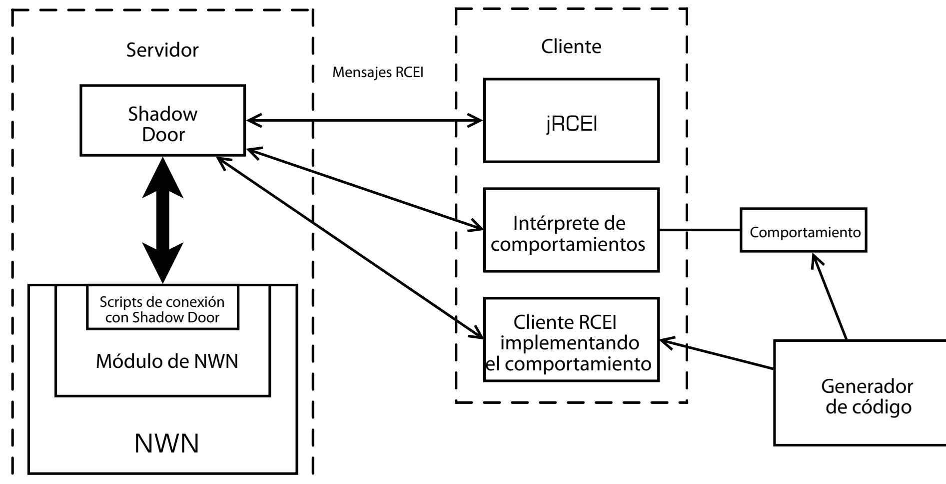


Figura 4.5: RCEI en Neverwinter Nights

- Valores para las condiciones climáticas y ambientales del entorno virtual.
- Valores sobre el estado de los diferentes objetos presentes en el entorno virtual (puertas, fuentes de luz, etc.).
- Transición entre localizaciones.
- Creación de una posición en una localización.
- Creación de agentes y objetos en una posición de una localización.
- Valores de posición de objetos dentro de una localización de un agente u objeto (por ejemplo, un objeto que está dentro del inventario de un agente).
- Valores de movimiento de un agente hacia una posición determinada dentro de una localización.
- Valores de acción o gesto de un agente hacia una posición dentro de una localización, como atacar, lanzar un hechizo o ejecutar una animación.
- Comunicación de un agente con otro.
- Valores de ordenación de la historia.
- Valores para manejar el foco de la historia (mover la cámara a una posición o un agente, realizar un zoom, etc.).

El formato de los mensajes es muy parecido al xml.

Dentro del proyecto RCEI se ha implementado una aplicación llamada jRCEI, que permite enviar mensajes RCEI a un entorno virtual utilizando este protocolo.

Para conectar NWN con jRCEI se utiliza una librería llamada Shadow Door[6], una interfaz de control remoto de personajes no jugadores a través de sockets TCP.

[6]Existe más información acerca de Shadow Door en http://www.zubek.net/robert//software/shadow-door/0.9/Shadow %20Door %20Documentation.html

En la figura 4.5 se muestra cómo se conecta jRCEI con el entorno virtual. A grandes rasgos, jRCEI envía los mensajes RCEI que son recibidos por Shadow Door. Shadow Door da acceso al texto incluido en los mensajes RCEI a través de diferentes funciones que se pueden incluir en `scripts`. Estos `scripts` deberán ser incluidos, a su vez, en el módulo, o partida, de NWN en ejecución. En sentido inverso el funcionamiento es análogo: cuando se produce un evento en el juego, se invoca al `script` asociado. Este `script` debe tener en su código una llamada, que enviará, a través de Shadow Door, el mensaje RCEI al cliente.

La integración con el editor de comportamientos pasaría por obtener un conjunto de sensores y actuadores a partir de los mensajes RCEI y construir un generador de código. El código obtenido a partir del generador debería ser un cliente RCEI ejecutable o un fichero interpretable por un cliente RCEI, como se muestra en la figura 4.5.

El problema principal es que no existe una forma de hacer una consulta al entorno virtual acerca del estado actual. Por lo tanto, el cliente tendrá que mantener en memoria una representación del mismo. Para ello, cuando se inicia la conexión, el servidor debe responder con un mensaje en el que se de la información relevante sobre el estado inicial de todos y cada uno de los elementos del entorno. Cuando hay un cambio en el entorno, el servidor envía un mensaje al cliente informando del cambio, para que actualice el estado.

Además, sería preciso modificar cada módulo en el que se vayan a introducir comportamientos, añadiendo los `scripts` necesarios para la comunicación con el cliente. Esto se puede hacer manualmente o modificando el `script` principal que dirige la inteligencia artificial en el juego, al que también se tiene acceso a través del Aurora.

Otra cuestión a tener en cuenta es que los mensajes existentes en RCEI permiten realizar algunas acciones que podrían ser utilizadas por el comportamiento para hacer trampas en el juego, como crear objetos o personajes que le ayuden. Este tipo de acciones no deberían estar disponibles, o al menos estar restringidas mediante el cliente.

A la hora de definir el vocabulario utilizado para la descripción de comportamientos debe tenerse en cuenta, no solo las actitudes del personaje frente al jugador (si es agresivo o defensivo, por ejemplo), sino también las habilidades que utilizará habitualmente (hechizos, habilidades de ladrón, etc.). El conjunto elegido es el siguiente:

- Agresividad: esta propiedad indica si el personaje se enfrenta a sus enemigos o si huye cuando tiene la oportunidad.
- Distancia: un valor alto en esta propiedad indica que el personaje prefiere pelear a distancia con sus enemigos. Los valores más bajos indican preferencia por el combate cuerpo a cuerpo.
- Sanador: indica si el personaje se cura a sí mismo o a sus aliados cuando sufre daños.
- Magia: indica lo propenso que es el personaje a utilizar conjuros mágicos.
- Sigilo: indica si el personaje tiene tendencia a utilizar habilidades de ocultación y sigilo.
- Comunicatividad: mediante esta propiedad se indica si el personaje habla con el jugador o no.

4.5. Integración con otros entornos de simulación

Como se ha visto en los ejemplos anteriores, la integración es un proceso iterativo y diferente para cada juego o entorno de simulación. A continuación se ofrecen algunas directrices genéricas que resumen este proceso.

El primer paso es conocer el juego. Este concimiento debe ser, por un lado, acerca del funcionamiento del juego: qué género de juego es, qué características tiene dentro de este género, cuáles son los objetivos del jugador y de las entidades que maneja la inteligencia artificial, etc. De esta manera se podrá identificar qué tipo de comportamientos se tendrán que implementar posteriormente. También es necesario un conocimiento sobre la arquitectura del juego o, al menos, sobre cómo se realiza la comunicación entre el juego y los comportamientos implementados.

A continuación, es conveniente definir un conjunto de comportamientos que se consideren representativos para el juego. Como ayuda para esta tarea se puede utilizar conocimiento sobre otros juegos del mismo género.

A partir de los comportamientos representativos se puede obtener una lista inicial de los sensores y actuadores, subsistemas y propiedades necesarios para describir los comportamientos.

El siguiente paso es la definición del modelo de juego, en un fichero xml, y la implementación del generador de código. También es necesario incluir los sensores, actuadores y subsistemas necesarios, obtenidos en el paso anterior, que no estuvieran previstos inicialmente en la interfaz de comunicación entre el juego y la inteligencia artificial. Éstos se pueden añadir al juego original, si se dispone del código fuente del mismo, o hacer que sea el propio generador de código quien los calcule.

Por último, es necesario implementar algunos comportamientos para comprobar que el conjunto de sensores, actuadores y propiedades definido es suficiente. De no ser así, se realiza una nueva iteración, añadiendo o eliminando los sensores, actuadores o propiedades que se considere oportuno.

Capítulo 5

Conclusiones y líneas futuras de trabajo

Las aportaciones principales de este trabajo se resumen en los siguientes puntos:

- Se ha realizado un estudio de diferentes técnicas aplicables a la representación y modelización de comportamientos inteligentes. Entre las más destacables se encuentran las máquinas de estado finito y las técnicas de planificación, como GOAP o redes jerárquicas de tareas.
- Se ha realizado un estudio de diferentes tipos y géneros de videojuegos, para comprobar la aplicabilidad de una herramienta para el diseño de comportamientos inteligentes. También se han estudiado los diferentes métodos empleados por estos juegos para representar e implementar comportamientos. Por último, se han caracterizado distintos comportamientos comunes a varios géneros de juego, para observar las similitudes y diferencias entre ellos.
- Se ha realizado el diseño y la implementación de un editor gráfico de comportamientos, que utiliza máquinas de estado para su representación. Este editor cumple los tres objetivos planteados inicialmente en el capítulo 1:
  - Sencillez de uso: mediante el uso de una interfaz gráfica de usuario y de máquinas de estado finito para el diseño de los comportamientos, no se hace necesario tener conocimientos técnicos o de programación para el desarrollo de comportamientos.
  - Aplicabilidad: durante el proceso de evaluación y pruebas, se ha comprobado que el editor es aplicable a distintos géneros de juegos y entornos de simulación.
  - Asistencia al usuario: esto se consigue mediante la interacción con el sistema CBR integrado en el editor.
- Paralelamente al desarrollo del editor gráfico, se ha diseñado e implementado un framework para realizar distintas tareas relacionadas con el razonamiento basado en casos. Entre las tareas que se pueden realizar figuran la creación y el mantenimiento de bases de casos, almacenadas en diferentes formatos, o ejecutar consultas sobre una base de casos. En el editor de comportamientos se han incluido un conjunto de interfaces gráficas de usuario (formularios y paneles) que dan acceso a este framework y permiten utilizarlo directamente desde el editor. De esta manera se pueden recuperar comportamientos editados con anterioridad o almacenar nuevos comportamientos en la base de

casos. El framework también puede ser invocado desde fuera del editor, lo que permite hacer consultas aproximadas sobre una base de casos para recuperar comportamientos desde otros entornos (por ejemplo, desde un videojuego, en tiempo de ejecución).

- Se ha planteado con éxito la aplicación del editor y el sistema CBR a distintos videojuegos y entornos de simulación. Los distintos entornos para los que se ha propuesto su aplicación durante la fase de evaluación han sido Soccerbots (sección 4.2), un entorno de simulación de partidos de fútbol entre robots, JV^2M(sección 4.3), un videojuego de acción en tres dimensiones utilizado para la enseñanza, y Neverwinter Nights (sección 4.4), un videojuego de rol.

- Se han realizado diversas pruebas del editor y del sistema CBR, utilizando como banco de pruebas el entorno de simulación Soccerbots. A través de los resultados de las pruebas se ha comprobado que el editor cumple los objetivos planteados al principio del trabajo.

Durante el proceso de investigación, diseño y desarrollo llevado a cabo, se ha observado también que existen varios puntos que requieren diferentes mejoras para dotar al editor de mayor potencia y usabilidad[1], y merecen una investigación más profunda. En lo que resta del capítulo se exponen los más importantes.

5.1. Líneas futuras de investigación

El prototipo desarrollado pretende ser un punto de partida para futuras investigaciones en el ámbito del diseño de comportamientos y su aplicación a los videojuegos.

Existen multitud de métodos para la representación de comportamientos, entre ellos, los presentados en la sección 2.2.

Una línea de investigación que queda abierta consistiría en estudiar en profundidad cuáles de estos métodos resultan más adecuados para la representación de comportamientos en el editor y qué ventajas e inconvenientes ofrece cada uno de ellos. Otra posibilidad que ofrece una perspectiva prometedora es la combinación de distintos métodos de representación de comportamientos en diferentes niveles de la jerarquía. Así, por ejemplo, se podría diseñar un comportamiento que, en su nivel principal fuera una máquina de estado finito jerárquica, pero sus subestados se construyeran mediante sistemas de reglas o redes jerárquicas de tareas.

Dentro del ámbito del CBR, sería interesante definir una ontología sobre los diferentes géneros de juegos, y asociarla a los atributos utilizados para describir los distintos comportamientos. De esta manera se podría obtener un vocabulario genérico para utilizar en las consultas en el CBR.

Además, los juegos del mismo género tienen conjuntos de sensores y actuadores muy similares. Esta propiedad se podría aprovechar para definir modelos de juego comunes para todos los juegos de cada género, lo que facilita la reutilización de comportamientos entre distintos juegos y el uso de la herramienta, ya que el usuario no tiene que aprender cómo utilizar cada modelo de juego. En la figura 5.1 se muestra, a la izquierda, el procedimiento actual de la aplicación y, a la derecha, el resultado de utilizar un modelo de juego genérico. La definición de una ontología sobre los géneros sería de gran ayuda en esta tarea.

[1]Según la norma ISO/IEC 9126, la usabilidad es la capacidad que tiene un producto software para ser atractivo, entendido, aprendido, usado por el usuario cuando es utilizado bajo unas condiciones específicas.

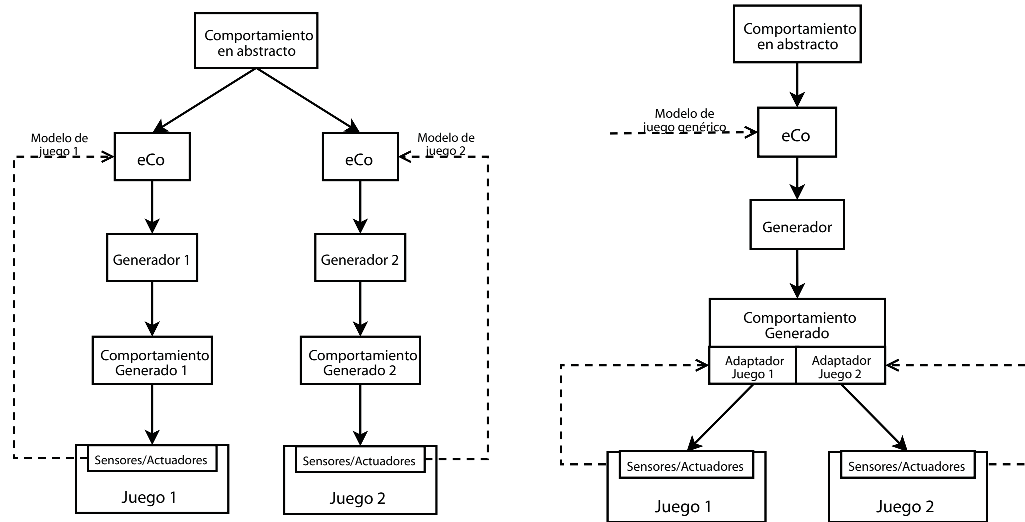


Figura 5.1: Uso de modelos de juego genéricos

Por otro lado, para dotar de mayor expresividad a las consultas, se propone ampliar las descripciones de los casos, admitiendo el uso de atributos numéricos y simbólicos, univaluados y multivaluados, como se mencionaba en la sección 3.2.1

Como se mencionó en la sección 4.1, el aprendizaje en el sistema CBR se realiza a través del usuario, que es quien proporciona los casos que considera relevantes y los valores para sus atributos. Los valores de los atributos, quedaban, pues, sujetos a la subjetividad del usuario y, en el peor de los casos, podría optar por no rellenarlos, dejando los valores por defecto. Esto puede dar lugar a una corrupción de la base de casos que puede afectar a las búsquedas realizadas sobre ella. Para evitar esta situación, se plantea el estudio de un sistema para obtener los valores de los atributos de un comportamiento automáticamente, en función de los subcomportamientos y las llamadas a sensores y actuadores que lo componen. El problema principal radica en que, en muchos casos, tanto los comportamientos como las llamadas a los sensores y actuadores no aportan un significado concreto con respecto a los atributos definidos para las consultas. Por ejemplo, el actuador `Kick`, en Soccerbots, suele darse en un comportamiento ofensivo, pero no es raro que aparezca en un comportamiento defensivo (el despeje de un defensa) o incluso en un portero. El caso del actuador `MACRO` es aún más complejo, ya que puede contener cualquier secuencia de código, incluyendo actuadores, sensores y operaciones sobre ellos.

Una técnica que parece bastante adecuada para resolver este problema es el CBR. Los casos estarían compuestos por las llamadas a sensores y actuadores y los subcomportamientos que contienen, y la lista de atributos de estos, y la solución sería la lista de atributos asignada al comportamiento. La similitud entre los casos se podría obtener comparando los subcomportamientos y llamadas que contienen, o comparando las listas de valores de atributos asociadas a ellos. Los resultados obtenidos de este tipo de búsquedas serían presentados al usuario para su confirmación, antes de introducirlos definitivamente en el comportamiento.

Por otro lado, quedan pendientes de implementar las consultas por estructura (sección 3.2.3). Antes de llevar a cabo su implementación y de integrarlas en el editor de comportamientos será necesario realizar un estudio completo sobre las distintas funciones de similitud aplicables, las restricciones que estas implican, por ejemplo, en la representación de los casos, y distintas técnicas de adaptación posibles.

En la parte de CBR textual, el procesamiento de los documentos se reduce a la eliminación de palabras vacías y la extracción de raíces, ambas tareas dentro de la *Keyword layer* del modelo propuesto por Lenz [33]. Sería interesante ampliar este procesamiento con las restantes capas, o, al menos, con algunas de ellas. En concreto, las capas de tesauro (*Thesaurus layer*) y glosario (*Glossary layer*) podrían dar buenos resultados en el tipo de consultas utilizadas, que suelen utilizar descripciones cortas, de una o dos frases.

En cuanto al prototipo en sí, se proponen una serie de mejoras para otorgarle mayor facilidad de uso.

En primer lugar, sería conveniente que se permitiera incluir cualquier tipo de fórmula en las aristas de las máquinas de estado. Para ello sería necesario introducir diferentes operadores lógicos y de precedencia (paréntesis) para relacionar las llamadas a los sensores. Además, sería muy ventajoso poder comparar entre sí los valores de los diferentes sensores, y no sólo con valores constantes. Una primera aproximación podría ser construir un lenguaje sencillo para la definición de las fórmulas que incluyera estos elementos. Por otro lado, iría contra uno de los principales objetivos del trabajo si este constructor de fórmulas supusiera un aumento en la dificultad de uso del editor, especialmente para usuarios no familiarizados con los lenguajes de programación. Por este motivo, parece que la solución más adecuada sería un editor gráfico de fórmulas, en el que se presenten al usuario los distintos elementos para la construcción de fórmulas, y en el que pudiera construir una representación gráfica (por ejemplo, un árbol) de la fórmula. Estas fórmulas complejas pueden almacenarse junto con el comportamiento o en el modelo de juego para su uso posterior en los comportamientos, funcionando de manera similar a las macros.

Otra carencia observada durante las pruebas del editor es el hecho de que los parámetros de los actuadores solamente admiten valores constantes. Esta inconveniencia se resuelve en el prototipo mediante el actuador `MACRO`, que permite introducir un fragmento del código fuente de destino para que sea ejecutado como una llamada a un actuador. Sería más conveniente poder introducir como parámetros, no sólo valores constantes, sino también actuadores y operaciones entre ellos. De nuevo, el método más adecuado parece un editor gráfico.

La depuración de comportamientos dentro de un juego puede ser una tarea bastante difícil y engorrosa. En primer lugar, es posible que el comportamiento que se desea depurar únicamente se de en determinadas circunstancias que habría que reproducir cada vez que se desea realizar la depuración. Además, en muchos casos, no es fácil distinguir a simple vista si el comportamiento realiza las acciones deseadas en el momento adecuado mirando únicamente la interfaz de usuario que ofrece el juego o el entorno de simulación. Para hacer más llevadera esta tarea se propone la creación de un módulo de depuración de comportamientos. El depurador puede ser una aplicación independiente o estar integrado dentro del editor y mediante el mismo se podrían ejecutar las máquinas de estado que representan los comportamientos paso a paso, permitiendo al usuario introducir en cada instante los valores de los sensores que guiarían el desarrollo de la prueba, o de una manera continua, recibiendo como entrada, por ejemplo, una lista de activaciones de sensores en determinados instantes de tiempo, y devolviendo una lista con las llamadas a los actuadores y el instante en el que se produjeron.

En cuanto a los modelos de juego, son ficheros `xml` y pueden editarse manualmente o con un editor adecuado. Al ser su estructura fija y conocida, podría incluirse un editor de modelos de juego dentro de la aplicación, para poder modificar y crear nuevos modelos de juego. Además, durante las pruebas del prototipo, se ha observado que, al estar ligados los comportamientos y los modelos de juego, modificar un modelo de juego implica modificar los comportamientos que tiene asociados. Por este motivo, también parece necesario incluir, independientemente del editor de modelos de juego, algún tipo de herramienta de importación o actualización del modelos de juego de un comportamiento, para ajustarlo a otro

diferente.

Otras mejoras secundarias pasarían por permitir la importación de comportamientos entre diferentes modelos de juego, definiendo, por ejemplo, tablas de equivalencia de sensores, actuadores y propiedades de los distintos modelos, y facilitar la edición de comportamientos "de abajo hacia arriba", para lo cual se puede añadir una herramienta que incluya el comportamiento editado como subcomportamiento de otro comportamiento nuevo.

Apéndice A: Ficheros XML

Esquema de modelos de juego

ModeloJuego.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
   xmlns="http://www.local.com/eCo/xml/ModeloJuego.xsd"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:ij="http://www.local.com/eCo/xml/InterfazJuego.xsd"
   xmlns:prop="http://www.local.com/eCo/xml/Propiedades.xsd"
   targetNamespace="http://www.local.com/eCo/xml/ModeloJuego.xsd"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">
  <xsd:import
     namespace="http://www.local.com/eCo/xml/InterfazJuego.xsd"
     schemaLocation="InterfazJuego.xsd"/>
  <xsd:import
     namespace="http://www.local.com/eCo/xml/Propiedades.xsd"
     schemaLocation="Propiedades.xsd"/>
  <xsd:element name="modeloJuego">
    <xsd:complexType>
      <xsd:all>
        <xsd:element ref="ij:interfazJuego"/>
        <xsd:element ref="prop:propiedades"/>
      </xsd:all>
      <xsd:attribute name="nombre" type="xsd:string"
         use="required"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

InterfazJuego.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
   xmlns="http://www.local.com/eCo/xml/InterfazJuego.xsd"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.local.com/eCo/xml/InterfazJuego.xsd">
  <xsd:element name="interfazJuego">
```

```
    <xsd:complexType>
      <xsd:all>
        <xsd:element ref="sensores" minOccurs="0"/>
        <xsd:element ref="actuadores" minOccurs="0"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="sensores">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="sensor" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="actuadores">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="actuador" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="actuador" type="tpActuador"/>
  <xsd:element name="sensor" type="tpSensor"/>
  <xsd:element name="descripcion" type="xsd:string"/>
  <xsd:complexType name="tpActuador">
    <xsd:sequence>
      <xsd:element ref="descripcion" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="nombre" type="xsd:string"
       use="required"/>
    <xsd:attribute name="comando" type="xsd:string"
       use="optional"/>
    <xsd:attribute name="parametros" type="xsd:integer"
       use="required"/>
    <xsd:attribute name="categoria" type="tpCategoria"
       use="required"/>
  </xsd:complexType>
  <xsd:complexType name="tpSensor">
    <xsd:sequence>
      <xsd:element ref="descripcion" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="nombre" type="xsd:string"
       use="required"/>
    <xsd:attribute name="comando" type="xsd:string"
       use="optional"/>
    <xsd:attribute name="tipo" type="tpTipoSensor"
       use="required"/>
    <xsd:attribute name="categoria" type="tpCategoria"
       use="required"/>
  </xsd:complexType>
```

```
  <xsd:simpleType name="tpTipoSensor">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="BOOLEAN"/>
      <xsd:enumeration value="NUMERIC"/>
      <xsd:enumeration value="STRING"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="tpCategoria">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="NATURAL"/>
      <xsd:enumeration value="SINTETICO"/>
      <xsd:enumeration value="DEFECTO"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Propiedades.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified"
   targetNamespace="http://www.local.com/eCo/xml/Propiedades.xsd"
   xmlns="http://www.local.com/eCo/xml/Propiedades.xsd">
  <xs:element name="propiedades">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="propiedad" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="propiedad" type="tpPropiedad"/>
  <xs:complexType name="tpPropiedad">
    <xs:attribute name="nombre" type="xs:string"
       use="required"/>
    <xs:attribute name="valDefecto" type="xs:decimal"
       use="required"/>
    <xs:attribute name="min" type="xs:decimal" use="required"/>
    <xs:attribute name="max" type="xs:decimal" use="required"/>
  </xs:complexType>
</xs:schema>
```

Ejemplo de modelo de juego

Soccerbots.mj.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<modeloJuego nombre=""
   xmlns="http://www.local.com/eCo/xml/ModeloJuego.xsd"
   xmlns:ij="http://www.local.com/eCo/xml/InterfazJuego.xsd"
```

```
  xmlns:prop="http://www.local.com/eCo/xml/Propiedades.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.local.com/eCo/xml/ModeloJuego.xsd
  C:\DOCUME~1\owner\MISDOC~1\Programación\eclipse\eCo\xml\Modelojuego.xsd">
<ij:interfazJuego>
  <ij:actuadores>
    <ij:actuador parametros="0" categoria="SINTETICO"
       nombre="Kick">
      <ij:descripcion>Da una patada a la pelota si esta se
         encuentra a la distancia adecuada</ij:descripcion>
    </ij:actuador>
    <ij:actuador parametros="1" categoria="NATURAL"
       nombre="setKinMaxRange">
      <ij:descripcion>Establece la distancia a la que se
         pueden percibir los demás jugadores</ij:descripcion>
    </ij:actuador>
    <ij:actuador parametros="1" categoria="NATURAL"
       nombre="setDisplayString">
      <ij:descripcion>Muestra un mensaje de estado del
         robot</ij:descripcion>
    </ij:actuador>
    <ij:actuador parametros="1" categoria="NATURAL"
       nombre="setID">
      <ij:descripcion>Establece el identificador del
         robot</ij:descripcion>
    </ij:actuador>
    <ij:actuador parametros="1" categoria="NATURAL"
       nombre="setObstacleMaxRange">
      <ij:descripcion>Establece la distancia máxima a la que
         se pueden detectar obstáculos</ij:descripcion>
    </ij:actuador>
    <ij:actuador parametros="1" categoria="NATURAL"
       nombre="setSpeed">
      <ij:descripcion>Establece la velocidad del
         robot</ij:descripcion>
    </ij:actuador>
    <ij:actuador parametros="1" categoria="NATURAL"
       nombre="setSteerHeading">
      <ij:descripcion>Establece el ángulo en radianes hacia
         el que mira el robot</ij:descripcion>
    </ij:actuador>
    <ij:actuador categoria="SINTETICO" parametros="1"
       nombre="colocarseDetras">
      <ij:descripcion>Modifica la dirección hacia la que
         mira el jugador y su velocidad para colocarse
         detrás de la pelota, apuntando a la posición
         indicada por el parámetro. Los parámetros posibles
         son "porteria_contra" y
         "porteria_propia".</ij:descripcion>
    </ij:actuador>
```

```
        <ij:actuador categoria="SINTETICO" parametros="1"
           nombre="bloquear">
      <ij:descripcion>Bloquea a un jugador del equipo
         contrario para evitar que se mueva. El jugador se
         puede indicar mediante un parámetro, que puede ser:
         portero (bloquea al jugador más atrasado),
         delantero (bloquea al jugador más adelantado) o
         jugador (bloquea al jugador más
         cercano)</ij:descripcion>
    </ij:actuador>
  </ij:actuadores>
  <ij:sensores>
    <ij:sensor nombre="getBallX" categoria="SINTETICO"
       tipo="NUMERIC">
      <ij:descripcion>Devuelve la componente X del vector
         que apunta hacia la pelota</ij:descripcion>
    </ij:sensor>
    <ij:sensor nombre="getBallY" categoria="SINTETICO"
       tipo="NUMERIC">
      <ij:descripcion>Devuelve la componente Y del vector
         que apunta hacia la pelota</ij:descripcion>
    </ij:sensor>
    <ij:sensor nombre="getBallR" categoria="SINTETICO"
       tipo="NUMERIC">
      <ij:descripcion>Devuelve la distancia del jugador a la
         pelota</ij:descripcion>
    </ij:sensor>
    <ij:sensor nombre="getBallT" categoria="SINTETICO"
       tipo="NUMERIC">
      <ij:descripcion>Devuelve el ángulo en radianes del
         vector que apunta hacia la pelota</ij:descripcion>
    </ij:sensor>
    <ij:sensor nombre="getJustScored" categoria="NATURAL"
       tipo="NUMERIC">
      <ij:descripcion>Indica si se ha marcado un gol (1 si
         hay un gol a favor, -1 si es en contra, 0 en otro
         caso)</ij:descripcion>
    </ij:sensor>
    <ij:sensor nombre="getOpponentsGoalX"
       categoria="SINTETICO" tipo="NUMERIC">
      <ij:descripcion>Devuelve la componente X de un vector
         que apunta hacia la portería
         contraria</ij:descripcion>
    </ij:sensor>
    <ij:sensor nombre="getOpponentsGoalY"
       categoria="SINTETICO" tipo="NUMERIC">
      <ij:descripcion>Devuelve la componente Y de un vector
         que apunta hacia la portería
         contraria</ij:descripcion>
    </ij:sensor>
```

```
<ij:sensor nombre="getOpponentsGoalR"
   categoria="SINTETICO" tipo="NUMERIC">
  <ij:descripcion>Devuelve la distancia a la portería
     contraria</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getOpponentsGoalT"
   categoria="SINTETICO" tipo="NUMERIC">
  <ij:descripcion>Devuelve el ángulo en radianes del
     vector que apunta hacia la portería
     contraria</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getOurGoalX" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la componente X de un vector
     que apunta hacia la portería propia</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getOurGoalY" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la componente Y de un vector
     que apunta hacia la portería propia</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getOurGoalR" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la distancia a la portería
     propia</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getOurGoalT" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve el ángulo en radianes del
     vector que apunta a la portería
     propia</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="canKick" categoria="NATURAL"
   tipo="BOOLEAN">
  <ij:descripcion>Indica si el jugador está a una
     distancia adecuada para chutar la
     pelota</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getPlayerNumber" categoria="NATURAL"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve el número de jugador entre 0
     y el número total de jugadores del
     equipo</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getID" categoria="NATURAL"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve el identificador único del
     robot</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getPositionX" categoria="SINTETICO"
```

```
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la componente X del vector de
     posición del robot en coordenadas
     globales</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getPositionY" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la componente Y del vector de
     posición del robot en coordenadas
     globales</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getPositionR" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la distancia al centro del
     campo</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getPositionT" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve el ángulo del vector de
     posición del robot en coordenadas
     globales</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getSteerHeading" categoria="NATURAL"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve el ángulo hacia el que mira
     el robot</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="getTime" categoria="NATURAL"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve el tiempo transcurrido desde
     que se instanció el robot
     (timestamp)</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="KICKER_SPEED" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Velocidad a la que se chuta el
     balón</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="KICKER_SPOT_RADIUS"
   categoria="SINTETICO" tipo="NUMERIC">
  <ij:descripcion>Distancia a la cual se puede chutar la
     pelota</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="MAX_STEER" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Máxima velocidad de giro del
     robot</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="MAX_TRANSLATION"
   categoria="SINTETICO" tipo="NUMERIC">
```

```
  <ij:descripcion>Máxima velocidad de traslación del
     robot</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="RADIUS" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Radio del robot</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="CAMPO" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Valor que indica en qué campo tiene
     que marcar nuestro equipo. Vale 1 si tiene que
     marcar en el campo de la derecha y -1 si marca a la
     izquierda</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="PelotaX" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la componente X del vector de
     posición de la pelota en coordenadas
     globales</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="PelotaY" categoria="SINTETICO"
   tipo="NUMERIC">
  <ij:descripcion>Devuelve la componente Y del vector de
     posición de la pelota en coordenadas
     globales</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="PosicionNormalizadaX"
   categoria="SINTETICO" tipo="NUMERIC">
  <ij:descripcion>Posición del jugador en coordenadas
     absolutas suponiendo que el eje X del campo se hace
     positivo hacia la portería contraria. Es el
     resultado de multiplicar CAMPO *
     getPositionX</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="PelotaNormalizadaX"
   categoria="SINTETICO" tipo="NUMERIC">
  <ij:descripcion>Devuelve la componente X del vector de
     posición de la pelota en coordenadas globales
     suponiendo que el eje X del campo se hace positivo
     hacia la portería contraria. Es el resultado de
     multiplicar CAMPO * PelotaX</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="DiferenciaGoles"
   categoria="SINTETICO" tipo="NUMERIC">
  <ij:descripcion>Goles a favor menos goles en
     contra</ij:descripcion>
</ij:sensor>
<ij:sensor nombre="MasCercaPelota" categoria="SINTETICO"
   tipo="BOOLEAN">
  <ij:descripcion>Es verdadero si el jugador se
```

```
            encuentra más cerca de la pelota que los
            demás</ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="DistanciaPelota&lt;DistanciaPorteria"
          categoria="SINTETICO" tipo="BOOLEAN">
        <ij:descripcion>Es verdadero si el jugador se
            encuentra más cerca de la pelota que que de su
            propia portería</ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="MasAtras" categoria="SINTETICO"
          tipo="BOOLEAN">
        <ij:descripcion>Es verdadero si el jugador es el más
            cercano a su portería</ij:descripcion>
      </ij:sensor>
    </ij:sensores>
  </ij:interfazJuego>
  <prop:propiedades>
    <prop:propiedad nombre="Movilidad" min="0" max="5"
        valDefecto="4"/>
    <prop:propiedad nombre="Ataque" min="0" max="5"
        valDefecto="3"/>
    <prop:propiedad nombre="Defensa" min="0" max="5"
        valDefecto="2"/>
    <prop:propiedad nombre="Portero" min="0" max="1"
        valDefecto="1"/>
  </prop:propiedades>
</modeloJuego>
```

Esquema de base de casos

BaseCasos.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns="http://www.local.com/eCo/xml/BaseCasos.xsd"
   targetNamespace="http://www.local.com/eCo/xml/BaseCasos.xsd">
  <xs:element name="baseCasos">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="modeloCaso"/>
        <xs:element ref="caso" minOccurs="0"
            maxOccurs="unbounded"/>
        <xs:element ref="diccionario" minOccurs="0"/>
        <xs:element ref="dominios" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="modeloCaso">
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref="defParametro" minOccurs="0"
            maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="defParametro">
    <xs:complexType>
      <xs:attribute name="nombre" type="xs:string"
          use="required"/>
      <xs:attribute name="tipo" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="MULTISIMBOLICO"/>
            <xs:enumeration value="NUMERICO"/>
            <xs:enumeration value="TEXTO"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="min" type="xs:decimal"
          use="optional" default="0"/>
      <xs:attribute name="max" type="xs:decimal"
          use="optional" default="1"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="caso">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="parametro" minOccurs="0"
            maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="nombre" type="xs:string"
          use="required"/>
      <xs:attribute name="ubicacion" type="xs:anyURI"
          use="required"/>
    </xs:complexType>
    <xs:key name="claveNombreParametro">
      <xs:selector xpath="caso"/>
      <xs:field xpath="nombre"/>
    </xs:key>
  </xs:element>
  <xs:element name="parametro">
    <xs:complexType>
      <xs:attribute name="nombre" type="xs:string"
          use="required"/>
      <xs:attribute name="valor" type="xs:string"
          use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="diccionario">
    <xs:complexType>
```

```
    <xs:sequence>
      <xs:element ref="entrada" minOccurs="0"
         maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="documentos" type="xs:integer"
       use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="entrada">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="ubicacion" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="palabra" type="xs:string"
       use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="ubicacion">
  <xs:complexType>
    <xs:attribute name="caso" type="xs:string"
       use="required"/>
    <xs:attribute name="frecuencia" type="xs:decimal"
       use="required"/>
  </xs:complexType>
  <xs:keyref name="refClaveNombreParametro"
     refer="claveNombreParametro">
    <xs:selector xpath="ubicacion"/>
    <xs:field xpath="caso"/>
  </xs:keyref>
</xs:element>
<xs:element name="dominios">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dominio" minOccurs="0"
         maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="dominio">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="elemento" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="atributo" type="xs:string"
       use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="elemento">
  <xs:complexType>
    <xs:attribute name="nombre" type="xs:string"
```

```
        use="required"/>
      <xs:attribute name="frecuencia" type="xs:integer"
        use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Ejemplo de base de casos

Soccerbots_ini.bc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<baseCasos>
  <modeloCaso>
    <defParametro nombre="Portero" tipo="NUMERICO" min="0.0"
       max="1.0" />
    <defParametro nombre="Defensa" tipo="NUMERICO" min="0.0"
       max="5.0" />
    <defParametro nombre="Movilidad" tipo="NUMERICO" min="0.0"
       max="5.0" />
    <defParametro nombre="Ataque" tipo="NUMERICO" min="0.0"
       max="5.0" />
    <defParametro nombre="descripcion" tipo="TEXTO" />
    <defParametro nombre="comportamientos"
       tipo="MULTISIMBOLICO" />
  </modeloCaso>
  <caso nombre="Conducir pelota a portería"
     ubicacion="C:\Documents and Settings\owner\Mis
     documentos\eCo\Comportamientos\Soccerbots\chafaris.eco.xml#Conducir
     pelota a portería">
    <parametro nombre="Portero" valor="0.0" />
    <parametro nombre="Defensa" valor="0.0" />
    <parametro nombre="Movilidad" valor="5.0" />
    <parametro nombre="Ataque" valor="4.0" />
    <parametro nombre="descripcion" valor="Lleva la pelota a
       la portería contraria" />
    <parametro nombre="comportamientos"
       valor="colocarseDetras·Kick·setDisplayString" />
  </caso>
  <caso nombre="Cubrir portería" ubicacion="C:\Documents and
     Settings\owner\Mis
     documentos\eCo\Comportamientos\Soccerbots\chafaris.eco.xml#Cubrir
     portería">
    <parametro nombre="Portero" valor="0.5" />
    <parametro nombre="Defensa" valor="5.0" />
    <parametro nombre="Movilidad" valor="3.0" />
    <parametro nombre="Ataque" valor="0.0" />
    <parametro nombre="descripcion" valor="Comportamiento
       atómico que mantiene la posición entre la pelota y la
       portería" />
```

```
  <parametro nombre="comportamientos"
     valor="MACRO·setDisplayString" />
</caso>
<caso nombre="Avanzar a pelota" ubicacion="C:\Documents and
   Settings\owner\Mis
   documentos\eCo\Comportamientos\Soccerbots\chafaris.eco.xml#Avanzar
   a pelota">
  <parametro nombre="Portero" valor="0.0" />
  <parametro nombre="Defensa" valor="2.5" />
  <parametro nombre="Movilidad" valor="5.0" />
  <parametro nombre="Ataque" valor="2.5" />
  <parametro nombre="descripcion" valor="Se mueve hacia la
     pelota en línea recta" />
  <parametro nombre="comportamientos"
     valor="MACRO·setDisplayString" />
</caso>
<caso nombre="Avanzar a portería" ubicacion="C:\Documents
   and Settings\owner\Mis
   documentos\eCo\Comportamientos\Soccerbots\chafaris.eco.xml#Avanzar
   a portería">
  <parametro nombre="Portero" valor="0.5" />
  <parametro nombre="Defensa" valor="1.0" />
  <parametro nombre="Movilidad" valor="4.0" />
  <parametro nombre="Ataque" valor="0.0" />
  <parametro nombre="descripcion" valor="Se mueve hacia la
     portería" />
  <parametro nombre="comportamientos"
     valor="MACRO·setDisplayString" />
</caso>
<caso nombre="Ir a centro" ubicacion="C:\Documents and
   Settings\owner\Mis
   documentos\eCo\Comportamientos\Soccerbots\chafaris.eco.xml#Ir
   a centro">
  <parametro nombre="Portero" valor="0.0" />
  <parametro nombre="Defensa" valor="1.0" />
  <parametro nombre="Movilidad" valor="3.0" />
  <parametro nombre="Ataque" valor="0.0" />
  <parametro nombre="descripcion" valor="Comportamiento
     atómico que mueve al jugador hasta el centro del campo"
     />
  <parametro nombre="comportamientos" valor="MACRO" />
</caso>
<diccionario documentos="5">
  <entrada palabra="camp">
    <ubicacion caso="Ir a centro" frecuencia="1.0" />
  </entrada>
  <entrada palabra="que">
    <ubicacion caso="Cubrir portería" frecuencia="1.0" />
    <ubicacion caso="Ir a centro" frecuencia="1.0" />
  </entrada>
```

```
<entrada palabra="jug">
  <ubicacion caso="Ir a centro" frecuencia="1.0" />
</entrada>
<entrada palabra="lín">
  <ubicacion caso="Avanzar a pelota" frecuencia="1.0" />
</entrada>
<entrada palabra="centr">
  <ubicacion caso="Ir a centro" frecuencia="1.0" />
</entrada>
<entrada palabra="al">
  <ubicacion caso="Ir a centro" frecuencia="1.0" />
</entrada>
<entrada palabra="comport">
  <ubicacion caso="Cubrir portería" frecuencia="1.0" />
  <ubicacion caso="Ir a centro" frecuencia="1.0" />
</entrada>
<entrada palabra="muev">
  <ubicacion caso="Avanzar a pelota" frecuencia="1.0" />
  <ubicacion caso="Ir a centro" frecuencia="1.0" />
  <ubicacion caso="Avanzar a portería" frecuencia="1.0" />
</entrada>
<entrada palabra="atómic">
  <ubicacion caso="Cubrir portería" frecuencia="1.0" />
  <ubicacion caso="Ir a centro" frecuencia="1.0" />
</entrada>
<entrada palabra="mantien">
  <ubicacion caso="Cubrir portería" frecuencia="1.0" />
</entrada>
<entrada palabra="a">
  <ubicacion caso="Conducir pelota a portería"
     frecuencia="1.0" />
</entrada>
<entrada palabra="pos">
  <ubicacion caso="Cubrir portería" frecuencia="1.0" />
</entrada>
<entrada palabra="contr">
  <ubicacion caso="Conducir pelota a portería"
     frecuencia="1.0" />
</entrada>
<entrada palabra="porterí">
  <ubicacion caso="Conducir pelota a portería"
     frecuencia="1.0" />
  <ubicacion caso="Cubrir portería" frecuencia="1.0" />
  <ubicacion caso="Avanzar a portería" frecuencia="1.0" />
</entrada>
<entrada palabra="pelot">
  <ubicacion caso="Avanzar a pelota" frecuencia="1.0" />
  <ubicacion caso="Conducir pelota a portería"
     frecuencia="1.0" />
  <ubicacion caso="Cubrir portería" frecuencia="1.0" />
```

```
    </entrada>
    <entrada palabra="rect">
      <ubicacion caso="Avanzar a pelota" frecuencia="1.0" />
    </entrada>
    <entrada palabra="se">
      <ubicacion caso="Avanzar a pelota" frecuencia="1.0" />
      <ubicacion caso="Avanzar a portería" frecuencia="1.0" />
    </entrada>
  </diccionario>
  <dominios>
    <dominio atributo="comportamientos">
      <elemento nombre="colocarseDetras" frecuencia="1" />
      <elemento nombre="Kick" frecuencia="1" />
      <elemento nombre="MACRO" frecuencia="4" />
      <elemento nombre="setDisplayString" frecuencia="4" />
    </dominio>
  </dominios>
</baseCasos>
```

Bibliografía

[1] Dictionary of philosophy of mind. http://philosophy.uwaterloo.ca/MindDict/index.html.

[2] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7:39–59, Marzo 1994.

[3] Björn Axelsson. Mindeditor - an end-user tool for implicit design of simulated human behaviour. Master's thesis, Umeå University, Junio 2003.

[4] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[5] Pavel Bekkerman. Fsmgenerator – finite state machine generating software. Disponible en http://fsmgenerator.sourceforge.net/ a 1/9/2007.

[6] Endika Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Diciembre 2002.

[7] Vincent Blondel, Anahi Gajardo, Maureen Heymans, Pierre Senellart, and Paul V. Van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching, July 2004.

[8] Michael Bowling, Johannes Fürnkranz, Thore Graepel, and Ron Musick. Machine learning and games. *Mach Learn*, 63:211–215, 2006.

[9] Alex J. Champandard. *AI Game Development - Synthetic Creatures with Learning and Reactive Behaviors*. New Riders Games, 2003.

[10] Neil Coaten. Blended e-learning, Octubre 2003.

[11] Ryan Cornelius, Kenneth O. Stanley, and Risto Miikkulainen. *Constructing Adaptive AI Using Knowledge-Based Neuroevolution*, chapter 8.8, pages 693–707. Charles River Media, 2006. AI Game Programming Wisdom 3.

[12] Belén Díaz Agudo, Pablo Gervás Gómez-Navarro, and Pedro A. González Calero. Introducción al razonamiento basado en casos.

[13] Marcin Detyniecki. Fundamentals on aggregation operators, 2001. AGOP.

[14] Eric Dybsand. Ai middleware: Getting into character part 4: Stottler henke's simbionic, Julio 2003.

[15] Kutluhan Erol, James Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[16] Kutluhan Erol, James Hendler, and Dana S. Nau. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, Institute for Advanced Computer Studies, Institute for Systems Research, Computer Science Department, University of Maryland, College Park, MD 20742, 1994.

[17] Dan Fu and Ryan Houlette. *The Ultimate Guide To FSMs In Game*, chapter 5.1, pages 283–302. Charles River Media, 2004. AI Game Programming Wisdom 2.

[18] Daniel Fu and Ryan Houlette. Putting ai in entertainment: An ai authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, 2002.

[19] Daniel Fu, Ryan Houlette, and Oscar Bascara. An authoring toolkit for simulation entities. en Proceedings of I/ITSEC 2001, Noviembre 2001.

[20] Daniel Fu, Ryan Houlette, and Randy Jensen. A visual environment for rapid behavior definition. en Proceedings of the 2003 Conference on Behavior Representation in Modeling and Simulation, 2003.

[21] Gaia. Propuesta de documento de diseño para jv2m, Mayo 2006.

[22] A. Gill. *Introduction To The Theory Of Finite-State Machines*. McGraw-Hill, 1962.

[23] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design*, 18(6):742–760, Junio 1999. Research report UCB/ERL M97/57.

[24] Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín, and Pedro A. González-Calero. Aprendizaje activo en simulaciones interactivas. In Manuel Lama and Eduardo Sánchez, editors, *Taller sobre Técnicas de la Inteligencia Artificial Aplicadas a la Educación at XI Conferencia de la Asociación Española para la Inteligencia Artificial (CAEPIA)*, pages 49–58, Santiago de Compostela, España, November 2005.

[25] Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín, Pedro A. González-Calero, and Guillermo Jiménez-Díaz. Jv2m, un sistema de enseñanza de la compilación de java. In Manuel Ortega-Cantero, editor, *I Simposio Nacional de Tecnologías de la Información y de las Comunicaciones en la Educación, SINTICE 2005*, pages 259–266, Granada, Spain, September 2005. Thomson.

[26] Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín, and Pedro Antonio González-Calero. *Javy: Virtual Environment for Case-Based Teaching of Java Virtual Machine*, volume 2773 of *Lecture Notes in Artificial Intelligence, subseries of LNCS*, pages 906–913. Springer Berlin / Heidelberg, 2003.

[27] Jilles van Gurp and Jan Bosch. On the implementation of finite state machines. In *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications*, pages 172–178, 1999.

[28] Stottler Henke. Simbionic data sheet. Disponible en http://www.simbionic.com/SimBionic_data_sheet_gamesim.pdf a 20/6/2007.

[29] Hai Hoang, Stephen Lee-Urban, and Héctor Muñoz-Avila. Hierarchical plan representations for encoding strategic game ai. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. AAAI Press, 2005.

[30] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[31] Boris Izyumov. Application of f-regression method to fuzzy classification problem. In *Proceedings of 3rd International Conference in Fuzzy Logic and Technology (EUSFLAT 2003), Zittau, Germany*, pages 761–766, 2003.

[32] Jesper Juul. The game, the player, the world: Looking for a heart of gameness, Noviembre 2003. Disponible en http://www.jesperjuul.net/text/gameplayerworld/ a 2/9/2007.

[33] Mario Lenz. Defining knowledge layers for textual case-based reasoning. *Lecture Notes in Computer Science*, 1488:298–??, 1998.

[34] Mario Lenz. Textual cbr and information retrieval – a comparison. In Lothar Gierl and Mario Lenz, editors, *Proceedings of the 6th German Workshop on Case-Based Reasoning, GWCBR'98, Berlin, Germany*, pages 59–66. Universität Rostock, 1998.

[35] Tom Leonard. Building an ai sensory system: Examining the design of thief: The dark project. In *Game Developer's Conference Proceedingsz*, Marzo 2003.

[36] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2007. To appear.

[37] Alice Mitchell and Carol Savill-Smith. The use of computer and video games for learning - a review of the literature. Technical report, Learning and Skills Development Agency, 2004.

[38] Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*. Princeton University Press, 1956.

[39] Hector Muñoz-Avila and Hai Hoang. *Coordinating Teams Of Bots With Hierarchical Task Network Planning*, chapter 5.4, pages 417–427. Charles River Media, 2006. AI Game Programming Wisdom 3.

[40] Jeff Orkin. *Applying Goal-Oriented Action Planning To Games*, chapter 3.4, pages 217–227. Charles River Media, 2004. AI Game Programming Wisdom 2.

[41] Federico Peinado and Álvaro Navarro. Protocolo y lenguaje rcei.

[42] Marc Prensky. *Digital Game-Based Learning*. McGraw-Hill, Agosto 2004.

[43] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, December 2002.

[44] Barry Smyth and Padraig Cunningham. The utility problem analysed: A case-based reasoning perspective. In *EWCBR*, pages 392–399, 1996.

[45] Penny Sweetser. *How To Build Evolutionary Algorithms For Games*, chapter 11.2, pages 627–637. Charles River Media, 2004. AI Game Programming Wisdom 2.

[46] Penny Sweetser. *How To Build Neural Networks For Games*, chapter 11.1, pages 615–625. Charles River Media, 2004. AI Game Programming Wisdom 2.

[47] Luis Valente, Aura Conci, and Bruno Feijó. Real time game loop models for single-player computer games. In *Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital, SBGames*, 2005.

[48] Neil Wallace. *Hierarchical Planning in Dynamic Worlds*, chapter 3.5, pages 229–236. Charles River Media, 2004. AI Game Programming Wisdom 2.

[49] Yong Wang and Naohiro Ishii. A method of similarity metrics for structured representations. *Expert Systems with Applications*, 12(1):89–100, 1997.