HERRAMIENTAS DE DISEÑO BASADO EN BOCETOS DEL COMPORTAMIENTO EN VIDEOJUEGOS

proyecto de investigación presentado por:

JORGE ANTONIO MAGALLANES BORBOR

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Departamento de Ingeniería del Software e Inteligencia Artificial
Trabajo de Fin Máster en Sistemas Inteligentes

Junio 2012

Directores:

Pedro Antonio González Calero,
Gonzalo Flores Puga

Calificación Obtenida: 5/10

HERRAMIENTAS DE DISEÑO BASADO EN BOCETOS DEL COMPORTAMIENTO EN VIDEOJUEGOS

proyecto de investigación presentado por:

JORGE ANTONIO MAGALLANES BORBOR

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Departamento de Ingeniería del Software e Inteligencia Artificial
Trabajo de Fin Máster en Sistemas Inteligentes

Junio 2012

Directores:

Pedro Antonio González Calero,
Gonzalo Flores Puga

Autorización de Difusión

JORGE ANTONIO MAGALLANES BORBOR

Junio 2012

El/la abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "HERRAMIENTAS DE DISEÑO BASADO EN BOCETOS DEL COMPORTAMIENTO EN VIDEOJUEGOS", realizado durante el curso académico 2011-2012 bajo la dirección de Pedro Antonio González Calero y Gonzalo Flores Puga en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

La construcción de comportamientos de personajes de videojuegos es un trabajo nada trivial y necesita de la asistencia conjunta de los programadores de IA y de los diseñadores de comportamientos. En este trabajo se presenta la propuesta de una metodología de trabajo que sirva como puente entre los diseñadores y los programadores en la construcción de estos comportamientos, proporcionando así mismo las herramientas que lo soporten. Se describe cada uno de los componentes de la metodología y son revisados en conjunto con la herramienta de autoría desarrollada. Entre los componentes más importantes la representación por medio de máquinas de estado jerárquicas, los comportamientos básicos y un sistema CBR. Se constatan los resultados esperados a partir de ejemplos prácticos en la construcción de comportamientos desde cero y asistidos por la herramienta. Se enfatiza la recuperación basada en bocetos como herramienta útil para el diseño de nuevos comportamientos. Las bases teóricas para la recuperación basada similitud estructural de grafos también son mencionadas y utilizadas para obtener una similitud entre comportamientos.

Palabras clave

Inteligencia artificial, IA para videojuegos, Comportamientos, CBR, herramientas de autoría, videojuegos.

Resumen en inglés

Building the behavior for characters in games is a complex task and needs the collaborative work of programmers and designers. In this, we propose a method for collaborative work between designers and programmers to built behaviors the form easier and supported by an authoring tool that we have developed. We describe each component contains in this methodologies and compares into the tool. The principal's block of components we have, Hierarchical State Machines for Behaviors, Behavior's Library that contains basic behaviors and complex behaviors, for the retrieval behaviors use a system CBR. We present different results for comparing sketch base in retrieval for building new behaviors with practice examples propose for the tool. We emphasis sketch base in retrieval is a useful tool for the design of new behaviors. Furthermore we introduce the structure-based similarity for graphs that preserves the behavior similarity.

Keywords

Artificial Intelligence, AI for games, CBR, Behaviors, authoring tools, computer games.

Índice de contenidos

Autorización de Difusión iii
Resumen en castellano iv
Palabras clave iv
Resumen en inglés v
Keywords v
Índice de contenidos 1
Índice de figuras 3
Índice de tablas y listados 5
Agradecimientos 6
1 Motivación y objetivos 7
2 Estado de arte 10
2.1 El uso de la Inteligencia Artificial en videojuegos 10
2.2 Los Programadores de IA versus los diseñadores 11
2.3 El modelo sensores/controlador/actuadores 12
2.4 Técnicas de representación de comportamientos en videojuegos 13
2.4.1 Sistemas basados en Reglas 14
2.4.2 Máquinas de estados finito 15
2.4.3 Máquinas de estados Jerárquica 21
2.5 Razonamiento basado en casos (CBR) 22
2.5.1 Introducción al razonamiento basado en casos 23
2.5.2 El ciclo CBR 24
2.5.3 Representación del conocimiento 26
2.5.4 Recuperación 27
2.5.5 Adaptación 33
2.5.6 Revisión 33
2.5.7 Aprendizaje 34
2.6 Similitud estructural en sistemas CBR 35
2.6.1 Representación estructurada 36
2.6.2 Distancia de edición en grafos 36
2.6.3 Una aproximación heurística para el cálculo de similitud 38
2.7 Herramientas de autoría para el diseño de comportamientos 41

2.7.1 BrainFrame 41
2.7.2 DASSIES 43
3 Diseño basado en bocetos 47
3.1 Herramienta de autoría de comportamientos 48
3.1.1 El modelo del juego 48
3.1.2 El editor de comportamientos 50
3.1.3 Librería de comportamientos 52
3.2 Sistema CBR 54
3.2.1 Representación de Comportamientos en un repositorio 55
3.2.2 Búsquedas por Similitud Funcional 57
3.2.3 Búsquedas por Similitud Estructural 58
3.2.4 Reutilización, Revisión y Aprendizaje 61
3.3 Generación de código 62
4 Edición y generación de código de comportamientos 65
4.1 La Herramienta de autoría 65
4.1.1 Gestión de repositorios 66
4.1.2 Gestión de librería de comportamientos básicos 68
4.1.3 Edición de comportamientos 69
4.1.4 Generación de código 74
4.2 Integración con el motor de simulación 75
5 Evaluación 79
5.1 Implementación de un comportamiento sin asistencia de una herramienta 80
5.2 Diseño de un boceto de comportamientos desde cero. 85
5.3 Diseño asistido de comportamientos basado en bocetos 88
5.4 Resultados de la evaluación 90
6 Conclusiones 92
Bibliografía 93

Índice de figuras

Figura 2-1 Representación de un comportamiento con un enfoque reactivo 12
Figura 2-2 Diagrama de bloques de un RBS integrado en un videojuego 14
Figura 2-3 Representación gráfica de un Autómata de estados finito (FSA)..... 16
Figura 2-4 Maquina de estados finitos Mealy produce una salida en cada transición..... 18
Figura 2-5 Ejemplo de una FSM de Moore, un elemento de la salida asociada a un estado .. 19
Figura 2-6 Ejemplo de una FSM que maneja las sensaciones en función de estímulos 20
Figura 2-7 Ejemplo de una representación jerárquica y anidada de una HFSM 21
Figura 2-8 Gráfico extendido del ciclo de CBR de (14) 25
Figura 2-9 Suposición básica de un sistema CBR (12)..... 28
Figura 2-10: Ejemplo de tres grafos dirigidos etiquetados..... 37
Figura 2-11: BrainFrame, herramienta de edición y motor de ejecución (6). 41
Figura 2-12: Editor de comportamientos, BrainFrame (6)..... 42
Figura 2-13: Diagrama de los 4 componentes de DASSIES (24)..... 45
Figura 2-14: Pantalla inicial de BehaviourShop, selección del rol para crear un agente. (24) 45
Figura 2-15: Conjunto de capas de comportamiento priorizadas. BehaviourShop (24). 46
Figura 3-1: Diagrama de bloques de la metodología propuesta 48
Figura 3-2: Esquema del archivo xml que soporta el modelo del juego. 49
Figura 3-3: Bloque de código que representa a un comportamiento básico 53
Figura 3-4: Esquema de una librería de comportamientos..... 54
Figura 3-5: Representación de un comportamiento como un caso CBR. 57
Figura 3-6: Boceto de un comportamiento con propiedades asociadas en sus nodos 62
Figura 3-7: Pila de ejecución creada por el generador de código..... 64
Figura 4-1: Captura de la ventana principal de la herramienta de autoría eCo-Behavior. 66
Figura 4-2: Herramienta de gestión de repositorios..... 67
Figura 4-3: Estructura de archivos y directorios de un repositorio..... 68
Figura 4-4: Panel de la lista de comportamientos básicos. (Ejemplo para SBTournament) ... 69
Figura 4-5: Editor de código de un comportamiento básico. (Ejemplo para SBTournament) 69
Figura 4-6: Representación de los tipos de nodo, (1 compuesto, 2 atómico, 3 no definido) .. 70
Figura 4-7: Propiedades de un nodo compuesto..... 71
Figura 4-8: Propiedades un nodo atómico..... 72
Figura 4-9: Ejemplo de navegación e inspección proporcionado por el editor. Nº1. 72

Figura 4-10: Ejemplo de navegación e inspección proporcionado por el editor. Nº2. 73
Figura 4-11: Dialogo de propiedades de una transición..... 74
Figura 4-12: Captura de pantalla del simulador SBTournament utilizado..... 76
Figura 4-13: Código fuente generado para la contrucción de tabla de estados y sus transiciones 78
Figura 4-14: Código utilizado para la ejecución de la pila de ventanas de ejecución 78
Figura 5-1: Diagrama UML de la HFSM utilizado para implementar los comportamientos . 80
Figura 5-2: FSM para el comportamiento "Centro delantero" usando notación UML 82
Figura 5-3: Comportamiento "Centro delantero" creado por la herramienta..... 86
Figura 5-4: Comportamiento "*Forward*" central ofensivo..... 87
Figura 5-5: Comportamiento "*Center*" central defensivo 87

Índice de tablas y listados

Tabla 2.1 Funciones de similitud local, $\boldsymbol{sim}(\boldsymbol{a}, \boldsymbol{b})$. 29
Tabla 2.2 Funciones de similitud global, $\mathrm{SIM}(\mathrm{A}, \mathrm{B})$..... 30
Tabla 2.3 Otras funciones de similitud global, $\mathrm{SIM}(\mathrm{A}, \mathrm{B})$ 31
Tabla 3.1: Relación entre objetos de una HFSM y un grafo dirigido 59
Tabla 5.1: Descripción de la implementación de la HFSM para comportamientos. 81
Tabla 5.2: Comportamientos básicos de un repositorio dirigido a SBTournament..... 86
Tabla 5.3: Operaciones de edición para crear el comportamiento "*Center*"..... 88
Tabla 5.4: Lista de comportamientos del repositorio inicial para SBTournament. 89
Tabla 5.5: Consultas por similitud estructural realizadas en la herramienta..... 90

Listado 5-1: Bloque de código de implementación del comportamiento "Centro delantero" 83
Listado 5-2: Bloque de código de las acciones del estado state1 y de la condición condition0 85

Agradecimientos

Agradezco a Dios por haberme dado la oportunidad de estar realizando este máster para mejorar mi desarrollo profesional en bien de las personas que constituyen mi hogar y mi familia, también al gobierno Ecuatoriano que ha confiado en mí para que los conocimientos adquiridos sean retribuidos en el trabajo profesional y de excelencia que necesitan las instituciones Ecuatorianas. Además agradezco a la ESPOL que he decido apoyarme en esta etapa de mi vida profesional y con ganas sabré recompensar a mi alma matter con mi trabajo. Además a mis directores y profesores del máster ya que sin su colaboración no hubiese terminado este trabajo de fin de máster. Finalmente dedico este trabajo a mis hijos Joshua y Sofía y a mi esposa María Auxiliadora ya que con su sacrificios más que demostrados han sabido estar junto a mí a pesar de la distancia, dándome las fuerzas y la motivación necesarias cuando más las necesitaba.

1 Motivación y objetivos

El desarrollo de videojuegos no es una tarea trivial y necesita de un gran esfuerzo y del trabajo de un grupo multidisciplinario de personas; para tener una idea de la magnitud que esto conlleva, como se indica en (1) el desarrollo del videojuego World of Warcraft (WoW) lanzado en el 2009 tomó alrededor de 36 meses, con un equipo promedio de 900 personas entre ellas 37 diseñadores del juego junto con 32 programadores y 51 grafistas, con una producción de 5.5 millones de líneas de código y un gasto de 200 millones de dólares.

Un videojuego exitoso depende de muchos factores entre ellos debe de contener una buena historia, tener unos buenos gráficos, pero sobre todo tiene que contener situaciones que deben ser las más naturales posibles y que supongan un reto o un desafío para el jugador. Una de las forma de conseguir estas situaciones es a través de comportamientos inteligentes para los personajes contrarios no controlados por el jugador que por su siglas en inglés se conocen como NPC (non-player character). El desarrollo de estos NPCs es una tarea complicada que no solo depende de los diseñadores del juego sino que además se necesita de conocimientos de programación para la elaboración de estas entidades del juego, que difícilmente las contiene un diseñador.

Existen varias alternativas para resolver el problema descrito anteriormente y que las empresas desarrolladoras de videojuegos han apostado; una de estas alternativas es tener en la fase de pre producción una guía a manera de contrato para organizar el trabajo entre los diseñadores del juego y los programadores; otra solución a este problema es que las compañías contraten diseñadores con conocimientos de programación, inclusive en algunos casos las compañías contratan estudiantes de ciencias computacionales con conocimientos de diseño (2); otra alternativa es desarrollar a través de lenguajes de scripts que son más intuitivos y fáciles de aprender y por ser menos complicados que los lenguajes tradicionales de alto nivel. Difícilmente los buenos diseñadores contienen conocimientos de programación de manera que otra buena solución es el uso de herramientas visuales que no necesiten de estos conocimientos, estas dos herramientas UNREALKISMET una herramienta integrada en el editor de juegos Unreal Developement Kit (3) y Sandbox Editor complemento de CryENGINE 3 SDK (4) son un ejemplo.

El objetivo de este trabajo es proponer una metodología de trabajo que sirva como puente entre los diseñadores y los programadores en la construcción de estos comportamientos, proporcionando así mismo las herramientas que lo soporten.

Para la representación de comportamientos inteligentes existen varias técnicas dentro de la rama de la IA; la técnica más utilizada en videojuegos es el uso de las máquinas de estados finitos que con una representación gráfica son fáciles de entender incluso para los diseñadores sin conocimientos de programación (5; 6). Debido a la naturaleza de los comportamientos, hace que los comportamientos más complejos sean construidos en función de otros comportamientos más simples como por ejemplo en un videojuego de fútbol el comportamiento "defender" puede estar compuesto de comportamientos más sencillos como "mantenerse en su campo" y cuando llegue el balón a su area "ir al balón" y finalmente "despejar" o por ejemplo un videojuego de disparos en primera persona el comportamiento un "Soldado patrulla" estaría compuesto de "Patrulla tu área" y cuando veas a un enemigo "dispara a discreción". La representación gráfica de bocetos propuestas en este trabajo considera esta naturaleza y utiliza las máquinas de estados finitos Jerárquicas que de ahora en adelante llamaremos HFSMs (Herarchical Finite State Machines) por sus siglas en inglés.

Para facilitar la creación o edición de los comportamientos se incluye un repositorio o librería de comportamientos a manera de colección sobre el cual se podrán realizar consultas con el objetivo de encontrar comportamientos anteriormente diseñados en esta herramienta con ciertas características deseadas. Para las búsquedas o consultas se utiliza otra técnica de la IA llamada CBR (Case Based Reasoning por sus siglas en inglés) o Razonamiento basado en casos, que es una técnica de sistemas basados en conocimientos y que para resolver un problema se basa en la experiencia de problemas resueltos anteriormente, la estrategia de búsqueda es encontrar comportamientos o sub-comportamientos similares a la consulta y cuyos resultados puedan ser comparados e incluso incluidos como parte de la edición de un nuevo comportamiento.

Para facilitar la edición de los comportamientos y agilizar el proceso de creación de los mismos se plantean los siguientes objetivos, contenidos en el editor de bocetos desarrollado:

Facilidad en el trabajo colaborativo entre los diseñadores del comportamiento y los programadores, con la creación de una librería reusable de comportamientos y sub-comportamientos básicos independientes del videojuego.

Sencillez de uso, al utilizar una interface gráfica para la edición y representación visual de los comportamientos utilizando las HFSM, con facilidades de inspeccionar las jerarquías de cada sub-comportamiento.

Ayuda en la creación de comportamientos, a través de las búsqueda basadas en razonamientos basados en casos para encontrar comportamientos deseados y poder compararlos y o reutilizarlos.

Rapidez en la producción e integración del comportamiento al juego, con la incorporación de un motor de generación de código que en un principio genera código en java para el API correspondiente a "*Soccerbots Tournament"*.

2 Estado de arte

2.1 El uso de la Inteligencia Artificial en videojuegos

La Inteligencia Artificial (IA) ha estado vinculada a los videojuegos desde sus inicios, como se lo indica en (7), desde los días de los videojuegos como Pong o Pacman se han incorporado técnicas de IA para conseguir que sean entretenidos para el jugador.

No hay una definición única para el término inteligencia, aunque este término por si mismo se describe según (Raymond Kurzweil[1]) inteligencia es *"El conjunto de habilidades humanas que nos ayudan a resolver problemas con recursos limitados"*, un sin número de habilidades como aprendizaje, pensamiento abstracto, planificación, imaginación, creatividad entre otras, problemas que pueden ser resueltos utilizando una combinación de las habilidades humanas. La IA aplicada a los videojuegos tiene dos enfoques:

Primero, que es re-crear habilidades humanas de manera artificial a través de las computadoras,

Segundo, ayudar a resolver problemas complejos utilizando característica de la inteligencia biológica y aplicándolas a maneras de algoritmos computacionales.

En la IA para videojuegos nos enfrentaremos al problema de incorporar esta seria de habilidades de manera artificial con el objetivo de resolver problemas complejos y re-crear un entorno virtual del juego que sea semejante a la realidad. Es innegable que esta inteligencia debe estar presente en los NPCs para mejorar la experiencia del juego, de tal manera que los videojuegos de un solo jugador no necesiten de compañeros reales para jugarlo, y los juegos multi-jugador sean más entretenidos. Muchas compañías desarrolladoras de videojuegos han apostado por incluir estas modernas técnicas de la IA consiguiendo muy buenos resultados por ejemplo el videojuego "Black and White" (7) con representaciones de criaturas muy realistas y con un nivel de dificultad creciente, todo con el fin de dar al jugador un propósito interesante a vencer. Técnicas como capacidad de aprendizaje, búsqueda de primero en anchura, y planificador de orden medio son utilizadas para hacer de estos NPCs dignos oponentes.

---

[1] Raymond Kurzweil: inventor estadounidense, científico y experto en inteligencia artificial.

En resumen la IA en los videojuegos es utilizada para dar comportamientos realistas a los NPCs con el objetivo de conseguir entretenimiento y realismo.

2.2 Los Programadores de IA versus los diseñadores

Dotar de IA a los NPCs trae consigo nuevos problemas concernientes al trabajo colaborativo entre los diseñadores del juego y los programadores de IA, mientras que:

- los programadores de IA se encargan de crear comportamientos básicos inteligentes (recoger, disparar, moverse, etc) de manera autonómica para los NPCs,
- los diseñadores quieren controlar en detalle cada una de las acciones de los NPCs

Indiscutiblemente el trabajo entre los diseñadores y programadores se contrapone y en especial cuando el desarrollo del videojuego es construido con un enfoque buttom-up en donde primero se construyen cada uno de los distintos NPC como agentes inteligentes de forma autonómica sin seguir una línea de juego[2] y luego incorporándolos al videojuego. De esta manera cada NPC está dotado de su propio comportamiento y toma sus propias acciones de manera implícita, es decir declaras pero que no pueden ser controladas directamente por el diseñador.

Para los diseñadores es necesario controlar las entidades del juego en todo momento y para conseguirlo es necesario que existan las guías de trabajo que definan las acciones que tienen control implícito y cuáles deben ser las que deben incorporar un control explicito para que los programadores de IA incluyan un mecanismo que sobre-escriba esas situaciones sobre el sistema.

Por otro lado, los desarrolladores deben incorporar en los videojuegos bloques de construcción de código que implementen la inteligencia de los NPCs usando técnicas de la IA. Entre los bloques de construcción están:

- Comportamientos básicos: acciones básicas como recoger objetos, usar cosas, etc
- Movimientos: cómo reaccionar ante obstáculos, como abrir puertas, etc
- Toma de decisiones: alto nivel de decisión para tomar distintas acciones con el propósito de alcanzar un objetivo

Es necesario conocer la arquitectura del juego antes de empezar a codificar la IA para los NPCs de manera que los desarrolladores no necesiten conocer en detalle el videojuego.

---

[2] Línea de juego: la acción o conjunto de acciones que un NPC decide tomar

Debido a la comunicación que existe entre desarrolladores de IA y diseñadores del juego es necesario crear los mecanismos que permitan que la construcción de comportamientos para videojuegos sea una tarea más llevadera.

Antes de revisar las técnicas de IA utilizadas para la representación de comportamientos es necesario revisar un modelo que nos permita integrar la IA de forma sencilla al videojuego.

2.3 El modelo sensores/controlador/actuadores

La construcción de un comportamiento puede parecerse al diseño de una caja negra que toma un conjunto de entradas y devuelve una particular salida, este enfoque permite crear los comportamientos a manera de agentes reactivos[3] que a determinadas configuraciones de entrada el resultado será una particular configuración de salida. La figura 2.1 muestra el diagrama en bloque de un comportamiento con un enfoque reactivo que considera como la configuración de entrada un conjunto de sensores y como la configuración de salida un conjunto de actuadores.

Figura 2-1 Representación de un comportamiento con un enfoque reactivo

El intercambio de información entre el mundo virtual del juego y el componente de IA es una parte importante en la construcción de los personajes virtuales del videojuego. La manera de recopilar la información acerca del entorno del videojuego es a través de los sensores, que no son más que interfaces provistas por el motor del juego[4]. Estos sensores servirán como entradas del componente de IA y en función de ellos poder determinar las siguientes acciones a realizar. Las acciones no son otra cosa que los actuadores que también

[3] agentes reactivos: en cada ejecución evalúan cual será su siguiente acción en función del contexto actual

[4] motor del juego: sistema que permite crear y desarrollar videojuegos

deben estar implementados en la interface del motor del juego, de forma similar a los sensores.

Es claro que el bloque de control, que representa el componente de IA, es el que está encargado en todo momento de obtener la información de los sensores y en base a esta decidir que actuadores deben ejecutarse con el fin de controlar a un personaje virtual en el mundo del videojuego.

El conjunto de sensores y el conjunto de actuadores por consiguiente es único y dependiente de cada videojuego, aunque dependiendo del género del juego pueden existir varias interfaces de sensores y acciones similares.

La inclusión de comportamientos dentro de un juego se puede implementar de diferentes maneras por medio de:

- archivos de configuración, incluyen propiedades del comportamiento como por ejemplo niveles máximos y mínimos permitidos; de esta forma es muy conveniente ya que no es necesario compilar el código, pero le resta flexibilidad,
- lenguajes comunes de programación, C++ y Java son los más utilizados (8), tienen la desventaja de que se necesita volver a compilar todo pero son mucho más flexibles y de mejor rendimiento
- lenguajes de scripts, como Lua o Python o los destinados para un juego específico como por ejemplo QuakeC para Queake[5]
- marcos de trabajo, vienen incluidos como una herramienta o complemento adicional de un motor de juego, un ejemplo de este es FEAR (9) un proyecto de código liberado que soporta la construcción de IA en videojuegos.

Por cualquiera de estas formas de implementar comportamientos se necesita tener conocimientos de programación y difícilmente los diseñadores del juego cuentan con estos conocimientos por lo que se necesita una forma fácil de crearlos y de representarlos con el fin de que sean manejados por ellos.

2.4 Técnicas de representación de comportamientos en videojuegos

Para que un videojuego contenga un mundo virtual más realista y consiga el entretenimiento deseado, es necesario incluir la inteligencia artificial. Existen un sin número de técnicas que serán explicadas a continuación en esta sección, pero no todas se verán en detalle, se estudiarán más, aquellas que sustentan este trabajo.

---

[5] Quake: videojuego de disparos en primera persona

2.4.1 Sistemas basados en Reglas

Los sistemas basados en reglas generalmente consisten de un conjunto de reglas a manera de sentencias (8); sentencias del tipo:

SI condición **ENTONCES** acción

, que son adecuadas para representar un conjunto codificado reglas, similar a la experiencia humana, que permite razonar cuales acciones debo tomar en función del entorno.

Los sistemas basados en reglas aparecieron en los inicios de la IA cuando la intención era re-crear sistemas inteligentes basados en la representación del conocimiento. Los sistemas basados en reglas (RBSs) permiten modelar el razonamiento humano con el objetivo de resolver problemas de manera que tienen un gran campo de aplicación en un sin número de áreas (medicina, industria, científica, etc).

En el contexto de los videojuegos los RBSs son algo más que una colección de sentencias. La arquitectura de estos sistemas está compuesta de tres componentes.

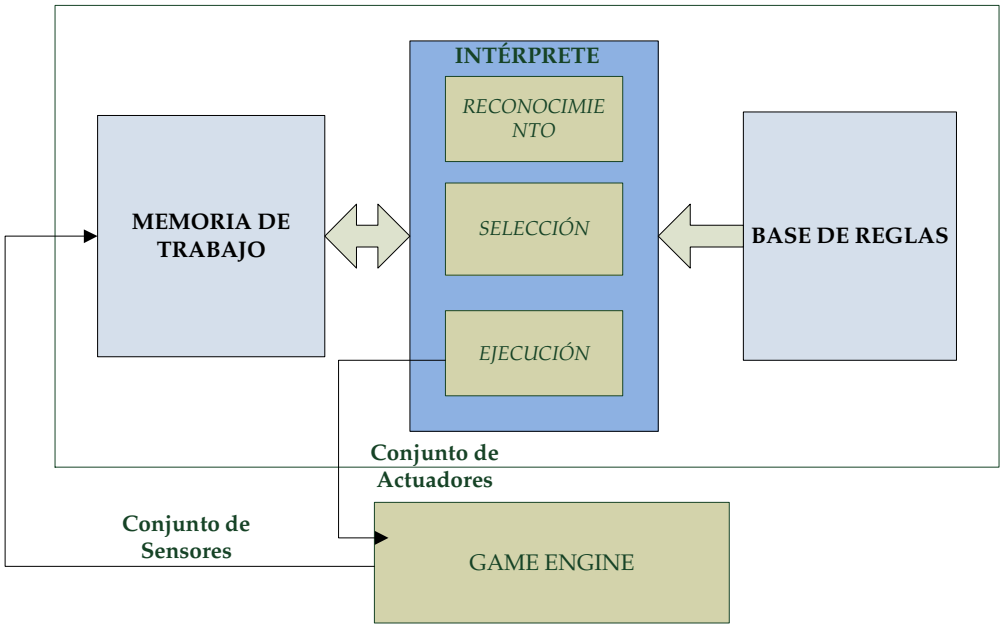

Figura 2-2 Diagrama de bloques de un RBS integrado en un videojuego

Memoria de trabajo, una colección de hechos acerca del mundo virtual del juego, generalmente es una representación interna del estado del juego.

Base de Reglas, es una base de datos en el que se encuentran almacenadas las reglas de producción para el sistema, ciertas condiciones sobre el estado del juego es decir sobre los sensores y las acciones correspondientes a realizar

Intérprete, es la parte del programa que controla el RBS e interactúa con la memoria de trabajo y con la Base de Reglas. Decide que reglas concuerdan y como afectan a la ejecución y estado actual del sistema.

El diagrama de bloques de la Figura 2-2 muestra la interacción entre cada bloque de un RBS en el contexto de un video juego. El bloque del intérprete también se lo conoce como motor de inferencia y divide su tarea en tres pasos:

- fase de reconocimiento: comparar el estado del sistema de la memoria de trabajo con la base de reglas para ver cuales condiciones son verdaderas,
- fase de selección: en esta etapa se selecciona del conjunto de reglas reconocidas las que se van a ejecutar
- fase de ejecución: realiza los cambios sobre la memoria de trabajo y ejecuta las acciones correspondiente a la regla

Entre las principales ventajas de este tipo de enfoque se pueden mencionar:

- Simplicidad: la base de reglas expresan de manera natural el razonamiento deductivo con reglas simples que definen el conocimiento.
- Modularidad: la base de reglas es independiente del motor de inferencia, por lo cual se hace simple de extender,
- Flexibilidad: se puede mejorar el conocimiento agregando más reglas para obtener resultados más precisos

Sin embargo, el mayor problema con los RBS es la adquisición del conocimiento, es necesario de expertos en el dominio de aplicación en este caso de los diseñadores del comportamiento para construir la base de datos.

2.4.2 Máquinas de estados finito

Las máquinas de estados finitos (Finite-State Machines) son un modelo computacional con una cantidad limitada de memoria conocida como *estado* (8). Cada máquina de estados tiene un número finito de estados posibles. Una función de transición determina como el estado cambia en el tiempo, de acuerdo a sus entradas.

En (8) se definen dos categorías para las FSM:

- *Autómatas de estado finitos*-, no genera ninguna salida hasta que el estado final es alcanzado, este modelo es usado para el reconocimiento de patrones (reconocer un enemigo), o clasificación de secuencias (determinar una estrategia).

- *Máquinas de estado finito*-, genera una salida cada ver que se consume una entrada, es usado para modelar el comportamiento de los NPCs (el estado de tareas actuales y cómo reaccionar ante las ciertas situaciones).

2.4.2.1 Autómatas de estado finito

Los autómatas de estado finito (Finite State Automatons, FSA por sus siglas en inglés) en el campo de la IA son ampliamente utilizados como reconocedores de patrones (8), de forma sencilla, su trabajo es reconocer un flujo de datos de entrada al autómata y determinar si corresponde ó cumple cierto patrón de entrada, la salida suele ser representada mediante un Booleano que indica la aceptación o reconocimiento del flujo de datos.

Formalmente los FSAs se define como una tupla de 5 elementos ***FSA*** = $\{\Sigma, Q, \delta, q_0, F\}$ cada elemento corresponde a:

Q, es el conjunto de estados a ser alcanzados en el autómata

δ, es la función de transición,

Σ, es la conjunto de elementos de entrada del autómata

$F \supseteq Q$, es el conjunto estados finales o de aceptación

$q_0 \in Q$, es el estado inicial

Gráficamente se puede representar mediante grafos dirigidos, véase Figura 2-3:

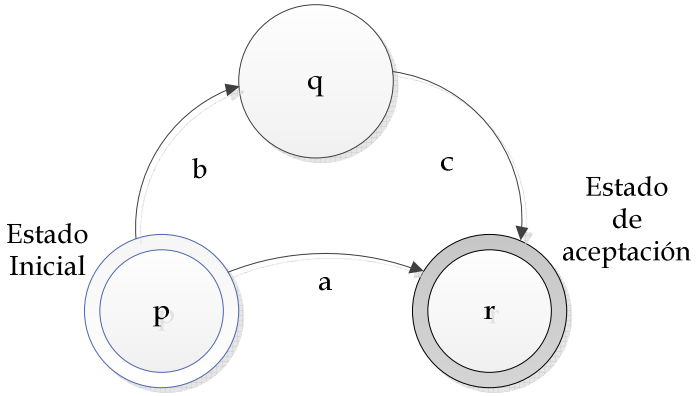

Figura 2-3 Representación gráfica de un Autómata de estados finito (FSA)

Donde los nodos representan a los diferentes estados a ser alcanzados, es decir los elementos de Q, y las transiciones son representadas mediante las aristas etiquetadas con los elementos de entrada, es decir Σ. Existen estados especiales como el estado q_0 que es el estado inicial de configuración del FSA, generalmente representado por un doble círculo,

también hay algunos estados de aceptación representados también por doble círculo pero mayormente resaltados.

La función de transición determina el siguiente estado a activar q_{t+1} basado en el estado actual q_t y el elemento de entrada σ_t y donde t denota el índice de la secuencia de entrada. De manera formal se la puede definir como el producto cartesiano de los estados y los elementos de entrada, de la siguiente forma:

$$\delta : Q \times \Sigma \rightarrow Q$$

$$(q_t, \sigma_t)\ \alpha\ q_{t+1} \mid \rightarrow (q_t, \sigma_t) \in Q \times \Sigma$$

En el ejemplo de la Figura 2-3 los elementos que representan al FSA serán:

$$\Sigma = \{p, q, r\}$$

$$\delta = \{a, b, c\}$$

$$F = \{r\}$$

$$q_0 = p$$

$$(p, \mathrm{a})\ \alpha\ r\ , (p, \mathrm{b})\ \alpha\ q, (q, \mathrm{b})\ \alpha\ r$$

Todas las transiciones deben estar indicadas en la función de transición δ, aunque para las combinaciones de estado y entrada que no se encuentren definidas en δ, se dice que son estados de no aceptación y del que no se puede salir (10).

2.4.2.2 Máquinas de estado finito

Las máquinas de estados finitos, ó FSMs por sus siglas en inglés, se diferencian de las FSAs porque producen una salida mientras se procesa la entrada. Con este enfoque no es necesario los estados de aceptación y aparecen el conjunto de elementos de salida y una función de salida.

Existen dos modelos para las FSM que son, el modelo de las máquinas de Mealy y las máquinas de Moore, la diferencia entre ellos está en la función de salida.

2.4.2.3 Máquinas de estados finitos de Mealy-

La definición formal de la FSM de Mealy está compuesta por un tupla de 6 elementos, ***FSA*** = $\{\Sigma, Q, \delta, q_0, O, \lambda\}$, donde al igual que en la FSA, Σ representa el conjunto de entrada, Q, representa el conjunto de estado y δ representa a la función de transición. Los nuevos elementos de la tupla son,

λ, la función de salida que determina el elemento de salida en función de la transición

O, el conjunto de elementos de salida

La función de salida definida formalmente es similar a la función de transición pero cambia el conjunto de llegada, la definición formal sería:

$$\lambda : Q \times \Sigma \rightarrow O$$

$$(q_t, \sigma_t)\ \alpha\ O_t \mid \rightarrow (q_t, \sigma_t) \in Q \times \Sigma$$

Como el dominio de la función de transición es igual que el dominio de la función de salida, se puede representar gráficamente por el mismo arco que define la transición dentro del modelo de grafos dirigidos que se utiliza en las FSA, de manera que, cada transición tendrá una etiqueta que corresponda al elemento de la entrada y también al elemento de la salida, una representación utilizada es la que se muestra en la Figura 2-4:

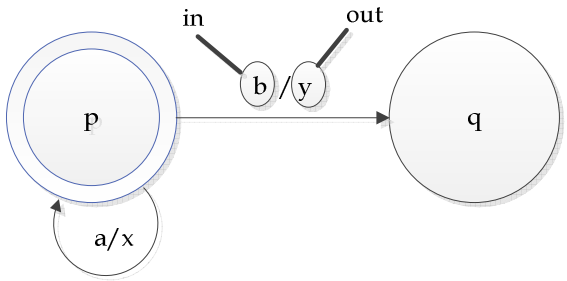

Figura 2-4 Maquina de estados finitos Mealy produce una salida en cada transición

Cada transición produce un elemento de salida y cada elemento de salida pertenece al conjunto de salida. Para el ejemplo mostrado en la Figura 2-4 el conjunto de salida y la función de salida están representados por:

$$O = \{x, y\}$$

$$\lambda = \{(p, a) \rightarrow x, (p, b) \rightarrow y\}$$

2.4.2.4 Máquinas de estados finito de Moore-

Las FSM de Moore es otro modelo alternativo, se diferencia de las máquinas de Mealy, porque en lugar de que exista una salida por cada transición, la salida es determinada por el estado alcanzado. El resto de la definición de las máquinas de Moore es el mismo que el de las máquinas de Mealy.

La función de salida es un poco más simple porque la salida solo depende del estado actual, de manera formal se puede representar a la función de salida como:

$$\lambda : Q \rightarrow O$$

$$q_t \; \alpha \; O_t \mid \rightarrow q_t \in Q$$

También la representación gráfica cambia de la siguiente manera:

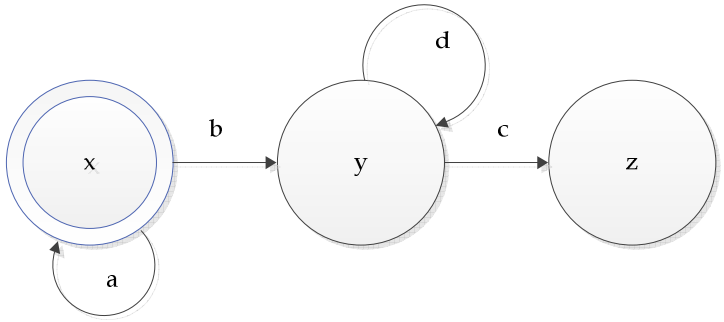

Figura 2-5 Ejemplo de una FSM de Moore, un elemento de la salida asociada a un estado

Con la máquina de Moore se aporta más simplicidad al modelo. Una cosa interesante, es que no se necesita modelar los estados recurrentes, la salida puede tomar el mismo valor mientras no se produzca o no exista una transición externa.

2.4.2.5 Enfoque de las FSMs en la integración de videojuegos

Al inicio de la sección 2.4 se introdujo el modelo "sensores/controlador/actuador" como una de forma de integrar la IA dentro de un videojuego, este enfoque puede ser aplicado de manera sencilla a las FSMs de Moore.

La representación de grafos dirigidos de las FSMs es muy intuitiva porque cada nodo del grafo y cada transición tienen una representación propia del problema, por lo tanto puede usarse como identificadores dentro del dominio del juego, En la Figura 2-6 se representa un comportamiento de un sistema de emociones para un NPC reaccionando a ciertos estímulos.

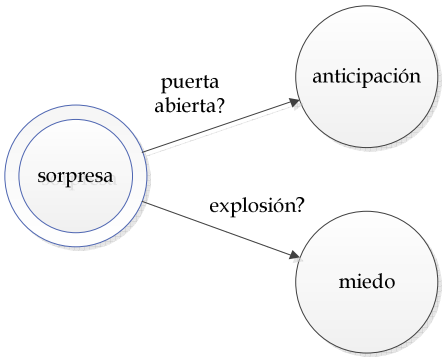

Figura 2-6 Ejemplo de una FSM que maneja las sensaciones en función de estímulos

En la técnica de representación propuesta en este trabajo el boceto de un comportamiento se representa con FSMs, en donde cada estado se interpreta como una sucesión de llamadas a los actuadores y la entrada es la información recibida por los sensores, por consiguiente, en las aristas del grafo se encontrarán condiciones sobre los sensores, cuando la condición sea cierta se producirá una transición al siguiente estado.

Para los diseñadores de comportamiento en videojuegos esta representación es adecuada permitiéndoles crear bocetos de máquinas de estados finitos para obtener una representación intuitiva y simple de implementar.

2.4.2.6 Ventajas y desventajas de las FSM

El uso de las FSM como técnica de representación para comportamientos ofrece varias ventajas entre las más importante (8; 6):

- Son simples, lo que las hace adecuadas para su utilización y también son fáciles de implementar.
- Es un modelo conocido, definido de manera formal y existe mucha teoría acerca de estas.
- Han sido bastante probadas en la IA de los videojuegos con resultados favorables

Pero también es cierto que tienen ciertos inconvenientes (8):

- Son complejas cuando se construyen grandes representaciones.
- No son adecuadas para resolver tareas sencillas como contar.
- El diseño es estático y no puede ser modificado en tiempo de ejecución.

A pesar de esta serie de desventajas, las ventajas obtenidas son indiscutibles, de manera que los problemas pueden ser manejados de alguna manera. En la siguiente sección se mostrará una mejora al modelo de las FSMs para la representación de comportamientos para hacer más simple la representación para grandes máquinas.

2.4.3 Máquinas de estados Jerárquica

La principal desventaja de las FSMs es que a mayor cantidad de estados la complejidad aumenta y es necesario encontrar un mecanismo de representación más adecuado, una forma de conseguirlo es sub-dividiendo las tareas complejas en tareas más simples. Esta forma de abstracción nos permite agrupar un conjunto de maquinas de estados finitos en una sola máquina que estaría compuesta de otras máquinas anidadas dentro de ella. Así el diseño de un comportamiento se convierte en una tarea más modular y se consigue reducir su complejidad. Esta forma de abstracción se conoce como máquinas de estados finito jerárquicas HFSM.

Las HFSMs tienen la misma representación que las FSMs, se diferencian de estas por contener estados anidados dentro de un solo estado, a este estado contenedor se lo conoce con el nombre de *super estado* y los estados interiores se llaman *sub nodos*, con esto se consigue un tipo de representación jerárquica en la cual aparece un nodo que engloba a todos los demás nodos llamado *root*. En la Figura 2-7 se muestran ambas representaciones.

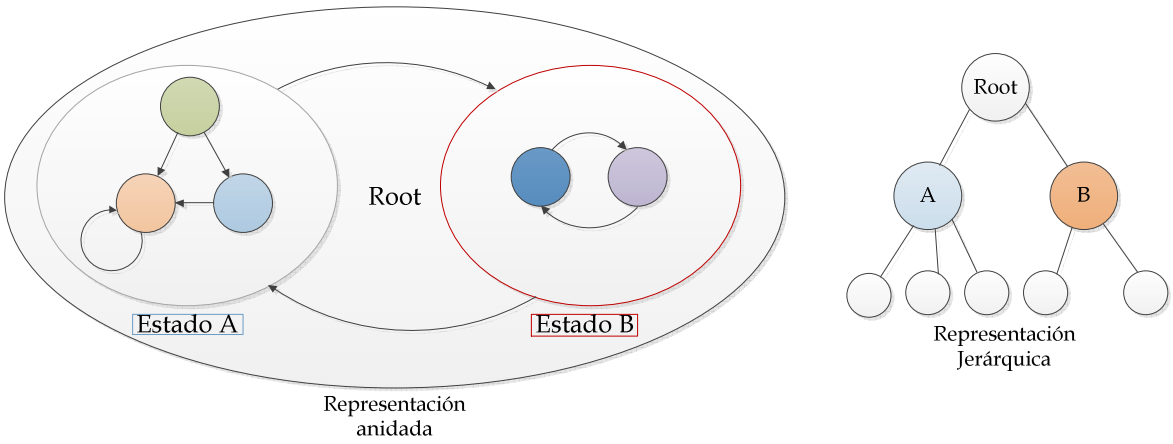

Figura 2-7 Ejemplo de una representación jerárquica y anidada de una HFSM

De esta manera se define una jerarquía de máquinas de estado en donde cada estado anidado es un componente con una funcionalidad propia, mientras que la máquina de estado

superior ensambla estos componentes de la manera más adecuada para proporcionar una funcionalidad más extensa (8).

La jerarquía, sin embargo, no aporta nada al modelo de computación y además aumenta el número de estados al introducir los nuevos super estados. Por otra parte se reducen de manera significativa el número de transiciones, lo que hace de estos modelos más simples, resultando así más fácil su representación y análisis (11).

El modelado a través de la jerarquía de las máquinas de estado provee las siguientes ventajas (8):

- Abstracción: con 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 super-estado como una caja negra.
- Refinación: Se puede afrontar el diseño empezando por los niveles más altos, definiendo super-estados abstractos, para luego continuar de la misma forma dentro de cada uno de los super-estados. De esta manera se puede seguir un diseño de arriba abajo (top-down).
- 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.

Sin embargo, existen ciertas desventajas que deben ser consideradas:

- Aunque las máquinas de estados se pueden representar con texto, su complejidad la hace difícil de entender sin el uso de alguna herramienta gráfica que represente la estructura general de la HFSM.
- Al igual que las FSMs, se han utilizado y probado en videojuegos específicamente para reproducir comportamientos complejos como un sistema de emociones y con excelentes resultados, pero su naturaleza hacen complicada su depuración y pruebas (8).

La integración con los videojuegos también es similar a las FSMs con la diferencia de que la existencia de los super-nodos y las transiciones entre ellos obliga a encontrar una forma de interactuar entre los estados interiores del super nodo y el estado inicial del super nodo que representa al siguiente super estado.

2.5 Razonamiento basado en casos (CBR)

El uso de la técnica de razonamiento basado en casos, (Case based reasoning) ó CBR por sus siglas en inglés, también sustenta este trabajo. El diseño de comportamientos es una tarea compleja y no basta con tener una forma fácil de representarlos gráficamente, sino que,

además se necesita de un mecanismo de recuperación que permita obtener partes de comportamientos construidos con anterioridad. Por este motivo, este trabajo propone también una librería de comportamientos, sobre la cual se puedan realizar consultas y recuperación de información, la forma de implementarlo es a través de las técnicas de CBR. A continuación introduciremos los detalles de esta técnica.

2.5.1 Introducción al razonamiento basado en casos

En la vida cotidiana, para resolver un nuevo problema, se trata de recordar situaciones similares y reusar toda la información o el conocimiento de aquella situación en particular. Por ejemplo, un médico que examina a un paciente en particular, trata de recordar a otro paciente que ha examinado antes cuyos síntomas principales sean similares, asumiendo que recuerda la causa de la enfermedad usa el mismo diagnóstico y el mismo tratamiento para este nuevo paciente. Los abogados que usan en sus argumentaciones basados en veredictos de casos anteriores con la finalidad de ganar un caso judicial.

Situaciones como las descritas anteriormente, usan implícitamente el razonamiento basado en casos, que no es otra cosa que un razonamiento basado en experiencias previas. Esta es la forma natural de resolver problemas de los seres humanos. Esta técnica está respaldada por investigaciones de psicología cognitiva, el fundamento es la Plausibilidad psicológica: *"El CBR es un modo natural de razonamiento de los seres humanos"* (12).

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 a manera de colección de casos. Kolodner (13) define *"A case is a contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner"*. En términos generales, un caso es la representación de una experiencia o vivencia que se compone de la *descripción del problema* y la *solución aplicada para resolverlo* (14).

El aprendizaje es el motor que hace posible que CBR pueda resolver problemas basados en casos anteriores. CBR en sí, utiliza una aproximación de aprendizaje natural, cuando se resuelve un problema de manera exitosa se registra la experiencia en la base de casos con el fin de ser utilizados para resolver futuros problemas. De forma contraria si un problema no se resuelve con determinadas experiencias se registra para no cometer el mismo error. "*Por lo tanto la noción de razonamiento basado en casos no solo tiene que ver con el método particular de razonamiento, independiente de cómo se adquieren los casos, también*

denota un paradigma de aprendizaje máquina que hace posible el aprendizaje por medio de la actualización de la base de casos cuando un problema ha sido resuelto". (14).

Dentro del campo de la IA, CBR es un paradigma para la construcción de sistemas expertos y es una alternativa a los sistemas basados en reglas (15) que revisamos en la sección 2.4.1. El problema principal de los sistemas basados en reglas es la adquisición del conocimiento, esto supone resolver las siguientes cuestiones como:

- ¿existe un experto dispuesto a dedicar el tiempo necesario para proporcionar el conocimiento?
- ¿existe un lenguaje común entre el experto y el ingeniero del conocimiento?
- ¿se puede formalizar todo el conocimiento obtenido?
- ¿existen principios aceptados por el conjunto de los expertos?

En lo que respecta a la adquisición de conocimiento CBR proporciona algunas ventajas:

- Para los expertos les resulta más sencillo concentrarse en la resolución de problemas concretos que definir un conjunto de reglas de aplicación general
- Es más práctico para los expertos encontrarse con una librería o una base de conocimientos ya creada porque ello supone transferencia de conocimiento y una explicación o justificación de la solución.
- En un dominio de un problema cualquiera del mundo real, sería prácticamente imposible, re-crear el conocimiento entero a través de reglas, es mejor tener los casos más importantes del dominio del problema (15).

2.5.2 El ciclo CBR

En (14), el proceso para resolver problemas basados en casos se describe a través del ciclo CBR de la Figura 2-8.

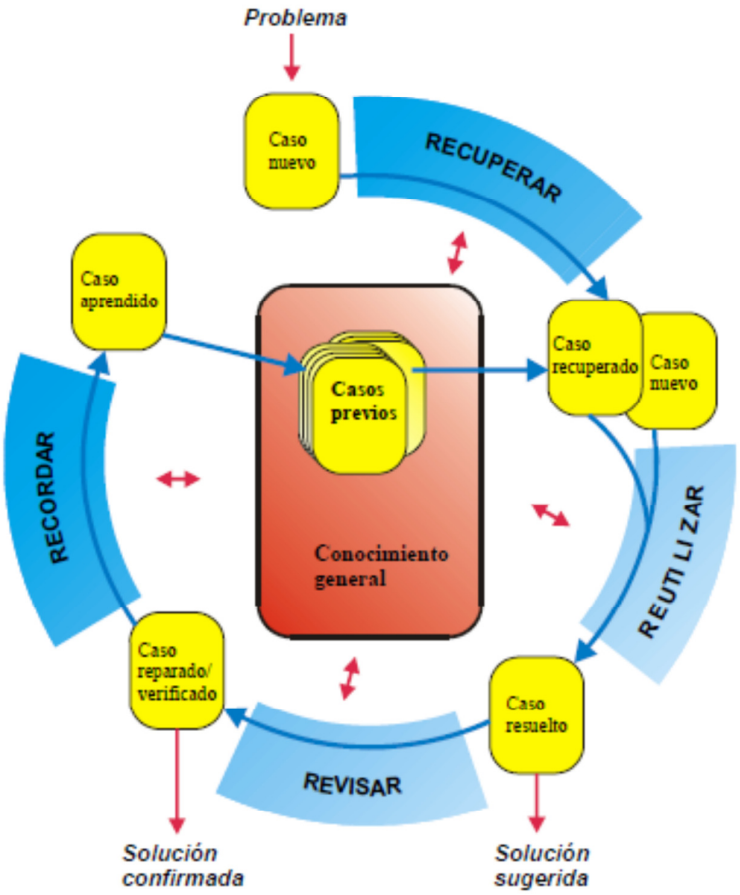

Figura 2-8 Gráfico extendido del ciclo de CBR de (14)

Básicamente el ciclo se resume en 4 procesos:

Recuperar los casos similares de la base de casos

Reutilizar los casos recuperados si fuera que resuelva el problema

Revisar la solución sugerida,

Recordar la experiencia y registrarla en la base de casos

Dentro del ciclo de CBR el proceso de recuperación es el punto de partida. Este inicia con la presentación de un problema el cual es utilizado como un nuevo caso con el fin de *recuperar* otros de la base de casos previos. El caso recuperado es combinado con el nuevo (*reusado*) para obtener una posible solución al problema inicial. Luego se *revisa* la solución comprobando si se resuelve el problema, si el problema es exitosamente resuelto se añade a la base de conocimiento como un caso exitoso, para formar parte de futuras soluciones; por el contrario, si el problema no es exitosamente resuelto se añade como un caso no exitoso con el fin de recordar que no es una solución para el problema inicial y mejorar el rendimiento.

¿Cómo se representa la base de conocimientos? CBR se ocupa de este problema y en detalle de cada uno de los procesos del ciclo CBR y que serán revisados a continuación.

2.5.3 Representación del conocimiento

CBR es estrictamente dependiente de la estructura y el modelo de la colección de casos, puesto que se necesita que los procesos de búsqueda y similitud sean efectivos y eficientes. Para la representación de un caso se debe decidir algunos aspectos, que son, la información que describe cada caso, y esto depende del dominio del conocimiento, cuál es la estructura más adecuada para describir el caso y como se va organizar e indexar la base de casos para que la recuperación de los casos se realice de forma eficiente (14).

Hay un sin número de literaturas que estudian la representación de experiencias en CBR, pero basadas en el estudio de (16), existen tres principales representaciones que son, vector de atributos (proposicional), estructural, y la textual (semi-estructural).

2.5.3.1 Vector de atributos

La representación tradicionalmente usada en CBR es la de vector de atributos (17), donde cada elemento del vector describe un par, atributo valor. El método más usual para determinar la medida de similitud en esta representación es la del vecino más cercano (nearest-neighbour) y está basada en la teoría de aprendizaje máquina coincidencia y aprendizaje basado en instancias, los k vecinos más cercanos (16).

2.5.3.2 Estructural

Para dominios de conocimiento un poco complejos, la representación de vector de atributos para un caso no es la más adecuada, porque existen representaciones intrínsecamente relacionadas que difícilmente pueden ser representadas con solo atributos, por ejemplo la estructura de las relaciones de los nodos en los grafos, necesitan más que atributos para ser representados. No existe una metodología a seguir para la representación de casos de forma estructurada, pero existen varias implementaciones que pueden tomarse de ejemplo. En (16) se referencian un sin número de trabajos que utilizan esta representación de casos para determinados problemas, cada uno de ellos con la finalidad de encontrar una similitud estructural que permita encontrar la mejor coincidencia.

2.5.3.3 Representación Textual

Como el objetivo de CBR es encontrar soluciones a partir de experiencias previas, la mayor cantidad de información se encuentra en documentos de texto, reportes de incidencias,

etc, la representación textual permite la fácil explotación de los documentos de texto. Este tipo de representación se la conoce como débilmente estructurada (Lenz & Burkhard, 1996). El proceso de forma general es definir un diccionario de palabras o frases relevantes del dominio, que se las conoce como entidades de información IEs (basada en el paradigma de *recuperación de información* ó *retrival information*), se consigue asociar un párrafo con IEs para permitir la recuperación automática de un caso. Estos casos son representados por redes de recuperación de casos y son representados por grafos dirigidos con nodos que representan a los casos y a las IEs (16).

2.5.4 Recuperación

Es claro que el proceso de recuperación está ligado intrínsecamente a la representación de los casos, pero también está relacionado a la tarea de determinar cuando un caso es similar a otro, esta forma de similitud entre casos necesita de un buen conocimiento de similitud con el objetivo de que la recuperación tenga buenos resultados y sea lo más eficiente posible.

2.5.4.1 Conocimiento de Similitud

Con el objetivo de encontrar buenos resultados de entre los casos previos, en (12) definen que la *similitud debe de modelar la utilidad de un caso previo para resolver el problema actual*. Esto implica una contradicción, puesto que para evaluar una similitud entre un caso anterior y un nuevo caso presentado, se debe determinar si la solución del caso recuperado permite determinar fácilmente la solución deseada, pero esto no se puede saber hasta haber revisado o evaluado el caso recuperado. Esto se puede sobrellevar de una manera muy sencilla, asumiendo que para soluciones parecidas existen problemas parecidos y el problema se reduce en determinar que tan parecidos son las descripciones de los casos, y basado en esta aproximación, CBR hace esta asunción, a problemas cercanos en el espacio de problemas tendrán soluciones cercanas en el espacio de soluciones (véase Figura 2-9).

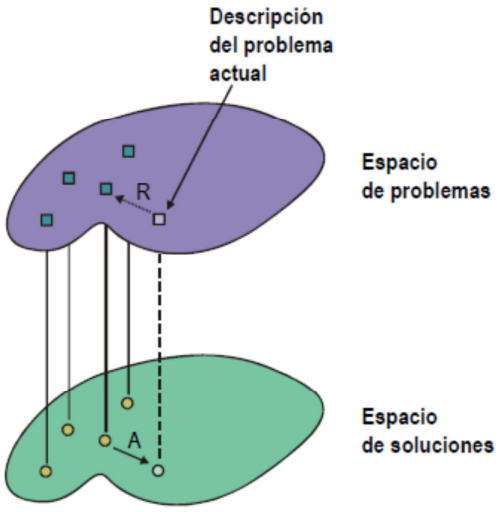

Figura 2-9 Suposición básica de un sistema CBR (12)

La tarea de encontrar similitud a partir de las descripciones entre casos es dependiente del tipo de representación que se utilice, no existe un solo método estándar para determinarla, pero en CBR se suelen utilizar ciertas aproximaciones, entre las más importantes el concepto de *similitud local* que determina la similitud entre atributos que describen a los casos *y similitud global* que obtiene un solo valor de similitud pero entre casos.

2.5.4.2 Similitud local:

Como se mencionó la similitud local es entre los atributos que describen a un caso, las aproximaciones que existen para determinar si un atributo es similar a otro, depende del tipo de atributo que se esté comparando, de forma general se obtiene un valor o medida de similitud que sirve para determinar que tan similares son entre ellos. Entre los principales tipos de atributos que se puede comparar están, números, símbolos o estructuras complejas; y si cada atributo puede tomar un único valor (uni-valuado) o varios valores simultáneamente (multi-valuado). Para atributos simbólicos y numéricos existen una serie de funciones estándares, las principales aproximaciones que existen están resumidos en la tabla.

Tabla 2.1 Funciones de similitud local, $sim(a, b)$.

	Uni-valuado	Multi-valuado	
Simbólico	$\begin{cases} 0, & si\ a \neq b \\ 1, & si\ a = b \end{cases}$	$\frac{card(a \cap b)}{card(a \cup b)}$ $\frac{card(a \cap b)}{\min(card(a), card(b))}$	$\frac{card(a \cap b)}{card(O)}$ $\frac{card(a \cap b)}{\max(card(a), card(b))}$
Numérico	$1 - \frac{\lvert a - b \rvert}{long(I)}$	$\frac{long(a \cap b)}{long(a \cup b)}$ $\frac{long(a \cap b)}{\min(long(a), long(b))}$	$1 - \frac{\lvert a_c - b_c \rvert}{long(O)}$ $\frac{long(a \cap b)}{\min(long(a), long(b))}$
$siendo,$	$O, el\ conjunto\ de\ valores\ posibles$ $long(I), la\ longitud\ o\ el\ tamaño\ del\ intervalo\ I$ $a_c\ el\ \ punto\ central\ del\ intervalo\ definido\ por\ a_{min}\ \ y\ \ a_{max}$ $, es\ decir\ los\ valores\ \ extremos\ de\ a$		

Para atributos simbólicos, una estrategia para calcular la medida de similitud es construir una tabla de similitudes donde se defina explícitamente un valor de similitud entre cada par de valores, de modo que a los atributos simbólicos ordenados se les pueden aplicar las mismas funciones que a los atributos numéricos, convirtiéndolos en números mediante el ordinal (12).

Existen ciertas propiedades que deben cumplir estas funciones, entre ellas, que una función de similitud ha de ser reflexiva y simétrica, sin embargo, en algunos trabajos se proponen refinamientos que señalan el interés de utilizar funciones de similitud no simétricas (12).

Otra posibilidad es que los atributos tengan alguna representación jerárquica, de manera que, la función de similitud será más compleja ya que debe considerar esta relación de jerarquías. Por ejemplo la siguiente función que considera la profundidad entre nodos como una medida de similitud:

$$sim(a,b) = \frac{prof(a \wedge b)}{\max(prof(a), prof(b))},$$

$$donde\ prof(n)\ es\ la\ profundidad\ del\ árbol\ en\ el\ nodo\ n,$$

$$y\ prof(a \wedge b)\ la\ profundidad\ común\ entre\ a\ y\ b$$

2.5.4.3 Similitud Global:

Como se dijo, la similitud global determina una medida de similitud para un caso general. Para encontrar esta medida, los valores de similitud local de cada uno de los atributos que conforman el caso se resumen en uno, de manera que, pueden ser usados operadores de agregación matemáticos para encontrar el resultado.

Tabla 2.2 Funciones de similitud global, SIM(A, B)

Operador de agregación	Función de similitud
Media Geométrica	$\prod_{k=1}^{n} sim(a_k, b_k)^{W_k}$
Media Aritmética	$\sum_{k=1}^{n} W_k \cdot sim(a_k, b_k)$
Media Cuadrática	$\sqrt{(\sum_{k=1}^{n} W_k \cdot sim(a_k, b_k))^2}$
$siendo,$	$n, el\ número\ de\ atributos\ de\ cada\ caso$ $a_k, b_k\ el\ atributo\ sub\ k\ para\ el\ caso\ a\ y\ b\ respectivamente$ $sim(a_k, b_k)\ la\ función\ de\ similitud\ local\ aplicada$ $\sum_{k=1}^{n} W_i = 1$

Otras posibles funciones de similitud global son:

Tabla 2.3 Otras funciones de similitud global, SIM(A, B)

Operador de agregación	Función de similitud
Minkowski	$\frac{1}{p}\left[\sum_{k=1}^{p} W_k \cdot (sim(a_k, b_k))^r\right]^{1/r}$
Máximo	$\max_{k=1..p}[W_k \cdot sim(a_k, b_k)]$
Mínimo	$\min_{k=1..p}[W_k \cdot sim(a_k, b_k)]$

La similitud en CBR supone a un caso dentro de un espacio métrico[6] con la finalidad de poder utilizar una función de similitud geométrica que determine la semejanza entre casos, las funciones de similitud de la tabla 2-1 y 2-2 son funciones geométricas y se encargan de computar la distancia en el espacio métrico definido por las funciones de similitud local (12).

2.5.4.4 Indexación de casos

Las medidas de similitud nos permiten comparar y determinar que tan parecidos son dos casos, pero existe otro factor que debe ser considerado al momento de la recuperación de casos. Con el objetivo de que en esta búsqueda se obtenga el caso de mayor relevancia para resolver el problema planteado, se debe determinar cuáles son los atributos más representativos antes de realizar la consulta. La elección de estos atributos no es una decisión trivial y que el diseñador del sistema CBR debe considerar al momento de sopesar entre rendimiento o exactitud, es decir, la elección más sencilla seria utilizar todos los atributos y realizar la consulta sobre toda la base de casos pero esto no sería lo óptimo. De modo que se debe elegir un conjunto de atributos a manera de índices que guíen en el proceso de recuperación.

El objetivo principal de un índice es que sea predictivo, de modo que, permita identificar las situaciones en las que los casos pueden aportar información útil. La elección de índices no es un problema trivial y su solución obliga a definir un compromiso entre el uso de

[6] espacio métrico: un conjunto de puntos M definidos con una métrica o función de distancia. (31)

características profundas, más predictivas pero también más difíciles de obtener, o características superficiales (12).

2.5.4.5 Algoritmo de Recuperación

El cálculo de la similitud entre casos requiere de un cómputo que afecta directamente al rendimiento de la recuperación de casos. El algoritmo más sencillo de recuperación sería calcular la similitud entre el caso de consulta y toda la base de casos, en sistemas reales donde la información de la base de conocimientos en considerable, no es factible hacer este tipo cómputo porque degradaría su utilidad.

Para sobrellevar esto, se han adaptado estructuras de datos y algoritmos apropiados para realizar este proceso eficientemente (12; 18). A continuación una breve explicación de algunas de estas estrategias.

Arboles de decisión- una vez fijados los atributos que actúan como índices, a partir del conjunto de casos son construidos mediante el algoritmo ID3[7], o alguna de sus variantes. Estos árboles de decisión permiten recuperar en un número mínimo de pasos los casos más relevantes para una consulta dada. El problema básico con los árboles de decisión es que no permiten manejar adecuadamente las consultas incompletas, por lo que se han desarrollado algunos adaptaciones como los árboles de decisión redundantes, o incluso la construcción dinámica de los mismos (12).

Modelos basados en memoria- suelen utilizarse en sistemas complejos incluyen conocimiento adicional en forma de generalizaciones de los casos con la finalidad de, no solo soportar la estructura de memoria sino también de optimizar el algoritmo de recuperación. Aadmont en su trabajo (14), destaca los siguientes, Memory Organization Packages (MOPs) de Schank o las Case Retrieval Nets desarrolladas en el proyecto INRECA.

Arboles k-d- es una generalización de los árboles de decisión desarrollada inicialmente en aplicaciones de la informática gráfica. Esta estructura está pensada para, dado un conjunto de puntos (casos) repartidos en un espacio métrico k-dimensional, recuperar eficientemente los k vecinos más próximos (k-nearest neighbors) a un punto dado (consulta). No se plantea la recuperación del vecino más cercano, simplemente recuperar los k más cercanos por cuestiones de eficiencia, basado en aproximaciones teóricamente comprobadas. (19)

---

[7] ID3: por sus siglas en ingles de “Iterative Dichotomizer (version) 3", es un algoritmo para inducir árboles de decisión desarrollado por Quinlan. ID3 es el precursor de C4.5 and C5. (32)

2.5.5 Adaptación

El proceso de adaptación es el menos estudiado dentro del ciclo CBR, en muchos casos esta parte es obviada dentro del sistema CBR y se deja a responsabilidad del usuario, facilitándole alguna forma de asistencia para realizar la reutilización del caso recuperado. En este proceso se pueden dar dos escenarios, el primero, cuando la solución propuesta es válida o es el usuario quien se encarga de adaptarla o interpretarla, en cuyo caso el sistema CBR no realiza ninguna acción, y segundo, que el sistema proponga una solución de adaptación para él caso ó los casos recuperados.

Para realizar esta adaptación de manera automática, se utiliza el conocimiento incluido en el caso recuperado. De forma general se identifican las diferencias entre las descripciones del caso nuevo y los casos recuperados, y luego se luego se aplican mecanismos que sugieran las diferencias identificadas.

Existen dos principales tipos de adaptación (19; 20):

Adaptación estructural: No trata de encontrar una solución al problema, sino de encontrar una equivalencia de soluciones, aplica transformaciones al caso recuperado usando determinados operadores de transformación sobre este. Requiere un fuerte modelo de dominio de los operadores de transformación, y un mecanismo de control para gestionar su aplicación. Los métodos utilizados son basados en la estructura, como, insertar, eliminar, o sustituir elementos; ó los basados en transformaciones, operaciones de sentido común (heurísticas generales), reparaciones guiadas apoyadas en un modelo causal.

Adaptación derivacional: este caso, lo que se reutiliza es el método mediante el cual se construyó el 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 sub-objetivos considerados, las distintas alternativas generadas o los caminos de búsqueda en el espacio de soluciones no satisfactorios. En resumen, la adaptación derivacional repite los pasos de aplicación del plan de solución recuperado, pero en el contexto del caso introducido como consulta (18).

2.5.6 Revisión

Básicamente el proceso de revisión se realiza en dos tareas, *la revisión* del caso que es determinar si la solución propuesta es aceptable, si no es adecuada, *se corrige* por intervención del usuario o utilizando algún conocimiento específico del dominio.

Para el proceso de revisión, la evaluación de la solución se da por medio de la respuesta de un experto (o del usuario) o a través de pruebas en entorno real o de simulación. En esta revisión se pueden identificar fallas, en cuyo caso, se debe determinar acciones para repararlas. Usualmente este paso se realiza fuera del sistema CBR, ya que, involucra una solución sugerida al problema real (14).

Si existieran fallos, una vez identificadas las causas del fallo, éste se repara y se genera una explicación del fallo de las expectativas. Luego, se solicita una reparación o el propio sistema se encarga de aplicar alguna estrategia de reparación. Esto implica que, en la mayoría de los casos, se incluya un conocimiento de soluciones erróneas junto con su explicación (14).

2.5.7 Aprendizaje

Es el proceso de incorporar lo que es útil de retener para resolver un problema o caso presentado dentro del conocimiento del sistema (14). Un sistema CBR mejora con el uso, al disponer de más casos a partir de los cuales obtener soluciones, es evidente suponer que mejora el comportamiento del sistema

El rendimiento de un sistema CBR, se mide por dos factores fundamentales, el espacio de cobertura o rango de problemas que es capaz de resolver, y por otro lado el tiempo de respuesta a una determinada consulta. Debido a esto hay que conseguir un balance entre estos factores con la finalidad de obtener un sistema eficiente y efectivo.

En lo concerniente a la mejora del espacio de cobertura en resolución de problemas, este se consigue con dos tareas principales, primero que crezca el tamaño de la base de casos, lo cual hace más probable que se recupere un resultado aceptable, y segundo por la relevancia del conocimiento retenido, con el fin de mejorar este resultado. La relevancia dependerá exclusivamente de la información de la revisión y de la adaptación que se registre en la base de conocimiento. La información que es necesaria registrar, entre otras es, la obtención de los índices que sirven para clasificar la nueva información, como se integra el conocimiento del nuevo caso en el lugar correcto dentro de la base de casos, realizar una reorganización de los índices, y como están representados de manera abstracta los casos (14).

Por otro lado, una base de casos lo más grande posible, no siempre es mejor, si la base de casos crece mucho, las búsquedas en ella son más lentas, es lo que se conoce como el problema de la utilidad. Para resolverlo, 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 (21).

En el aprendizaje máquina existen otros tipos de aprendizaje similares al aprendizaje de CBR, la Recuperación de Información (IR) que también permite, recuperación aproximada, medida de la similitud. A diferencia de CBR, los métodos de IR están especialmente orientados a la recuperación de texto mientras que en los sistemas CBR se accede a información heterogénea. En IR las consultas se suelen realizar en lenguaje natural sin colaboración del sistema, estas técnicas pueden ser complementarias dentro de un sistema CBR.

Otra técnica del aprendizaje máquina es uso de árboles de decisión y redes neuronales, a partir de un conjunto de casos de entrenamiento construyen una estructura que permite clasificar nuevos casos. Entre las principales diferencias, es que no se retienen los casos de entrenamiento, lo cual es muy importante en CBR, los casos sirven para justificar las decisiones. Se separan claramente las fases de aprendizaje y resolución de problemas, un sistema CBR puede empezar a funcionar con pocos casos y seguir aprendiendo indefinidamente. También pueden ser utilizadas como técnicas complementarias, algunos sistemas CBR utilizan ID3 para organizar automáticamente la memoria de casos (12).

2.6 Similitud estructural en sistemas CBR

La mayor parte de sistemas CBR utilizan para representar casos un conjunto de atributos-valor que los describan (22), pero existen problemas más complejos que no pueden ser expresados del mismo modo. Hay dominios de aplicación (representación de estructuras moleculares, problemas de grafos, etc.), donde la representación de problemas no pueden ser expresados solo con atributos, ya que, los objetos que intervienen están relacionados unos con otros. En la siguiente sección se revisará el enfoque de representar los casos de una forma estructurada que en el campo de representación del conocimiento se conoce como “representación estructurada” y de cómo encontrar una medida de similitud en este tipo de representación.

2.6.1 Representación estructurada

En (22) se utiliza la representación de grafos etiquetados dirigidos (o simplemente grafos) en lugar de una colección de atributos-valor para la representación de casos. Con este enfoque los objetos de un problema están representados por nodos y las relaciones entre ellos están representadas por aristas.

Formalmente se define un grafo como:

$$g = \{N, E, \alpha, \beta\}, \qquad donde$$

$- N\ es\ el\ conjunto\ finito\ de\ nodos$

$- E \subseteq N \times N\ es\ el\ conjunto\ finito\ de\ aristas$

$- L_N\ es\ el\ conjunto\ finito\ de\ etiquetas\ de\ nodos$

$- L_E\ es\ el\ conjunto\ finito\ de\ etiquetas\ de\ aristas$

$- \alpha : N \longmapsto L_N\ \ es\ la\ función\ de\ etiquetado\ de\ nodos$

$- \beta : E \longmapsto L_E\ \ es\ la\ función\ de\ etiquetado\ de\ aristas$

El cálculo de una medida de similitud para casos representados por atributos es básicamente la ponderación de los pesos de la medida de similitud individual de cada atributo, para el caso de una representación estructurada, el cálculo es bastante distinto porque debe también considerar los datos que conciernen a las relaciones entre los objetos de un problema.

2.6.2 Distancia de edición en grafos

En (22), se muestra un método para obtener una medida de similitud estructural, el cual es basado en una ponderación de cambios sobre un grafo con el fin de obtener una medida de similitud entre dos grafos, el autor lo define como "*weighted graph edit distance*" o *distancia de edición* para grafos y es una generalización de "*string edit distante*".

Para obtener esta *distancia de edición*, se definen un conjunto elemental de operaciones de cambios, o de edición, sobre el grafo. Operaciones como añadir un nodo, borrar un nodo, cambiar el etiquetado de un nodo, añadir una arista, eliminar una arista, cambiar el etiquetado de una arista. Estas operaciones son definidas como *operaciones de edición*. Se pueden aplicar estas operaciones para transformación de grafos, consideremos el siguiente ejemplo:

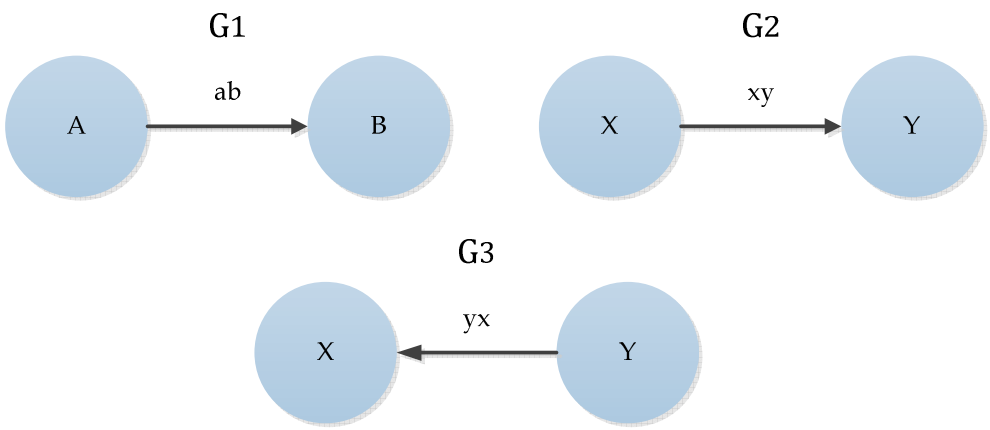

Figura 2-10: Ejemplo de tres grafos dirigidos etiquetados

De la Figura 2-10, dado un grafo G1 queremos llegar al grafo G2, para lo cual, se realizan las siguientes operaciones de edición:

- Cambiar el etiquetado del nodo "A" por etiquetado del nodo "X"
- Cambiar el etiquetado del nodo "B" por etiquetado del nodo "Y"
- Cambiar el etiquetado de la arista "ab" por etiquetado de la arista "xy"

En otro caso, dado un grafo G1 queremos llegar al grafo G3, se realizarán las mismas operaciones de edición anteriores con excepción de la última, en donde no se puede cambiar el etiquetado para llegar a la estructura deseada:

- Eliminar la arista etiquetada por "ab"
- Añadir la arista etiquetada por "yx"

En (22) formalmente se define que el conjunto de operaciones de edición es dado por,

$$EO = \{del_node, insert_node, subst_node, del_edge, insert_edge, subst_edge\},$$

Con el ejemplo mostrado en la Figura 2-10, al aplicar estas operaciones de edición podemos obtener a partir de un grafo g un nuevo grafo g'. Por lo tanto, una forma de establecer las diferencias entre dos grafos es a través de ponderar cada una de estas operaciones de edición que me permitan llegar de un grafo g a otro grafo g'. La ponderación nos permite establecer un peso o una medida entre cada operación de edición, debido a que no todas son iguales.

Formalmente, dada una función de ponderación sobre una operación de edición $c(eop) \geq 0$ y una secuencia de operaciones de edición $es = \{e_1, e_2, e_3 \dots, e_n\}$ tal que $e_i \in EO$; $i = 1, \dots, n$, se define como costo de una secuencia de edición a $c(es) =$

$\sum_{i=1}^{n} c(e_i)$. Finalmente, sean dos grafos g y g', se define como distancia de edición, a la función

$$dist(g\ ,g') = \min\{\ c(es) \mid \boldsymbol{es}\ \ transforma\ \ \boldsymbol{g}\ \ a\ \ \boldsymbol{g'}\}$$

Claramente es una función de de-similitud, ya que, es basada en las diferencias entre dos grafos, pero que fácilmente puede ser convertida en una función de similitud:

$$sim(g\ ,g') = \frac{1}{1 + dist(g, g')}$$

Ya tenemos una medida de similitud entre grafos, ahora el problema es cómo obtener esa secuencia de edición que me permita encontrar el costo mínimo necesario para la distancia de edición. En el ejemplo de la Figura 2-10 el grafo inicial y el grafo final son muy sencillos, compuesto por dos nodos cada uno, obviamente el conjunto de posibilidades de transformación es pequeño y sería fácil encontrar la secuencia de edición deseada. Sin embargo para grafos de mayor complejidad, el espacio de posibilidades es enorme y el cálculo computacionalmente sería inviable.

Un posible enfoque para encontrar la distancia de edición, es utilizar *espacios de búsqueda de estados* de configuración, que en el peor de los casos sería necesario explorar todo el espacio de búsqueda. Han sido utilizadas ciertas heurísticas para evitar explorar las partes del grafo que no contribuyan, pero sin intentar utilizarlas, el costo de computar la distancia de edición entre grafos depende exponencialmente del tamaño del grafo (22), lo cual sigue siendo costoso computacionalmente hablando.

2.6.3 Una aproximación heurística para el cálculo de similitud

Una aproximación novedosa para resolver el problema anterior es presentada en (23), la propuesta para el cálculo de la distancia de edición es basada en el *problema de asignación* y con una adaptación de las propuestas de (23).

Se considera lo siguiente, dados los siguientes grafos $G = \{N,\ E,\ \alpha, \beta\}$, $G' = \{N',\ E',\ \alpha', \beta'\}$, se define como una *correspondencia válida* (*valid mapping*) entre los grafos G y G' al conjunto M, donde, (1) todos los nodos de G están correspondidos un nodo de G' o a un nodo vacio y viceversa, (2) y todos las aristas de G estas correspondidas a una arista de G' y viceversa.

(1) $$\forall n \in N, \qquad (n, \emptyset) \in M\ \vee\ \exists n' \in N', (n, n') \in M$$

$$\forall n' \in N', \qquad (\emptyset, n') \in M \ \vee \ \exists n \in N, (n, n') \in M$$

(2)
$$\forall e \in E, \qquad (e, \emptyset) \in M \ \vee \ \exists e' \in E', (e, e') \in M$$
$$\forall e' \in E', \qquad (\emptyset, e') \in M \vee \ \exists e \in E, (e, e') \in M$$

También se define al subconjunto de correspondencias válidas entre nodos $M_N \subseteq M$,

$$M_N = \{ \ (n, n') \in M \mid \ n \in N \cup \{\emptyset\} \wedge n' \in N' \cup \{\emptyset\} \ \}$$

Fácilmente se puede encontrar una equivalencia con las operaciones de edición de grafos etiquetados de la sección 2.6.2,

$subst_node(n, l')$	(n, n')	$n \in N, n' \in N', l' \in L_N$
$del_node(n)$	$(n, \emptyset)$	$\alpha(n') = l'$
$insert_node(l')$	$(\emptyset, l')$	

Y también para las aristas:

$subst_edge(n, l')$	(e, e')	$e \in E, e' \in E', l' \in L_E$
$del_edge(e)$	$(e, \emptyset)$	$\beta(e') = l'$
$insert_edge(l')$	$(\emptyset, l')$	

El enfoque propuesto en (23), de manera general, se basa en que para encontrar una solución, no se considera todo el espacio búsqueda, sino más bien, que se concentra en una parte del espacio, el cual es definido por las correspondencias solo entre nodos y excluye a las aristas. Pero en el cómputo final, se consideran las operaciones sobre las aristas de cada nodo y sus costos.

Para reducir el espacio de búsqueda, la idea es considerar únicamente los nodos que nos permitan obtener un mejor resultado. A partir del conjunto M_N , definimos una función de coste, que es la suma individual de cada correspondencia entre nodos.

$$c(M_N) = \sum_{\forall (n,n') \in M_N} c((n, n'))$$

La mejor correspondencia entre nodos, es la correspondencia de nodos con menor coste. El problema de obtener la mejor correspondencia es equivalente al *problema de*

asignación (23). El cual puede ser resuelto por el algoritmo Húngaro con una complejidad de $O(N^3)$, siendo N el número de nodos.

Obtenida la mejor correspondencia de nodos M_{N-best}, para obtener una *correspondencia válida* es necesario incluir el coste de correspondencia de las aristas adyacentes a cada nodo que pertenezca a M_{N-best}.

Hay que tener en cuenta que, el resultado obtenido es a partir una solución al problema de asignación por lo tanto no es una solución completamente optima, puesto que para encontrar la mejor correspondencia, solo se consideran las operaciones sobre los nodos y no el de las aristas, pero el resultado obtenido puede ser utilizado como una aproximación al valor a la distancia de edición (23).

2.7 Herramientas de autoría para el diseño de comportamientos

A continuación se presentan, una serie de herramientas de autoría[8] de diseño de comportamientos de IA, con la finalidad de tener una idea general de que componentes facilitan la creación del comportamiento y la forma en que estos se integran al motor del videojuego.

2.7.1 BrainFrame

BrainFrame (6) es una herramienta de autoría gráfica de diseño de comportamientos para videojuegos y simulaciones, desarrollada por la empresa Stottler Henke Associates. En un principio destinado a la enseñanza de estudiantes de oficiales de Marina de Guerra para simulación de estrategias de guerra.

Esta herramienta provee una interface de edición gráfica de los comportamientos, junto con un motor de ejecución, que posteriormente se encarga de integrarlos dentro de la herramienta de simulación. En la Figura 2-11 se muestra el diagrama de bloques, compuesto por la herramienta de autoría de comportamientos (authoring tool) y el motor de ejecuón (engine).

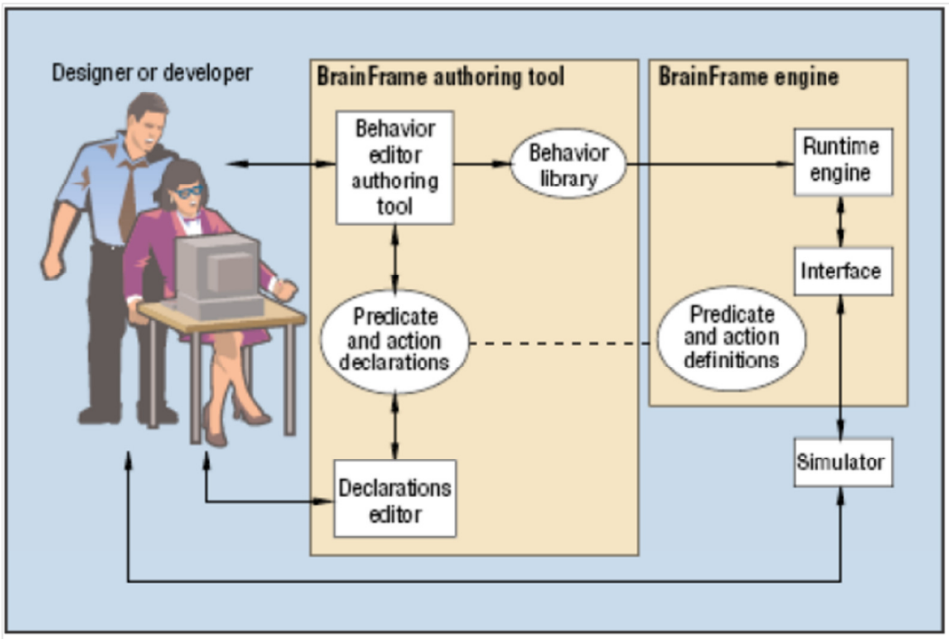

Figura 2-11: BrainFrame, herramienta de edición y motor de ejecución (6).

[8] herramienta de autoría: por su nombre en inglés authoring tool, es una herramienta que asiste en la creación de otros programas o base de datos (33)

2.7.1.1 Editor de comportamientos-

Conformado por dos bloques, el editor de declaraciones donde el creador del comportamiento proporciona un vocabulario básico de condiciones y acciones que serán utilizados como bloques de construcción por el editor de comportamientos; el editor de comportamientos en sí, que utiliza las condiciones y las acciones para construir comportamientos por medio de FSMs. Las acciones se corresponden con estados de la FSM y las condiciones se usan para determinar las transiciones entre estados (6).

La Figura 2-12 muestra una captura de pantalla del editor gráfico con un comportamiento de ejemplo construido.

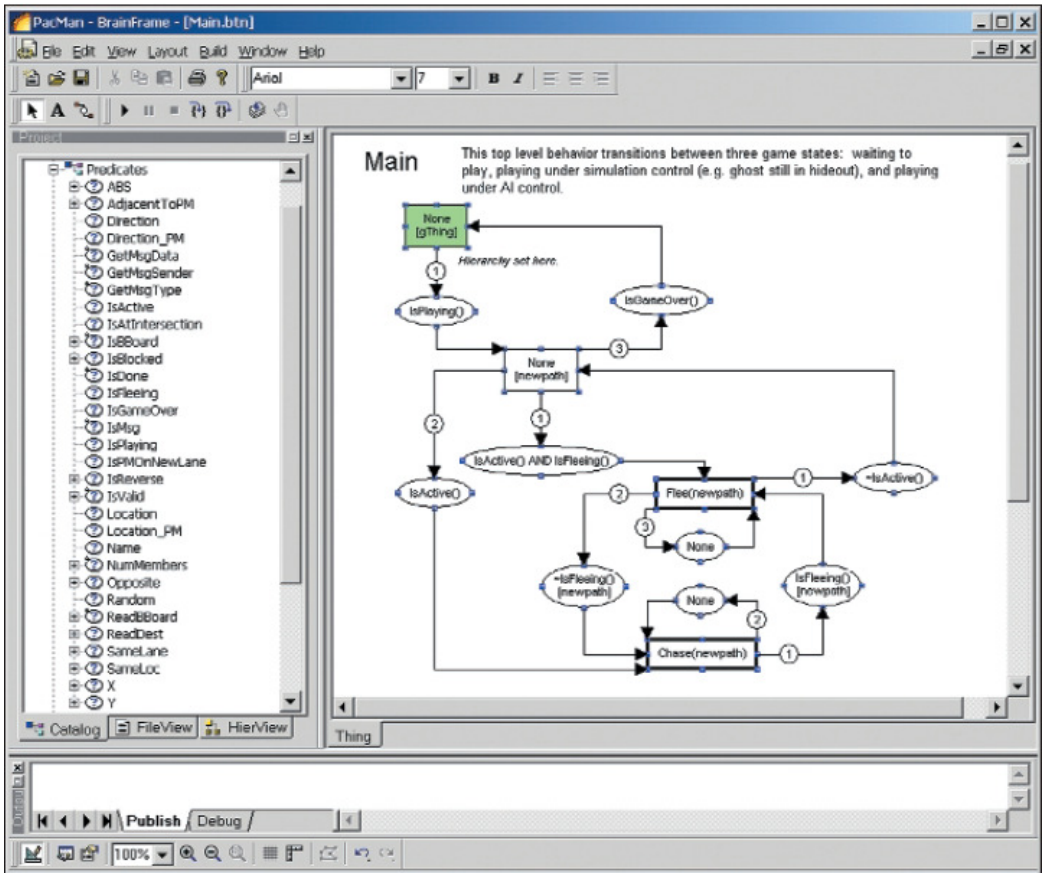

Figura 2-12: Editor de comportamientos, BrainFrame (6)

Panel izquierdo, diccionario de acciones, condiciones y comportamientos definidos,

Rectángulos: representan a los estados, las acciones o los comportamientos,

Óvalos: a las condiciones fórmulas lógicas de verdadero falso

Líneas dirigidas, para las transiciones entre estados

Cada comportamiento puede referenciar otros comportamientos permitiendo que estos sean jerárquicos, de modo que cada comportamiento está compuesto de estados simples que referencian a las acciones y de estados compuesto que referencian a los comportamientos ya

construidos, estos se agrupan en una biblioteca de comportamientos. Un comportamiento además contiene líneas dirigidas (aristas) que son condiciones compuesta por fórmulas lógicas a partir de las condiciones incluidas en el diccionario, de manera que cuando se evalúen a cierto se produzca un cambio de estado.

2.7.1.2 Motor de ejecución-

El motor de ejecución procesa el comportamiento, toma la descripción construida en el editor y la hace operacional en el juego. Para conseguir su ejecución dentro del juego, un desarrollador debe escribir el código que sirva de interfaz y que asigne las operaciones en el juego a cada acción y condición.

Para soportar la ejecución de la jerarquía de 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 y cada marco de ejecución almacena el estado de un comportamiento (nombre, estado actual y valores de las variables). Por cada 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 y se añade a la cima de la pila.

Para procesar la pila el motor de IA lo divide en dos tareas. Primero, se ejecuta la acción del estado actual del marco que se encuentra en la cima de la pila. Luego, se determina cuál es la próxima acción evaluando todas transiciones que salen del nodo actual de cada uno de los marcos, empezando por la parte de baja 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.

2.7.2 DASSIES

DASSIES (24) (de sus siglas en inglés Dynamic Adapatative Super-Scalable Intelligent Entities), es un marco de trabajo destinado a la creación de comportamientos de agentes inteligentes para un entorno de simulación.

Esta herramienta diseña comportamientos de agentes inteligentes con una técnica llamada *control basado en comportamientos*. De manera general, la creación de agentes con control basado en comportamientos, consiste de una seria de capas de comportamientos priorizados, donde cada capa enlaza con una serie de perceptores (sensores) y acciones. En

tiempo de ejecución el agente recibe uno o más sensores, activando uno o más capas de comportamientos, cada capa dispara un conjunto de acciones asociadas con el fin de manejar al agente. En (24; 5) se describe en detalle la técnica utilizada.

El entorno de simulación de esta herramienta se enfoca en simulaciones en primera y tercera persona, compuesto por una herramienta de diseño de agentes inteligentes que utiliza control basado en comportamientos, un motor de simulación que hace operacional la definición construida en el editor, y un motor de simulación que implementa los comportamientos estándares de motores de juego en primera y tercera persona (F3P).

La Figura 2-13 muestra los tres principales componentes descritos en el párrafo anterior, no se incluye al editor de entorno porque a diferencia de los otros componentes este puede ser diseñado por los modeladores 3D del entorno de simulación.

2.7.2.1 BehaviorShop-

Este componente se encarga de la construcción de agentes inteligentes, dirigido a usuarios con poca experiencia en IA, utiliza una interface que permite la construcción de agentes por medio de control basado en comportamientos. Para crear un agente se selecciona un rol que corresponde a una entidad del motor de simulación en F3P llamado FI3RST (5) que permite obtener una seria de bloques de construcción para definir al agente. Luego se debe diseñar el conjunto de capas de comportamientos que conforman al agente. Cada capa contiene sentencias del tipo if-then con la posibilidad de desencadenar un conjunto de acciones. El principal enfoque de este paradigma es que el control de comportamiento está basado en una serie de acciones y perceptores asociados a una capa dentro de un conjunto de capas priorizadas. (24).

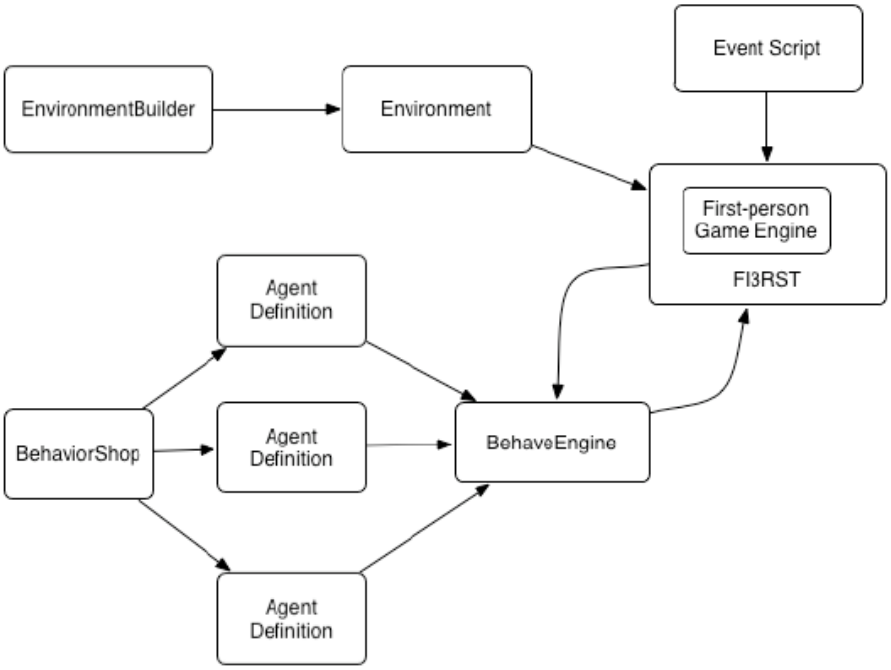

Figura 2-13: Diagrama de los 4 componentes de DASSIES (24)

La Figura 2-14 y Figura 2-15 muestran capturas de pantalla de la interface de BehaviorShop.

Figura 2-14: Pantalla inicial de BehaviourShop, selección del rol para crear un agente. (24)

Figura 2-15: Conjunto de capas de comportamiento priorizadas. BehaviourShop (24).

2.7.2.2 BEHAVEngine-

Permite la ejecución de los agentes definidos en BehaviourShop. Por medio de este componente y junto con el motor de simulación FI3RST se hace operacional la construcción de los agentes inteligentes. Se encarga de manera constante de monitorear las percepciones de los agentes, y realiza las acciones correspondientes al diseño de cada agente pasando mensajes de ejecución a FI3RST. Estos mensajes son interpretados por FI3RST y las acciones apropiadas de animación (por ejemplo, caminar, saltar, disparar a un oponente, etc.) son elegidas de la librería básica de acciones y son ejecutadas por el motor de juego. BEHAVEngine es un motor de ejecución con control basado en comportamientos que es multi-hilos de ejecución y que implementa agentes inteligentes en un entorno de videojuegos o simulación.

2.7.2.3 FI3RST-

Es un middleware entre BEHAVEngine y el motor del juego que provee de la interface estándar de un motor de videojuego de F3P, actualmente soporta motores de juego como Panda3d y Irrlicht (24).

3 Diseño basado en bocetos

El objetivo principal de este trabajo es proponer una metodología de trabajo, que sirva como puente entre los diseñadores del juego y los programadores de IA. en la construcción de comportamientos de agentes inteligentes en un entorno de videojuegos; utilizando una serie de técnicas descritas anteriormente. Esta metodología permite el diseño de comportamientos basados en bocetos de HFSMs y que puede ser aplicable a cualquier motor de videojuego o entorno de simulación. Al utilizar el modelo de agentes reactivos, no es necesario restringir su uso al ámbito de videojuegos, sino que se pueden emplearse para cualquier tarea que requiera el procesamiento de eventos reactivos (modelo sensor/controlador/actuador).

Otro de los principales objetivos de esta metodología es que la creación de bocetos de comportamientos sea dirigida a personas que no necesiten demasiados conocimientos de IA y de programación. De manera que cada uno de los componentes de esta metodología, son pensados con la finalidad de dar asistencia al diseñador y también al programador y ahorrar enormemente el tiempo empleado en la creación de comportamiento inteligente para un NPC.

La arquitectura a proponer consta de tres componentes principales. El primero es una herramienta de autoría para el diseño de comportamientos, que permite al diseñador de comportamientos diseñar comportamientos basados en máquinas de estado finito. Este editor, junto con el segundo componente que es un sistema de razonamiento basado en casos permite crear una colección o librería de comportamientos, que serán utilizados por el mismo editor para asistir al diseñador a crear nuevos comportamientos desde cero, o utilizando un comportamiento como punto de partida. El sistema CBR utiliza en las consultas la librería de comportamientos como la base de casos para obtener diseños de comportamientos similares a los que el diseñador esté bosquejando, a manera de sugerencias se ofrece al diseñador comportamientos parecidos que sirvan para la elaboración del diseño final. Esta idea es el principal aporte de la metodología para crear los diseños de comportamientos. El tercer componente es un generador de código, que a manera de interface entre la herramienta de autoría y el motor de videojuego, crea el código necesario para hacer el diseño del comportamiento operacional en un entorno de videojuego.

La Figura 3-1 muestra la arquitectura en bloques de la metodología y como se interrelacionan los componentes. A continuación se describirá en detalle cada uno de estos componentes que conforman esta arquitectura.

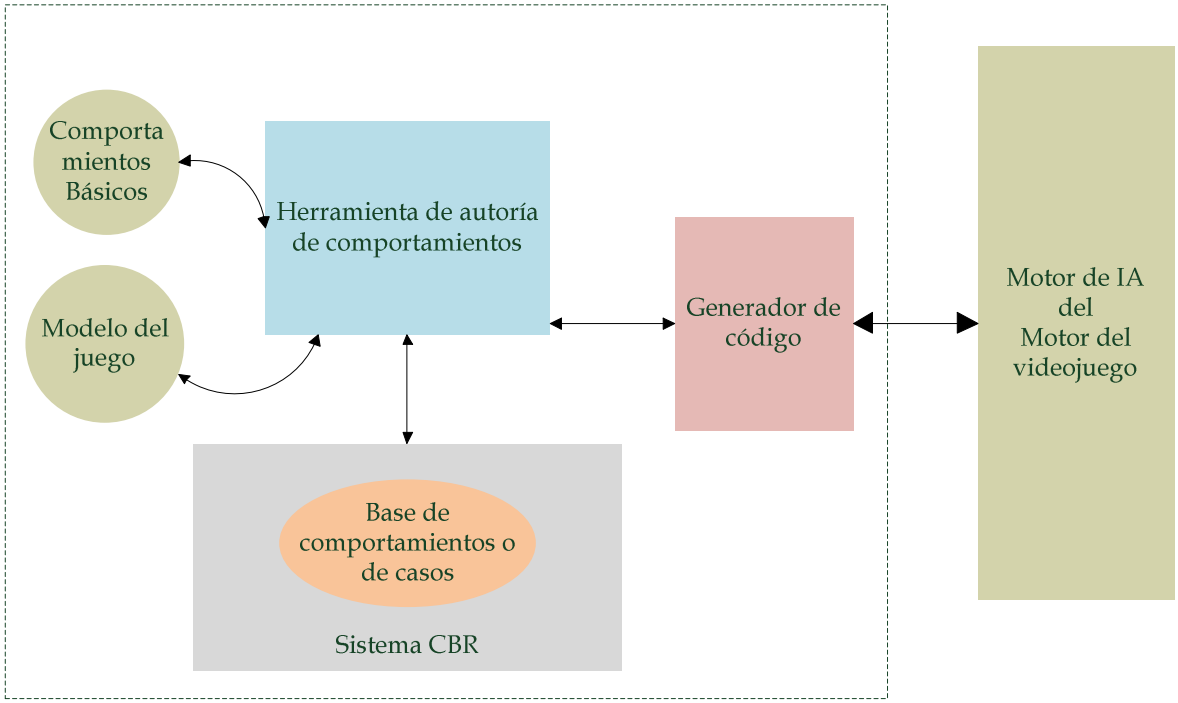

Figura 3-1: Diagrama de bloques de la metodología propuesta

3.1 Herramienta de autoría de comportamientos

La herramienta de autoría está compuesta de tres partes interrelacionadas:

El modelo del juego, que es el conjunto de sensores y actuadores que especifican la interfaz entre el motor de videojuego y la herramienta de autoría.

El editor de comportamientos basado en HFSMs, que permite crear, editar, y modificar comportamientos de forma asistida al diseñador.

Dos librerías de comportamientos, una librería de comportamientos, construida dentro de la misma herramienta de autoría por el diseñador, y una librería de comportamientos básicos construida por el programador de IA, que son bloques de construcción de código a ser utilizados por el editor, cada bloque está compuesto de un conjunto de llamadas a los actuadores y sensores.

3.1.1 El modelo del juego

Cada comportamiento es una secuencia de llamadas a las operaciones provistas por el API del motor del videojuego. Con el objetivo de soportar los distintos entornos de

simulación de videojuegos y abstraer estas llamadas se utiliza el *modelo del juego*, que básicamente es el conjunto de sensores y actuadores que deben ser provistos por el motor del videojuego. En la sección 2.3 se describió el modelo de sensores y actuadores como las interfaces para recopilar la información acerca del entorno del videojuego. Cualquier motor de videojuego que sea bueno, cuenta con un API del juego desarrollado.

La herramienta de autoría, debe conocer de este vocabulario básico provisto por el modelo del juego. Cada librería de comportamientos creada dentro de la herramienta debe tener un modelo del juego asociado, la manera más práctica es a través de un fichero xml que hace explícito el conjunto de sensores y de actuadores. Este fichero debe ser creado por alguien con conocimientos sobre la arquitectura y la estructura del videojuego, por lo general provista por un programador.

Parte de la estructura propuesta para el archivo de configuración del modelo del juego se esboza en la Figura 3-2.

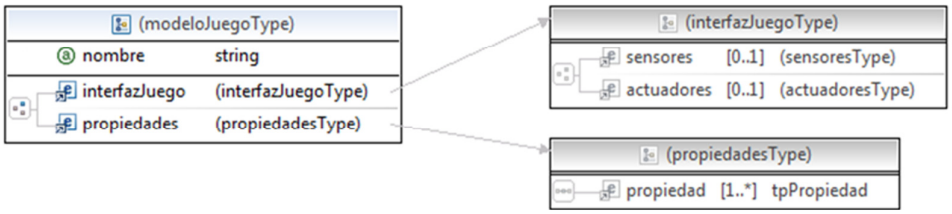

Figura 3-2: Esquema del archivo xml que soporta el modelo del juego.

- *Modelo de juego*: descrito por un atributo nombre
- *Interfaz de juego*: contiene una lista de sensores y actuadores.
- *Sensores*: definidos por el nombre del sensor, el tipo de dato primitivo que retorna, y una descripción más detallada. Además incluye el comando asociado directamente al API del motor del juego.
- *Actuadores*: contiene el nombre que lo describe, el número de parámetros que recibe, una descripción textual detallada y al igual que los sensores el comando asociado al API del motor del juego.
- *Lista de propiedades*: el modelo de juego se complementa mediante una lista en la que se declaran una serie de propiedades que se utilizarán durante el proceso de búsqueda basada en CBR. Para cada una de estas propiedades se detalla su nombre y una serie de valores.

A través de los sensores se obtiene información acerca del entorno del videojuego (ver sección 2.3), es por ello, que la herramienta de autoría los utiliza de dos maneras distintas.

- Para la construcción de condiciones a manera de fórmulas lógicas, que determinan cuando se debe realizar una transición entre los estados de la HFSM.
- Para la construcción de bloques de códigos que realizan alguna acción específica del dominio del videojuego (moverse a un lugar, caminar, saltar, etc), por lo general escrito por el programador o por alguien con conocimientos del API del motor del juego.

A través de los actuadores se realizan acciones dentro del entorno del videojuego (mover, disparar, patear, etc), no devuelven un tipo de dato en concreto y son utilizados únicamente por el editor para crear los bloques de construcción de comportamientos básicos.

A manera de ejemplo, antes de poder crear un comportamiento en la herramienta de autoría que sea dirigido al motor de simulación Soccerbots Tournament, primero se debe determinar el modelo del juego, el cual, debe estar compuesto por los sensores y actuadores descritos en la API de este motor simulación. Luego de un determinado análisis sobre las funciones de la API, se determina que por la naturaleza de las funciones, se definen los siguientes sensores: ***canKick***, ***getBallX*** y ***getBallY***, y los siguientes actuadores ***kick*** y ***setSpeed***.

3.1.2 El editor de comportamientos

Es el componente central de la herramienta de autoría, permite crear comportamientos de manera asistida basados en una representación de HFSMs. Las HFSMs, como se describe en (24), es el paradigma más utilizado y destinado a la creación de comportamientos.

Con la finalidad de facilitar el trabajo entre el diseñador de comportamientos, y el trabajo del programador de IA, el editor de comportamientos permite entre otras cosas:

- Gestionar librerías de comportamientos básicos, destinados al programador
- Gestionar librerías de comportamientos creados por el mismo editor, destinados al diseñador y programador.

Comportamientos Básicos, son bloques de construcción de código, que realizan una acción más compleja (que simplemente una llamada a un actuador) dentro del entorno del videojuego. Un comportamiento básico es utilizado por el editor para enlazarlo a un estado de

la HFSM del comportamiento que se diseña dentro del editor. Las acciones realizadas por el bloque de código, obviamente son escritas por alguien con conocimientos de programación y del modelo del juego.

Comportamientos creados por el editor, son las HFSMs creados por el editor y que representan a un comportamiento. Donde cada nodo corresponde a un estado relacionado a un comportamiento básico ó corresponde a otra máquina de estados jerárquica, y donde cada arista dirigida corresponde a una transición. La transición está asociada a una condición que cuando evalúa a cierto se produce el cambio al siguiente estado. La condición está compuesta por una fórmula lógica construida a partir de llamadas a sensores del modelo del juego.

Para incluir el formalismo que demanda el uso de HFSMs para la edición de comportamientos, se propone la siguiente representación.

- *Nodos*: representados por rectángulos redondeados.,
- *El estado inicial*: representado por un nodo con un doble borde.
- *Nodo atómico*: contiene a un nodo simple de la HFSM y está asociado a un comportamiento básico. Lo llamaremos comportamiento atómico.
- *Nodo compuesto*: contiene a un super-nodo y asociado a un sub-comportamiento que también es diseñado por el editor. Lo llamaremos comportamiento compuesto.
- *Transiciones*: representado por un conector o flecha dirigida, contiene una descripción que indica la condición de cambio de estado, y una fórmula lógica construida a partir de los sensores del modelo del juego.
- *Fórmulas lógicas*: conjunto de preposiciones lógicas compuesta de conjunciones y disyunciones. Se utilizan operadores matemáticos booleanos y de comparación y son aplicados al resultado de un sensor. La siguiente fórmula determina si un jugador puede patear el balón "puedo_patear?": "distancia_al_balon<=0,5 && alineado_al_balon".

Una característica inherente de la jerarquía de las HFSMs es que a través de este editor, se consigue que el diseño de comportamientos sea modular y se puedan construir comportamientos a partir de otros comportamientos o sub-comportamientos más básicos. Esto consigue que el diseño sea más orientado a usar una metodología "top-down", pero no obstante, también puede desarrollarse con un enfoque bootom-up. (Véase sección 2.2).

Para facilitar el diseño jerárquico en comportamientos muy extensos o con niveles de jerarquía muy profundos, este editor cuenta con una herramienta de inspección que permite la navegación en toda la jerarquía de nodos. Se propone utilizar una técnica de navegación llamada "*Path Breadcump trail*" (25), y que permite mantener en todo momento informado al diseñador en qué nivel de la jerarquía del comportamiento se encuentra.

3.1.3 Librería de comportamientos

La herramienta de autoría está compuesta por dos librerías de comportamientos. La *librería de comportamientos básicos* que es una colección de bloques de código construidos por los programadores de IA, que realizan acciones más complejas dentro del entorno de videojuegos. La *librería de comportamientos* que es una colección de comportamientos a manera de repositorio construidos por la misma herramienta.

La librería de comportamientos básicos, con la finalidad de contar con un conjunto de comportamientos generales, que puedan ser utilizados en el diseño de un comportamiento, esta librería facilita la construcción de comportamientos compuestos a partir de cada uno de los comportamientos básicos. Antes de poder diseñar un comportamiento dentro de la herramienta es necesario haber definido el conjunto de comportamientos básicos.

Un comportamiento básico, no es más que un bloque de código que hace llamadas a los sensores y actuadores del modelo del juego, con la finalidad de realizar una acción determinada dentro del juego. Para crear el bloque de código es necesario conocer el modelo del juego y de un lenguaje de programación en concreto. La Figura 3-3 muestra un ejemplo de un bloque de código dirigido a un motor de juego de fútbol, que representa a un jugador que tiene que conducir el balón en una posición defensiva.

Considerando la construcción de un comportamiento básico, se puede decir que, es un comportamiento compuesto por un conjunto de sensores y actuadores definidos dentro del API del motor del juego. De manera abstracta los elementos de este conjunto representan las acciones que corresponde a un comportamiento inteligente, y que darán un significado semántico al bloque de código. Este conjunto de elementos son utilizados por el sistema CBR para encontrar una medida de similitud.

```
//Bloque de código de un comportamiento básico
…
PosicionDestino = ObtenerPosicionDelBalon();
Jugador.EstablecerVelocidad("max");
Jugador.IrAPosicion(PosicionDestino);
If (Jugador.EstaEnPeligro())
      Jugador.RematarAFueraDelCampo();
Else
      Jugador.LlevarACampoContrario();
End If
…
```

Figura 3-3: Bloque de código que representa a un comportamiento básico

La librería de comportamientos, lo que se propone con este componente, es que el diseño de comportamientos sea basado en la existencia de un repositorio de comportamientos a manera de catálogo y que puedan contribuir a la construcción de un nuevo comportamiento. El diseño de comportamientos basados en HFSMs puede ser una tarea compleja, en especial, cuando se trata de modelar comportamientos complicados, por ejemplo, como los comportamientos virtualmente realistas de un personaje de un videojuego. Para asistir en esta ardua tarea, la librería de comportamientos es utilizada como base de casos para la búsqueda y recuperación de comportamientos similares a los que se están diseñando y permitir así el reusó de comportamientos existentes en la edición de un nuevo comportamiento. La principal contribución de esto, es asistir al diseñador en la construcción de un comportamiento, soportado por la recuperación de comportamientos enteros o parte de ellos con ciertas condiciones de similitud.

Con este enfoque la librería de comportamientos es utilizada como catálogo de comportamientos, de modo que, cada comportamiento editado por la herramienta contribuye con este catálogo. Cada comportamiento debe contener la información semántica necesaria acerca de las acciones realizadas por este. En nuestra metodología un comportamiento no es simplemente una HFSMs, compuesta por nodos y arcos dirigidos, es un modelo aumentado con información inferida y provista por la misma herramienta de autoría, a través de componentes como el modelo del juego y comportamientos básicos. En la Figura 3-4 se

muestra el modelo aumentado de la librería de comportamientos, compuesto por sensores y actuadores del modelo del juego.

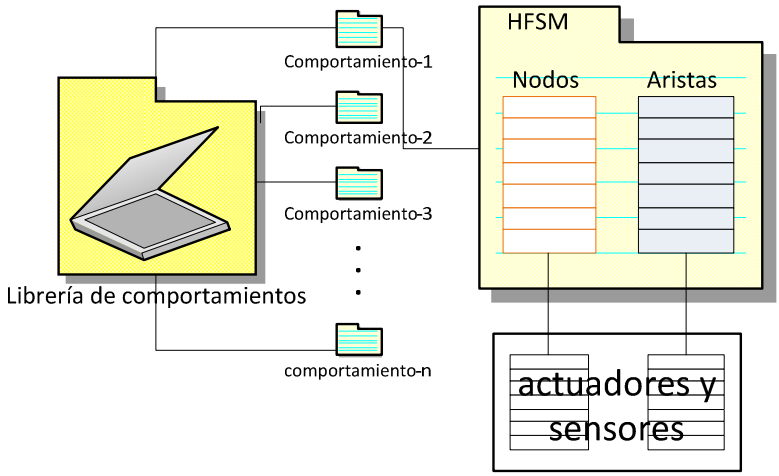

Figura 3-4: Esquema de una librería de comportamientos

3.2 Sistema CBR

Como se ha descrito anteriormente, la herramienta de autoría permite al diseñador crear de manera asistida comportamientos inteligentes que serán incluidos en un entorno de videojuegos. El diseñador puede reutilizar los comportamientos incluidos en el catálogo para modificarlos o para incluirlos dentro de comportamientos más complejos. Una manera de reutilizarlos, es buscando dentro del catálogo, pero esto puede ser una tarea bastante tediosa, si es que no se conoce en detalle, que comportamientos existen dentro del repositorio. Este problema se resuelve, con un mecanismo de búsqueda por coincidencia y recuperación de comportamientos provisto por un sistema CBR, el cual se propone en esta metodología.

El sistema CBR, pretende realizar consultas a una base de casos, en la que cada uno de los casos representa a un comportamiento creado previamente por la misma herramienta de autoría. Por medio de estas consultas se puede obtener una recuperación aproximada de comportamientos anteriores con características parecidas al boceto que se está siendo diseñado. Los comportamientos recuperados podrán ser revisados, comparados y evaluados por el diseñador, una vez revisados, se crea un nuevo comportamiento diferente a partir del resultado, o parte del comportamiento o todo, se incluye dentro de un comportamiento más complejo.

Se permiten dos tipos de consultas contra la base de casos: consultas por funcionalidad y consultas por estructura. En las consultas por funcionalidad se determina una medida de similitud a partir de la lista de propiedades de cada comportamiento, mientras que, en las consultas por estructura se determina la medida de similitud a través de la representación estructurada derivaba de la HFSM de cada comportamiento.

3.2.1 Representación de Comportamientos en un repositorio

La representación de los casos en este dominio de aplicación, (diseño de comportamientos inteligentes en videojuegos), utiliza los 3 tipos de representaciones vistas en la sección 2.5.3.

Cada comportamiento es representado, por un vector atributos-valor que es la lista de propiedades, por una descripción a manera de documento de texto y por una representación jerárquica derivada del modelo de HFMSs. A continuación se describen cada una de las representaciones que componen a un comportamiento, que son utilizadas por el sistema CBR.

Propiedades atributo-valor, conjunto de atributos numéricos que describen distintas funcionalidades de un comportamiento. A partir de la lista de propiedades definida en el modelo de juego (véase en la sección 3.1.1) son heredadas a cada comportamiento. El diseñador debe establecerlas manualmente en cada comportamiento.

Se propone la siguiente estructura para cada atributo-valor:

- El nombre.
- Los valores de umbral, es decir mínimo y máximo, o los valores posibles (en caso de atributos multi-valuados).
- Un valor por defecto.

Se pueden definir tantos atributos como sea necesario. Los atributos escogidos deben ser lo suficientemente representativos como para poder describir los comportamientos, pero no deben ser demasiados ya que deben definirse explícitamente para cada comportamiento del catálogo. Se necesita seleccionar un conjunto de atributos a manera de índices para ser utilizados en el proceso de recuperación del sistema CBR. El proceso se describe en más detalle en la sección de

A continuación, para las búsquedas por similitud se proponen 2 tipos, la búsqueda por similitud funcional y la búsqueda por similitud estructural.

Búsquedas por Similitud

Considere el siguiente ejemplo, en el caso de Soccerbots Tournament, se podría utilizar un atributo tipo rol, que indique el rol que cumple un comportamiento en el equipo de fútbol ("delantero", "defensa", "portero", etc.). El diseñador, por ejemplo, estaría interesado en obtener todos los comportamientos que tengan el tipo de rol "defensa" que debe ser utilizado por el sistema CBR para realizar una búsqueda por similitud de propiedades.

Descripción Textual, contiene una descripción textual acerca de las acciones realizadas por el comportamiento. Esta representación es utilizada por el razonador del sistema CBR para recuperar comportamientos aplicando CBR textual. Además, sirve de ayuda al diseñador para obtener un resumen en lenguaje natural de las acciones del comportamiento. Lo cual facilita la selección al momento de elegir el comportamiento más adecuado del resultado obtenido en la recuperación de comportamientos.

Estructura del comportamiento, en secciones anteriores (3.1.2 y 3.1.3) se describió como se representa un comportamiento desde el punto de vista de la herramienta de autoría. Desde el punto de vista del sistema CBR esta representación se explicará a continuación.

Las relaciones (transiciones) entre estados y la naturaleza jerárquica de las HFSMs utilizadas por el editor para la representación de comportamientos, hacen de la representación estructural la más adecuada para el sistema CBR.

La estructura para cada comportamiento está definida por el grafo dirigido asociado a la HFSM, de manera que, cada nodo del grafo representará a un comportamiento básico, que a su vez, está compuesto por un conjunto de elementos del modelo del juego (sensores y actuadores y propiedades), y cada arista está compuesta por un conjunto de llamadas a sensores del modelo del juego. Por esta razón y para simplificar la representación el grafo dirigido estará compuesto por un conjunto de sensores y actuadores dispuestos de manera jerárquica.

La Figura 3-5 representa la estructura de un comportamiento, desde el punto de vista del sistema CBR este representa el modelo de un caso,

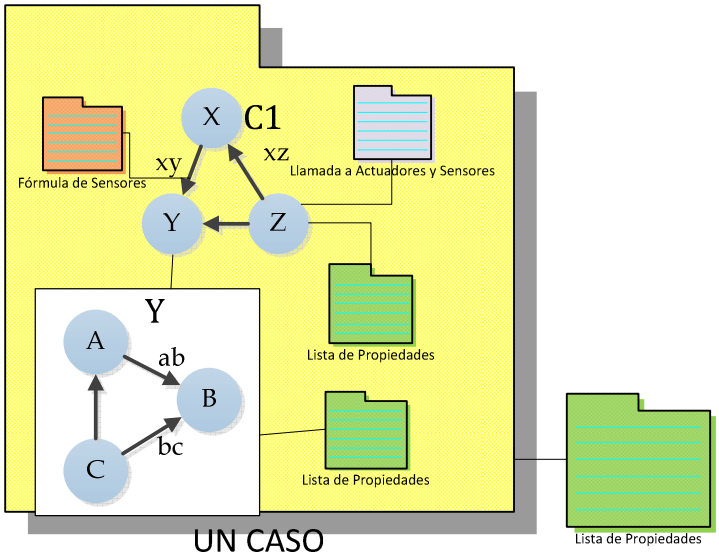

Figura 3-5: Representación de un comportamiento como un caso CBR.

A continuación, para las búsquedas por similitud se proponen 2 tipos, la búsqueda por similitud funcional y la búsqueda por similitud estructural.

3.2.2 Búsquedas por Similitud Funcional

Cada comportamiento (caso) está compuesto por una lista de propiedades (vector de atributos-valor) heredadas del modelo del juego y que describen una funcionalidad. A partir de estas propiedades puede obtenerse una medida de similitud entre dos casos. Puesto que, esta representación de vector de atributos esta soportado en CBR, se aplicarán las técnicas descritas en la sección 2.5.4 revisadas anteriormente.

El proceso de manera general para este tipo de representación es obtener una medida de similitud por cada propiedad que describe al comportamiento, se puede aplicar cualquiera de los métodos de similitud local vistos anteriormente. Finalmente se aplica una operación de agregación y se resumen las medidas de similitud las propiedades en una sola medida de similitud entre casos.

Cada propiedad es un atributo-valor del tipo numérico uni-valuado o multi-valuado, por lo tanto, son aplicables las funciones de similitud local de la

Tabla 2.1. La función para comparar una propiedad del tipo numérico uni-valuado entre dos casos, a (caso nuevo) y b (caso base) es:

$$1 - \frac{|a-b|}{long(I)}, \text{donde } I \text{ es el intervalo de los valores de umbral}$$

Luego de la aplicada la fórmula de similitud local, dependiendo del tipo de dato de cada atributo se obtiene el vector de medida de similitud local, al cual, se debe aplicar una fórmula de la Tabla 2.2 para obtener la medida de similitud entre dos casos. A manera de ejemplo se toma como operador de agregación la media aritmética para definir las siguientes fórmulas. Dado dos vectores de propiedades entre dos comportamientos *A* y *B*:

$$P_A = \{p_{1A}, p_{2A}, p_{3A}, \dots p_{NA}\}$$
$$P_B = \{p_{1B}, p_{2B}, p_{3B}, \dots p_{NB}\}$$

Siendo *N* el número de propiedades que definen a un comportamiento, el vector de similitud local entre *A* y *B* será:

$$SIM_{Local}(A,B) = \{\, sim_{Local}(p_{iA}, p_{iB})\} \;\; ; \;\; i = 1 \dots N$$

A partir de este vector se obtiene la medida de similitud entre dos casos *A* y *B*, siendo:

$$S_L = \{S_{1AB}, S_{2AB}, S_{3AB}, \dots, S_{NAB}\}$$
$$SIM_g(A,B) = \sum_{k=1}^{N} W_k \cdot \mathrm{S_{kAB}} \;\; ; \;\; \sum_{k=1}^{N} W_k = 1$$

La desventaja del método propuesto para obtener una medida de similitud funcional radica en que los valores de las propiedades son ingresadas a criterio del usuario, de manera que, la medida de similitud dependerá de la correcta asignación de las propiedades. Para futuros trabajos se debe considerar algún mecanismo de asignación automática de las propiedades a través de la herramienta.

3.2.3 Búsquedas por Similitud Estructural

El modelo de HFSMs para representar comportamientos es como un conjunto de grafos dirigidos etiquetados, donde cada estado es un nodo y cada transición una arista que corresponde a un modelo estructurado para el sistema CBR. Por consiguiente, se aplicarán las técnicas revisadas en la sección 2.6 para encontrar una medida de similitud y comparar dos comportamientos.

Para poder comparar dos comportamientos necesitamos encontrar la distancia de edición entre los dos grafos que representan a cada comportamiento. Por tanto, debemos establecer la relación entre los objetos de nuestra HFSM de un comportamiento y el grafo dirigido del método para calcular la medida de similitud estructural, que ya ha sido revisado.

La relación más evidente es que una máquina de estados jerárquica corresponde directamente a un grafo dirigido, resta definir la relación entre las etiquetas. En nuestro gráfico tenemos dos tipos de nodos, el nodo atómico que se relaciona a un nodo del grafo y su descripción o nombre se relaciona con la etiqueta del nodo, en el caso del nodo compuesto, es necesario convertir su estructura de HFSM independiente en una jerarquía a partir de su nodo inicial y del nodo anterior a este. La transformación de un nodo compuesto la hemos llamado proceso de aplanamiento. Es evidente que las transiciones son las que se multiplican dependiendo del número de nodos del nodo compuesto y su jerarquía. En otras palabras el proceso de aplanamiento convierte una HFSM en una FSM. Cada transición está relacionada con una arista etiquetada, a diferencia de los nodos, su descripción no corresponde a la etiqueta, en su lugar se establece una relación con el conjunto de sensores.

Máquina de estados finitos jerárquica	Representación de grafo dirigido etiquetado
Nodo atómico	Nodo etiquetado
Descripción de un nodo atómico	Etiqueta de un nodo
Nodo compuesto	Conjunto de nodos jerarquizado
Transiciones	Aristas dirigidas
Fórmula lógica de sensores	Conjunto de etiquetas en aristas

Tabla 3.1: Relación entre objetos de una HFSM y un grafo dirigido

Establecida la relación entre las dos representaciones, se puede aplicar la propuesta descrita en la sección "*Una aproximación heurística para el cálculo de similitud*" para obtener una medida de similitud entre dos comportamientos. A manera de algoritmo los pasos a seguir serán:

Sean dos comportamientos A y B, que están representados por el grafo g_A y el grafo g_B respectivamente,

- Aplicar el proceso de aplanamiento a cada comportamiento.

- Crear el conjunto de correspondencias para todos los nodos del grafo g_A con cada nodo del grafo g_B.
- Asignar a cada elemento del conjunto, el costo de las operaciones de edición para transformar los nodos correspondientes al par (n_A, n_B) de cada elemento.
- A partir de este conjunto de correspondencia de nodos M_N, y de la fórmula de la sección 2.6.3, se busca encontrar la correspondencia de nodos con menor coste.

$$c(M_N) = \sum_{\forall (n,n') \in M_N} c((n, n'))$$

- Se resuelve con resuelve con el algoritmo Húngaro haciendo una equivalencia con el problema de asignación y se obtiene M_{N-best}
- Se adicionan los costos relacionados a las transiciones adyacentes de cada nodo del conjunto M_{N-best} y se obtiene una medida aproximada para la distancia de edición entre dos comportamientos $dist(g_A\ , g_B\ )$.
- Finalmente se calcula la medida similitud entre dos casos con la siguiente fórmula

$$sim(A\ , B) = \frac{1}{1 + dist(g_A\ , g_B)}$$

Utilizando la novedosa propuesta heurística se utilizan dos funciones para el cálculo del costo de transformación entre dos nodos.

- $Identidad: id(n_{Origen}\ , n_{Destino})$, retorna 0, si el nodo origen tiene la misma etiqueta que el nodo destino y 1 si las etiquetas son distintas.
- $Identidad_{completa}: id(n_{Origen}\ , n_{Destino})$, se comparan dos nodos usando la función $Identidad$ y se adiciona la medida de similitud de las aristas que llegan y salen del nodo.

Para calcular el costo de las operación de edición sobre las aristas y que este resultado sea sumado al costo de similitud de los nodos, se utiliza la siguiente función, que como hemos dicho anteriormente utiliza, en vez de la etiqueta, los sensores de la arista.

$Sean\ dos\ aristas, E_A\ y\ \ E_B\ , se\ define\ la\ función\ de\ similitud\ entre\ aristas\ como,$

$$SIM_{Aristas}(E_A, E_B) = \frac{|sensores(E_A) \cap sensores(E_B)|}{|sensores(E_A) \cup sensores(E_B)|}$$

$$Donde\ sensores(E_A) = \{s_{1A}, s_{2A}, s_{3A}, \dots, s_{kA}\}, \qquad k\ sensores\ utilizados\ en\ la\ arista\ E_A$$

El cálculo de la medida de similitud, a través de la estructura de las HFSMs, es lo que hace posible la edición de comportamientos basados en bocetos, la herramienta propuesta utiliza el repositorio de comportamientos para encontrar uno o varios comportamientos de similar estructura al boceto que se dibuja en el editor, y sin tener que diseñar completamente su funcionalidad encontrar el que mejor se adapte al diseño deseado.

3.2.4 Reutilización, Revisión y Aprendizaje

El proceso de adaptación, es realizado por el diseñador del comportamiento utilizando la herramienta de autoría. Partiendo del comportamiento recuperado, el diseñador debe modificarlo para conseguir el resultado deseado. En esta metodología no se propone realizar la adaptación de manera automática, pero se ofrece el soporte al diseñador para llevarla a cabo.

Para dar al diseñador una retroalimentación adecuada, se remarcan los estados candidatos a ser modificados. Para determinar cuáles son los nodos candidatos, el sistema evalúa nodo a nodo del comportamiento y verifica qué atributos son diferentes con relación al caso consultado. Aquellos nodos que sean diferentes son los candidatos a ser adaptados. Además otra facilidad que se propone es que la herramienta de autoría permita comparar ambos comportamientos de manera visual, (el recuperado y la consulta) con el objetivo de dar una vista general al diseñador.

El tipo de adaptación a ser empleado por el diseñador es transformacional, ya que se transforma el comportamiento recuperado para que se ajuste a las características del comportamiento deseado. En otras palabras, se realiza una adaptación por ajuste de atributos, 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.

Dado el gráfico de la Figura 3-6, se describe a manera de ejemplo el siguiente proceso de adaptación, El nodo "*Cubrir portería*" está relacionado con los atributos "*Defensa=1*" y "*Movilidad=4*", y el nodo "*Conducir pelota a portería*" con "*Movilidad=4*" y "*Ataque=5*". Si el diseñador está buscando un comportamiento con un atributo "*Ataque=3"*, el sistema indicará que el nodo que se debe modificar es "*Conducir pelota a portería"*, ya que es el nodo que tiene definido este atributo, pero con un valor distinto al buscado.

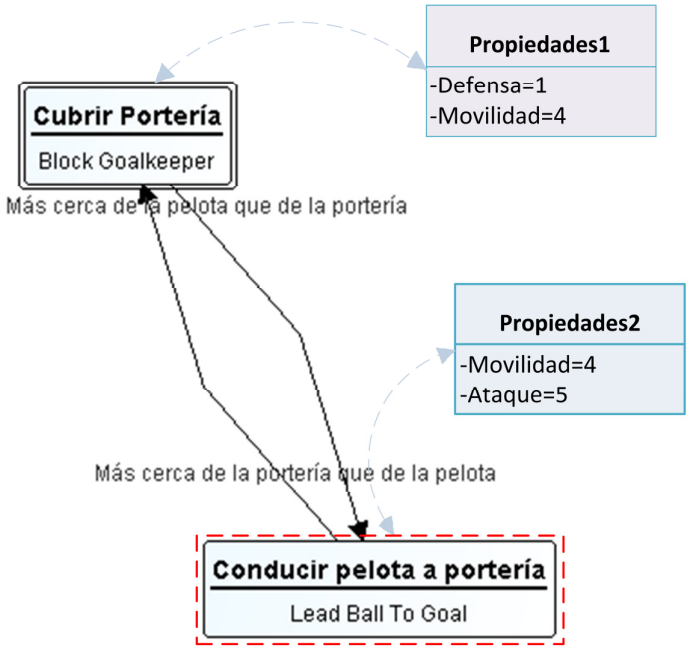

Figura 3-6: Boceto de un comportamiento con propiedades asociadas en sus nodos

El aprendizaje, es en gran medida responsabilidad del diseñador, que del mismo sistema CBR. Por un lado, El diseñador es quien decide qué comportamientos son almacenados en el repositorio y además, también es responsabilidad del diseñador asignar valores a los atributos de cada comportamiento de manera que sean lo bastante representativos y las descripciones que definen la funcionalidad del cada comportamiento o sub-comportamiento.

3.3 Generación de código

Es el componente, que se utiliza como interface entre la herramienta de autoría y el motor de videojuego, crea el código necesario para hacer el diseño del comportamiento operacional dentro de un entorno de videojuego o de simulación (véase la Figura 3-1).

Este componente entra en juego en la implementación de un comportamiento dentro de un entorno de simulación, y como su nombre lo indica, su función es generar un fichero ó un conjunto de ficheros de código fuente, sean estos compilables o interpretables, destinados a ser ejecutados dentro del entorno de simulación.

El código fuente generado por este componente depende directamente de la plataforma de programación a la que va destinada, como ejemplo: El motor de simulación de

Soccerbots ejecuta comportamientos escritos en Java como lenguaje de programación. Esta dependencia hace que la implementación de este componente sea rígida, en cuanto a la selección del lenguaje de programación o de etiquetas que se necesite para hacer operacional un comportamiento.

Por otro lado, el conjunto de actuadores y sensores definidos en el modelo del juego y utilizado para la construcción de comportamientos, hace posible abstraer el conjunto de llamadas a las funciones que conforman la API del motor del juego. De este modo, el generador de código no depende de la implementación de las funciones de la API, lo cual, hace de esta herramienta de autoría extensible para cualquier motor de videojuego o simulación.

Existen un sin número de propuestas para transformar una representación de HFSMs en código ejecutable, pero hemos optado por el de una pila de ejecución por su sencillez y efectividad. El funcionamiento de manera general es que se recorre todos los nodos y aristas del comportamiento, traduciendo su contenido en llamadas a sensores o a actuadores al lenguaje de destino.

Un comportamiento creado por la herramienta de autoría puede tener un diseño no determinista, heredado de las HFSMs, ya que desde un nodo pueden existir dos transiciones que se evalúen a cierto al mismo tiempo. Para sobrellevar esto, el generador de código puede establecer un orden de precedencia basado en el orden en que se han definido las aristas, o dar la posibilidad al diseñador a establecer este orden de precedencia. Al editar un comportamiento es importante tener en cuenta cómo se trata el no determinismo por el generador de código, de lo contrario, la ejecución del comportamiento podría no mostrar los resultados deseados en el diseño.

A continuación el proceso de generación de código propuesto para el generador de código.

- Debido a que cada comportamiento, está compuesto por otros sub-comportamientos, se utiliza una ventana de ejecución por cada comportamiento, que guarde el estado actual en el que se encuentra.
- Una única ventana de ejecución será ejecutada por cada ciclo del juego, de manera que existirá, una ventana de ejecución activa dentro de una pila de ventanas de ejecución (véase).
- La ventana de ejecución, contiene el código ejecutable asociado a cada estado, es decir, el bloque de código correspondiente a cada comportamiento básico y las llamadas a los sensores correspondientes a cada transición.

- Por cada ventana de ejecución, se ejecuta el código del comportamiento básico del estado actual, y luego se evalúan las condiciones de las transiciones.
- Las condiciones son evaluadas por el orden definido en las transiciones, cuando se evalúa una condición a cierto, se cambia el contenido de la ventana de ejecución por el código ejecutable correspondiente al siguiente estado.
- Cuando el motor de ejecución evalúa una condición a cierto y determina que el estado siguiente es un comportamiento compuesto, creará una nueva ventana de ejecución con las acciones antes descrita y la coloca en el inicio de la pila de ejecución.

La Figura 3-7, representa a la pila de ejecución creada por el generador de código.

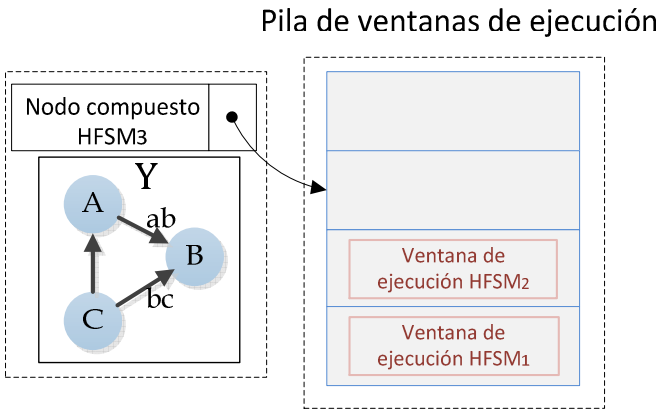

Figura 3-7: Pila de ejecución creada por el generador de código

4 Edición y generación de código de comportamientos

En el capítulo anterior se describieron las técnicas de trabajo que permiten a los diseñadores de comportamientos y a los programadores de IA trabajar conjuntamente de una forma más colaborativa, de manera que, se máxime su trabajo y se disminuya el tiempo de producción e implementación de comportamientos de agentes inteligentes en un videojuego. En este capítulo se revisará el funcionamiento de la herramienta de autoría de comportamientos para NPC desarrollada, que soporta la metodología descrita en el capítulo de Diseño basado en bocetos.

4.1 La Herramienta de autoría

El objetivo principal de la herramienta de autoría es ayudar al diseñador en la construcción de comportamientos de NPC, para lo cual, se han utilizado las técnicas descritas en el capítulo 0 para cumplir con este objetivo.

La herramienta está compuesta de una serie de componentes que soportan la edición de comportamientos y que serán descritos desde un punto de vista funcional. Entre las principales tareas que se pueden realizar en esta herramienta están:

- Gestión de repositorios de comportamientos, permite realizar las operaciones básicas de gestión de repositorios (importar, navegar, crear, borrar, etc.). Cada repositorio es una librería de comportamientos destinado a un motor de videojuego en particular. El paradigma utilizado en la gestión de repositorios es similar al paradigma de carpeta de proyectos utilizado por Eclipse.
- Gestión de librería de comportamientos básicos, permite la gestión de los bloques de código que representan a las acciones básicas de un NPC, destinados a ser utilizados en la edición de comportamientos más complejos.
- Edición de comportamientos, permite realizar las operaciones básicas de gestión de nodos, transiciones, condiciones, y demás, todas relacionadas a la edición de comportamientos basados en bocetos de HFSMs
- Generación de código, permite producir el código ejecutable en un motor de juego en particular, en esta primera versión se genera código en Java dirigido al motor de juego SBTournament.

La Figura 4-1 muestra una captura de pantalla del aspecto de la interface de la herramienta desarrollada, en la parte izquierda se encuentra la herramienta de gestión de repositorios, la parte central es el editor de comportamientos, y la parte inferior de la ventana se encuentra la herramienta de gestión de comportamientos básicos.

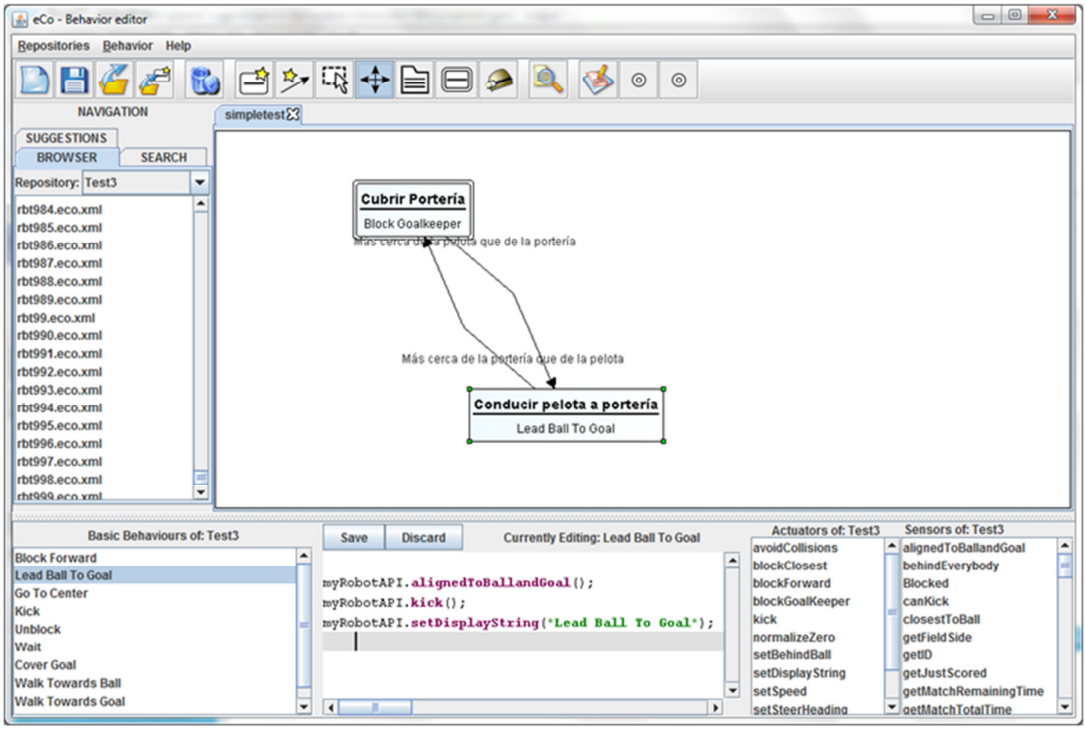

Figura 4-1: Captura de la ventana principal de la herramienta de autoría eCo-Behavior.

A continuación, se describirán las funcionalidades básicas de la herramienta de autoría con un enfoque práctico.

4.1.1 Gestión de repositorios

Toda la perspectiva de edición de comportamientos va a ser orientada al uso de un repositorio, la Figura 4-2 muestra el panel de gestión de repositorios. Esta herramienta permite realizar las siguientes tareas de gestión sobre los repositorios de comportamientos:

- Crear un nuevo repositorio, antes de empezar la edición de cualquier boceto de un comportamiento es necesario crear un repositorio. Un repositorio está compuesto por la siguiente estructura: una librería de comportamientos básicos, una librería de comportamientos creados dentro del editor, y un modelo del juego al cual van destinados los comportamientos a ser generados por la herramienta.

- Cargar un repositorio existente, permite cargar la estructura de un repositorio existente, dentro del ambiente de la herramienta.
- Navegar a través del repositorio: permite inspeccionar la colección de los comportamientos contenidos dentro del repositorio.
- Añadir, modificar o eliminar un comportamiento del repositorio: permite al diseñador ir gestionar nuevos bocetos de comportamientos, y al programador de IA gestionar los comportamientos básicos.
- Buscar sobre el repositorio, permite las búsquedas por nombre de los comportamientos contenidos dentro del repositorio.

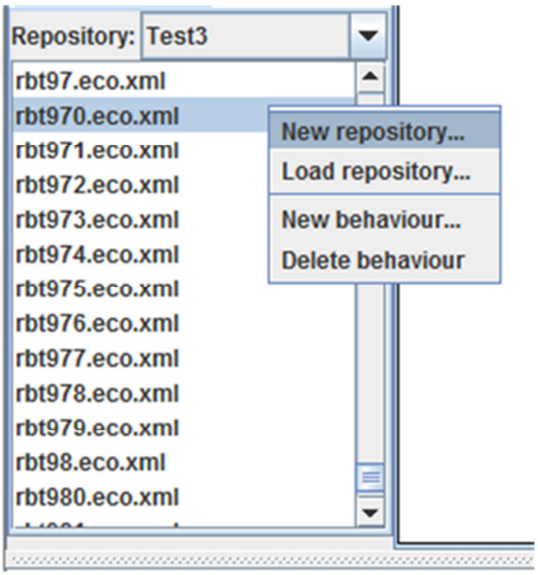

Figura 4-2: Herramienta de gestión de repositorios.

Corresponde al panel izquierdo de la herramienta y contiene:

Lista desplegable de repositorios, Lista de comportamientos contenidos

Cada repositorio contiene una estructura de archivos y carpetas que son utilizados por la herramienta como almacenamiento de datos. Los archivos utilizados son en su mayoría archivos de configuración xml, en la sección de apéndices encontramos a manera de ejemplo la representación de cada uno de ellos. La Figura 4-3 muestra la estructura de archivos y directorios utilizada.

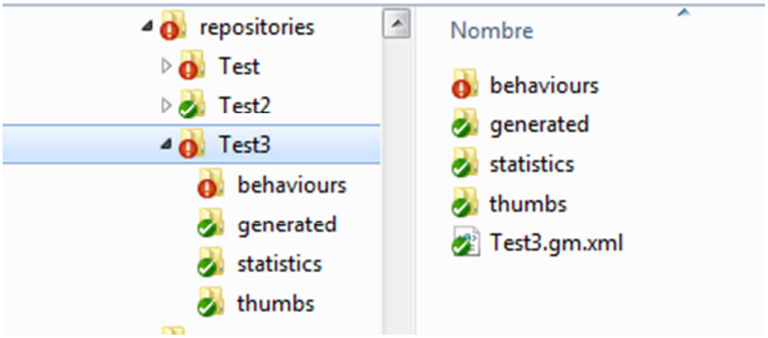

Figura 4-3: Estructura de archivos y directorios de un repositorio

4.1.2 Gestión de librería de comportamientos básicos

Esta herramienta de autoría contiene un componente de gestión de comportamientos básicos, provisto por un editor de código que permite al programador de IA, codificar las acciones básicas de un NPC, y que va a ser utilizado por el diseñador para construir los bocetos de un comportamiento. Esta herramienta permite realizar las siguientes tareas de gestión sobre la librería de comportamientos básicos.

- Crear, modificar y eliminar un comportamiento básico de la librería de comportamientos básicos.
- Editar el bloque de código que corresponde al comportamiento básico utilizando una GUI de edición de texto correspondiente al lenguaje de programación al cual vaya destinado la generación de código.
- Contener el diccionario de las llamadas a los sensores y actuadores del modelo del juego correspondiente al repositorio.
- Permitir navegar a través del conjunto de sensores y actuadores a través de listas desplegables.
- Permitir la selección de sensores y actuadores para que sean incluidos dentro del editor de código, facilitando al programador el conjunto de llamadas.

Los comportamientos básicos son almacenados también en el repositorio y por lo tanto comparten el mismo modelo del juego. Las Figura 4-4 y Figura 4-5, muestran la herramienta de gestión de comportamientos básicos que corresponde al panel inferior de la herramienta de autoría.

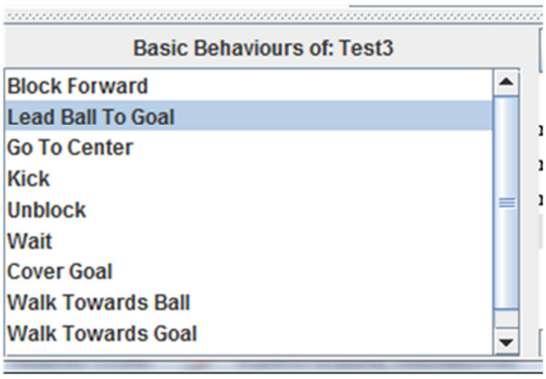

Figura 4-4: Panel de la lista de comportamientos básicos. (Ejemplo para SBTournament)

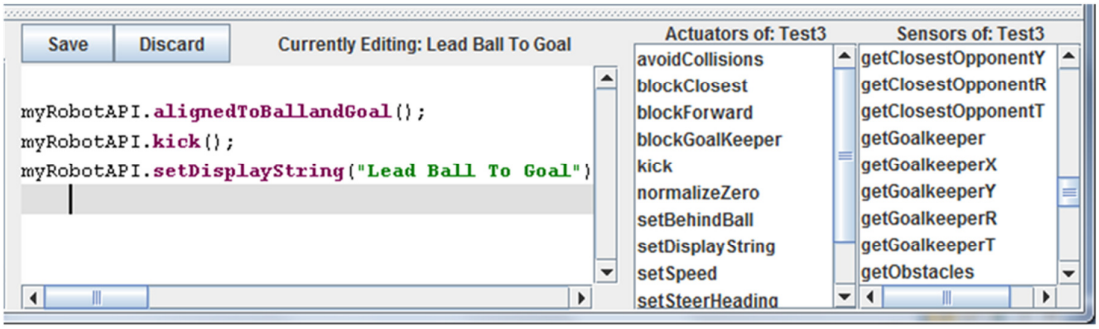

Figura 4-5: Editor de código de un comportamiento básico. (Ejemplo para SBTournament)

Es importante recordar que el bloque de código fuente que representa a un comportamiento básico es utilizado por el componente de generación de código y son implementados como funciones. El nombre del comportamiento básico será utilizado como nombre de la función, por consiguiente, el nombre debe ser adecuado a las descripciones del lenguaje de programación destino. Por otro lado, los comportamientos básicos también son utilizados para ser asignados a los nodos de un comportamiento, de modo que, el nombre debe ser descriptivo y único.

4.1.3 Edición de comportamientos

Siendo el componente central de la herramienta de autoría, realiza la mayoría de las funciones del sistema. En esta sección solo se describirán las tareas más relevantes.

Antes de describir las tareas que se realizan en la edición de comportamientos se describe de forma general los tipos de elementos y sus propiedades, utilizados para la construcción de bocetos.

Tipos de nodos, existen tres tipos de nodos, el editor permitirá seleccionar entre los siguientes tipos, compuesto, atómico y sin comportamiento. La representación de los diferentes tipos se muestran en la Figura 4-6 en el orden descrito.

Figura 4-6: Representación de los tipos de nodo, (1 compuesto, 2 atómico, 3 no definido)

Nodo compuesto, representado por un recuadro con doble línea (ver Figura 4-6) contiene un sub-comportamiento y consta de una etiqueta "*Composite*" seguido del nombre.

Nodo atómico, representado por un recuadro con una línea simple, contiene un comportamiento básico y consta de una etiqueta que describe al comportamiento básico asociado. (ver Figura 4-6, 2.-comportamieto básico "*Kick*").

Conectores, representan a las aristas que unen a los estados, es decir, las transiciones. Representados por una flecha que nace del estado de partida y apunta al estado destino.

A continuación se describen las principales funciones de la edición de comportamientos.

Gestión de nodos, permite:

- Editar un boceto con las operaciones básicas de nodos, crear, borrar ó modificar el contenido de cada nodo.
- Establecer las propiedades de cada nodo. Dependerá del tipo de nodo seleccionado las propiedades a ser establecidas dentro del nodo (véase Figura 4-7, Figura 4-8).
- Seleccionar y posicionar los nodos dentro del boceto.

- Inspeccionar los nodos compuestos, y permitir la edición de un sub-comportamiento en otro editor de bocetos.
- Navegar a través de toda la jerarquía de una HFSM por medio de un panel con múltiples editores y un control Breadcump. En todo momento de la edición de un comportamiento el diseñador sabe en qué parte de la jerarquía se encuentra a través del control (véase Figura 4-9, Figura 4-10).

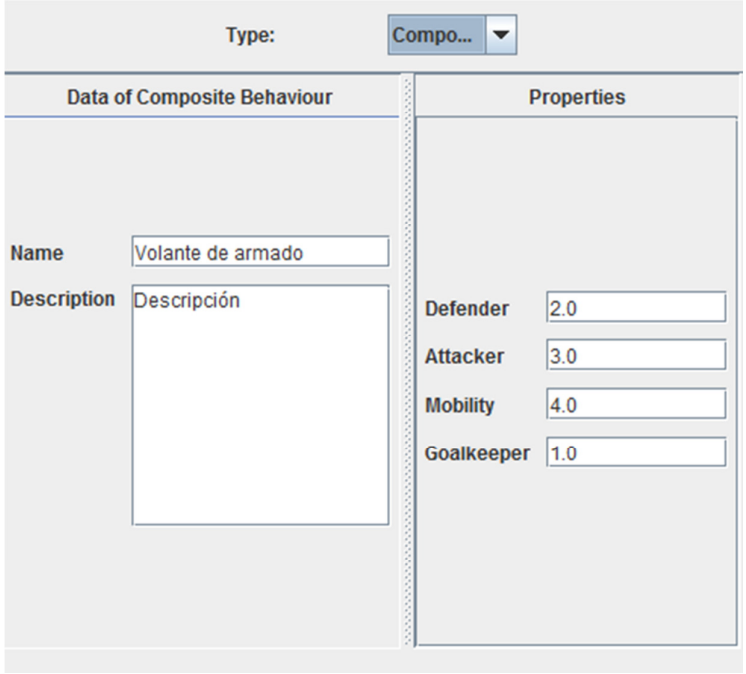

Figura 4-7: Propiedades de un nodo compuesto

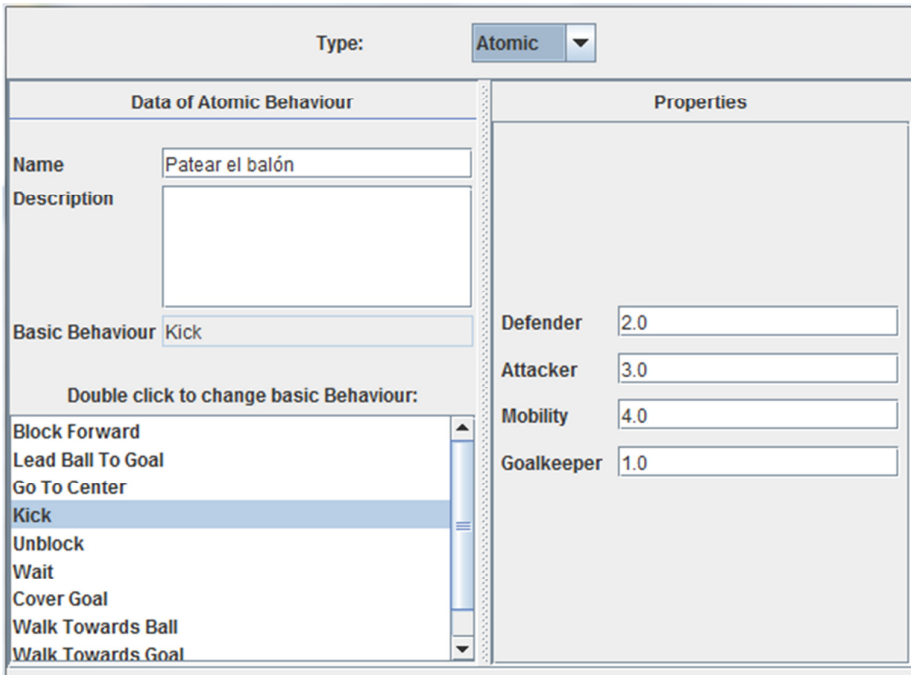

Figura 4-8: Propiedades un nodo atómico

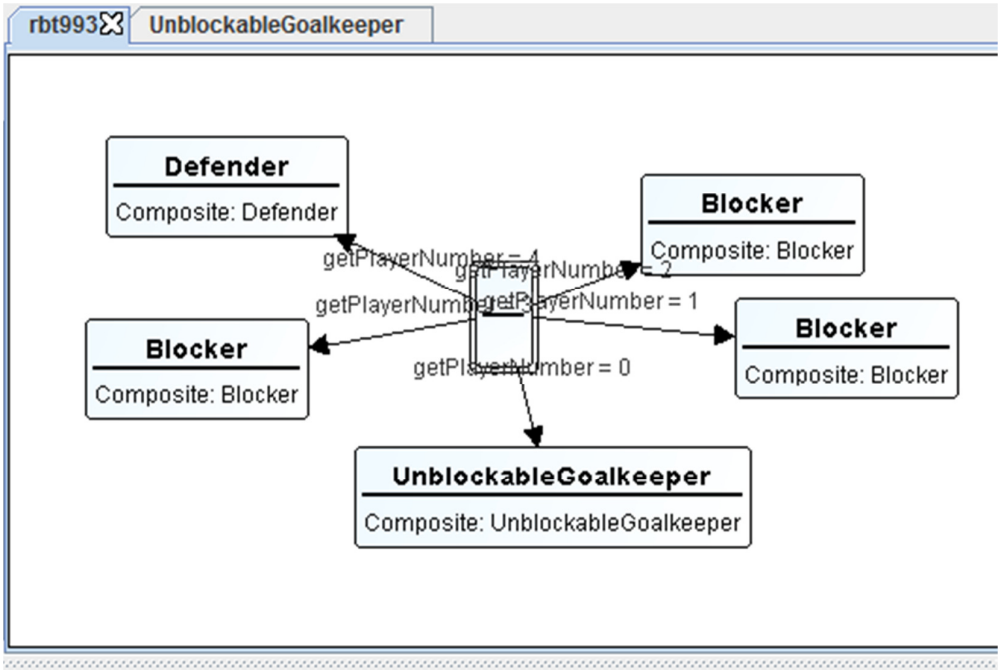

Figura 4-9: Ejemplo de navegación e inspección proporcionado por el editor. Nº1.

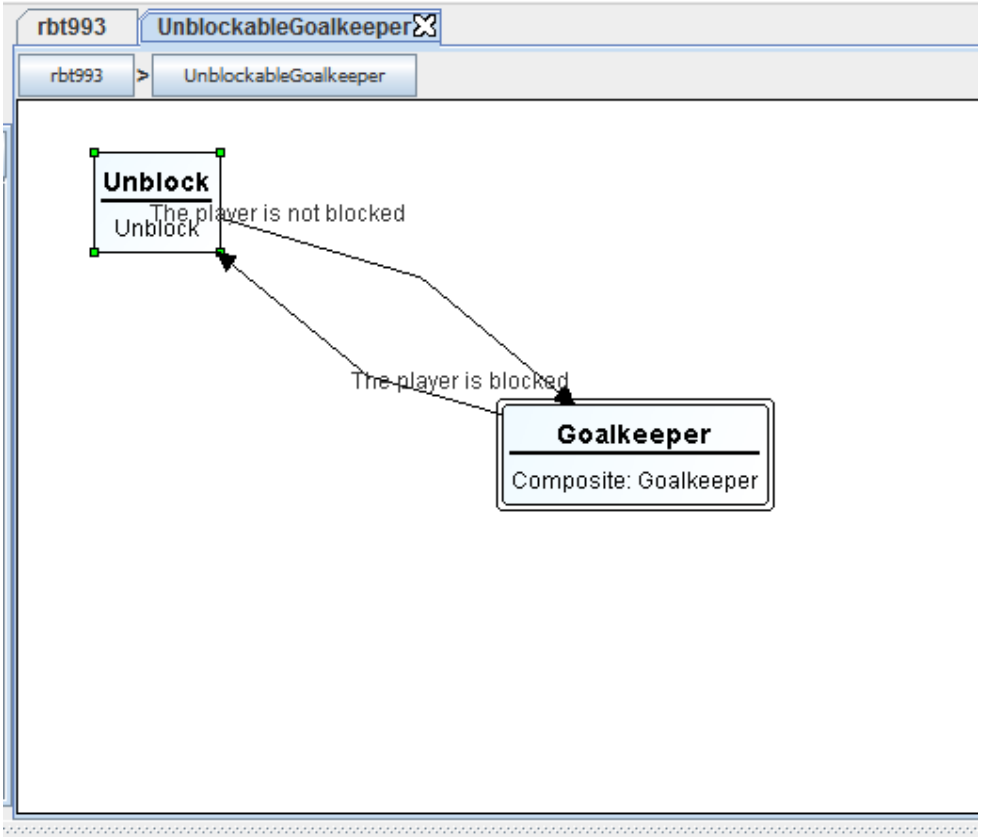

Figura 4-10: Ejemplo de navegación e inspección proporcionado por el editor. Nº2.

Gestión de conectores, los conectores pueden tener asociadas fórmulas lógicas cuando se evalúan a cierto se produce la transición al siguiente estado. Cada fórmula está compuesta por una disyunción de conjunciones lógicas del tipo sensor <comparador> valor, y un nombre descriptivo definido por el diseñador. La gestión de conectores permite entre otras cosas:

- Editar un boceto con las operaciones básicas de aristas, crear, borrar ó modificar el contenido de una arista que representa una transición.
- Establecer las propiedades de cada transición. Indicando el nombre las disyunciones de conjunciones que componen la fórmula lógica.
- Establecer el nodo origen y el nodo destino dentro del boceto.

La Figura 4-11 muestra las propiedades a ser establecidas en una transición, a manera de ejemplo el modelo de juego a ser utilizado es el de SBTournament, el cual se puede apreciar en la lista de sensores.

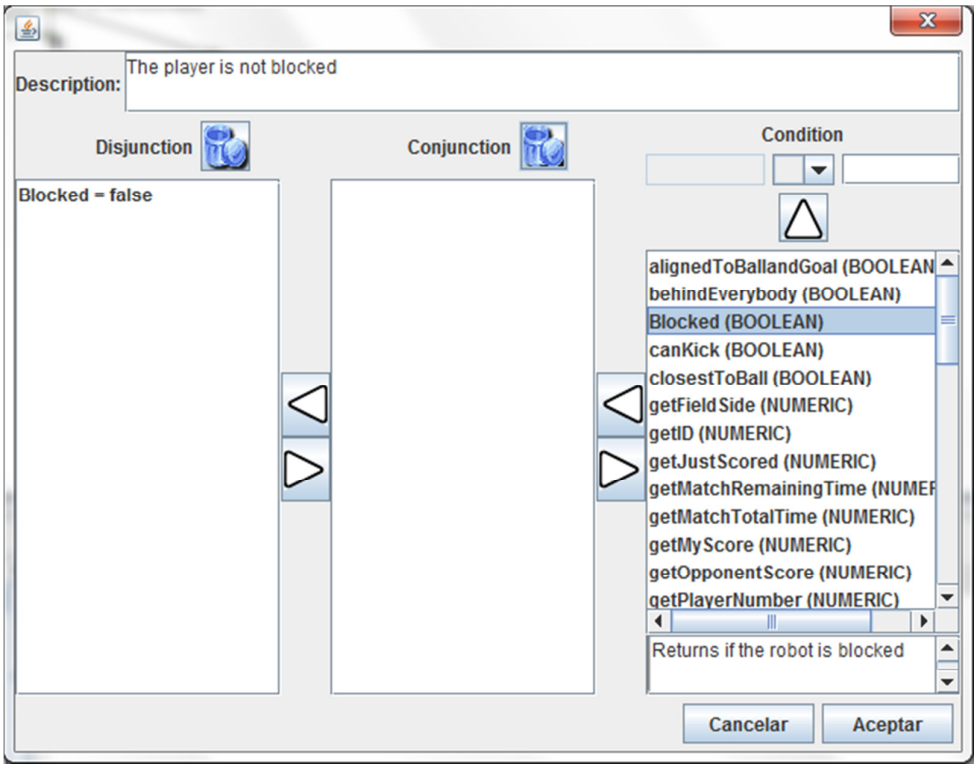

Figura 4-11: Dialogo de propiedades de una transición.

4.1.4 Generación de código

La herramienta de autoría cuenta con un componente de generación de código, que como su nombre lo indica, es el encargado de producir el código a ser ejecutado dentro del motor del juego.

Este componente es utilizado en la última etapa de desarrollo de un comportamiento para un NPC. La función exclusiva de este, es generar el código fuente en un lenguaje de programación soportado por el motor de simulación o de videojuego al que vaya destinado, este código se construye a partir del boceto del comportamiento diseñado en la herramienta de autoría.

En la versión actual de la herramienta únicamente se soporta el motor de simulación correspondiente a SBTournament, el cual tiene como lenguaje de programación destino Java. Al ser dependiente del motor al que vaya destinado este componente debe ser escrito en Java para adaptarse como un componente generador dentro de la herramienta de autoría, es decir que, para soportar un nuevo marco de destino (motor de simulación destino) se debe escribir un nuevo generador de código que debe ser incluido dentro del código fuente del sistema,

debe ser compilarlo y luego generar una nueva versión de la herramienta. Esto puede ser considerado como una debilidad de la herramienta y con posibilidades de mejoras a futuro.

4.2 Integración con el motor de simulación

Si bien la herramienta de autoría hasta ahora descrita, puede integrarse a cualquier motor de simulación o de videojuego, en esta sección se introducirá de manera particular la integración con el motor de simulación SBTournament,

"SBTournament" es un entorno de simulación mejorado basado en el motor de simulación de "*SoccerBots"* que simula la dinámica y las dimensiones de un partido reglamentario de la "*Robocup"*, que es la copa mundial de fútbol para robots de pequeño tamaño (26; 27).

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", es decir 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 4-12 se muestra una captura de pantalla del motor de simulación utilizado para las pruebas y parte de la interface de configuración que necesita ser establecida para simular un torneo virtual de SoccerBots.

Para poder utilizar el editor de comportamientos con SBTournament, es necesario definir el modelo de juego y crear, al menos, un generador de código. El modelo de juego visto en la sección 3.1.1, está conformado por dos partes: la interfaz de juego, que declara el conjunto de sensores y actuadores a ser utilizados como interface de comunicación entre el simulador y la implementación del comportamiento; y las propiedades que describen la funcionalidad de cada comportamiento.

Luego del análisis realizado a las funciones que corresponden a la API de SBTournament se han obtenido el conjunto de sensores y actuadores descritos en la tabla del Apéndice B.

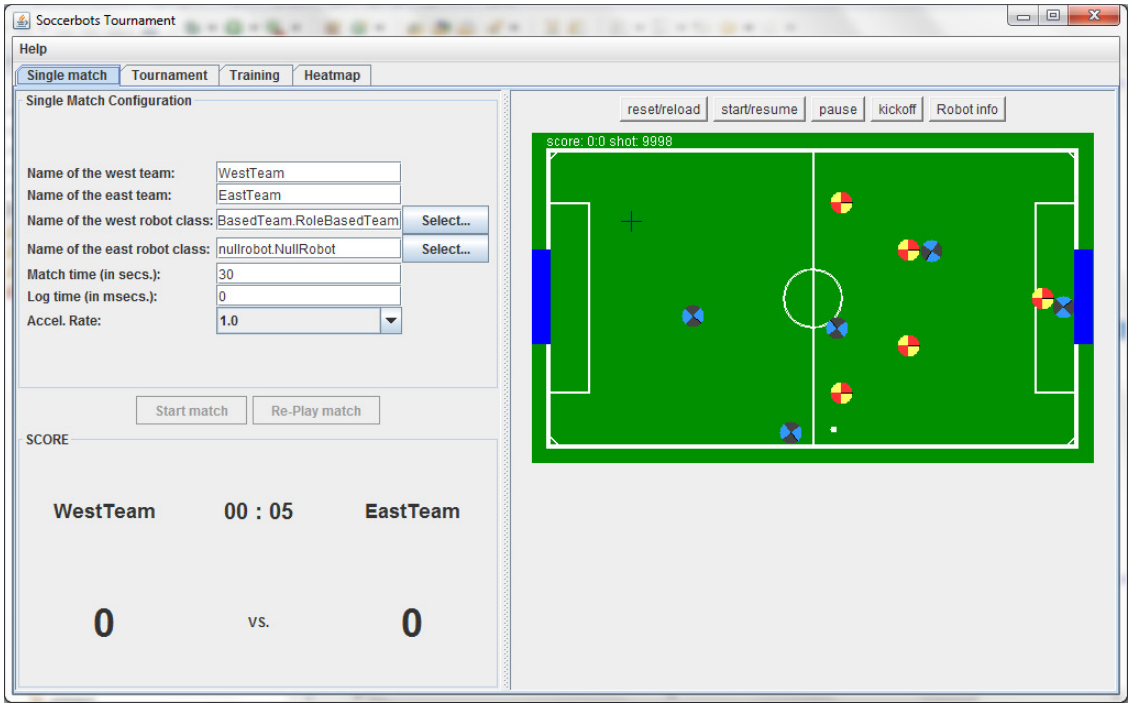

Figura 4-12: Captura de pantalla del simulador SBTournament utilizado.

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 su 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 componente de Generación de código, debe generar el código Java que corresponda a la implementación de la HFSM de un comportamiento diseñado por la herramienta de autoría. El proceso utilizado para la generación del código es basado en la técnica descrita en la sección 3.3 y para su implementación se utiliza una tabla que almacena

cada ventana de ejecución. Una ventana de ejecución en esta implementación está compuesta por:

- Un puntero a la función que ejecuta el bloque de código del comportamiento básico asociado al nodo.
- Una lista con las aristas que salen del nodo.
- Por cada arista se almacena, un puntero a la función que es construida por el generador de código utilizando la formula lógica y las preposiciones lógicas con las llamadas a los sensores, que componen a la arista..
- Por cada arista, se tiene un puntero al estado siguiente en el caso de que la condición se evalúe a cierto.

En Java no se permite almacenar punteros a funciones directamente, por lo cual, se utilizan las técnicas de *Java Reflection* para invocar a los métodos creados por el generador, sin necesidad de utilizar punteros. El paquete utilizado para tal cometido, es java.lang.reflect.

Para implementar la jerarquía de la HFSM se incluye una pila, que contendrá en cada paso de ejecución, la ventana de ejecución descrita en el párrafo anterior que corresponda al nodo actual de cada uno de los comportamientos y sub-comportamientos activos. Además de las operaciones habituales en una pila (añadir, 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 Apéndice C se muestra el código correspondiente a la ejecución de un comportamiento y que ha sido generado por la herramienta de autoría.

En el método "*OnConfigure*" se inicia antes de empezar la simulación, en este método se genera la tabla de estados e introduce en la pila el estado inicial. Durante la generación de la tabla de estados se generan los métodos correspondientes a las condiciones y las acciones de los comportamientos básicos.

El método "*takeStep*" es llamado por el entorno de simulación cada paso de ejecución del bucle de la simulación. En este método se ejecutan, en primer lugar, las acciones asociadas al nodo actual, el cual se encuentra en la cima de la pila. Luego se evalúan las condiciones asociadas a las transiciones de los nodos de la pila. Esta evaluación empieza por la parte más baja de la pila 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.

```
//Create the table of State. It has been generated by code
generator
	private void GenerateStates()
	{
		states = new Vector<State>();
		State e;
		// State 1 : Node 0
		e = new State();
		e.transitions = new Transition[1];
		e.transitions[0] = new Transition();
		e.transitions[0].condition = "cond_0_0";
		e.transitions[0].state = 1; //Go to node 1
		e.actions = "acc_0";
		states.add(e);

		// State 2 : Node 1
		e = new State();
		e.transitions = new Transition[1];
		e.transitions[0] = new Transition();
		e.transitions[0].condition = "cond_1_0";
		e.transitions[0].state = 0; //Go to node 0
		e.actions = "acc_1";
		states.add(e);

		}
```

Figura 4-13: Código fuente generado para la contrucción de tabla de estados y sus transiciones

5 Evaluación

La construcción de comportamientos de NPC puede ser una tarea no trivial y que demanda gran cantidad de tiempo a la hora de implementarlos. En este capítulo se describirá paso a paso con ejemplos prácticos la creación de comportamientos sin ninguna herramienta que los soporte. Luego utilizaremos la herramienta de autoría desarrollada con el fin de demostrar el tiempo que esta ahorra y el trabajo que se máxima a la hora de trabajar en conjunto diseñadores de comportamientos y programadores de IA.

En las primeras interacciones con el editor, o 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.

Los bloques de código incluidos dentro de la librería de comportamientos son los primeros elementos a ser construidos. El programador de IA junto con el Diseñador, deben acordar que bloques de acciones básicas deberán ser incluidos para el motor del juego destino. Por tal motivo, es conveniente partir de una base de casos básicos (comportamientos básicos) que contenga algunos casos predefinidos del modelo del juego de SBTournament y que mostraremos en detalle en esta sección. La estructura del modelo del juego que será utilizada para los ejemplos y en la construcción de la librería de comportamientos se muestra en el Apéndice A.

El siguiente paso es la construcción de la base de casos, inicialmente no contamos con una buena cantidad de comportamientos, pero se incluirán los comportamientos necesarios que nos permitan inferir el tiempo en la construcción de nuevos comportamientos. Luego de construidos los comportamientos de ejemplo, estos serán comparados con el código creado sin asistencia de la herramienta, para encontrar una medida de comparación que demuestre la utilidad de la herramienta.

5.1 Implementación de un comportamiento sin asistencia de una herramienta

Con la finalidad de crear un comportamiento sin asistencia de una herramienta, se ha elegido una implementación de Millington para las HFSMs. Existen un sin número de estructuras para máquinas de estados finitos, luego de una serie de análisis el autor ha elegido la que soporta más flexibilidad a la hora de incluir el código y en su depuración (28).

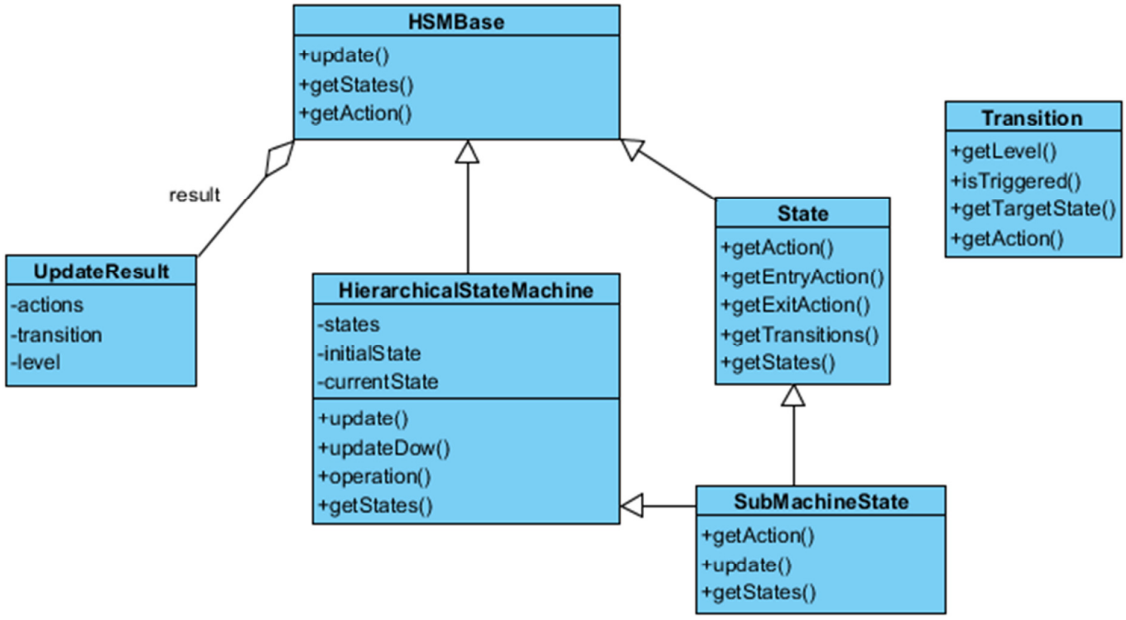

Figura 5-1: Diagrama UML de la HFSM utilizado para implementar los comportamientos

Como el autor lo describe, con la implementación de este modelo se puede incluir cualquier tipo de código, la máquina de estados se preocupa por la ejecución del conjunto estados y registra el estado actual, junto con cada estado una serie de transiciones son mantenidas. Cada transición es una interface genérica que puede implementar las condiciones adecuadas. En cada iteración la máquina de estados llama a la implementación de una función *update()*, esta verifica si hay alguna transición para el estado actual, luego se ejecutan las acciones asociadas al estado activo, si la transición ha sido registrada se realiza.

El diagrama de la Figura 5-1 muestra el modelo de clases utilizado en la implementación descrita en la

Tabla 2.1. Todas las clases excepto la clase Transition hereda de HSMBase, la cual simplifica el algoritmo con funciones que permitan tratar la jerarquía de una misma manera.

Clase	Descripción	Líneas de código
State: Interface	getAction retorna la lista de acciones a llevarse a cabo. getEntryAction getExitAction solo son llamados cuando se ingresa o se sale de un estado. En cada ejecución el estado activo llama a getAction. Y getTransitions obtiene las transiciones	aprox. 11
Transition	isTriggered devuelve cierto si la transición debe ser producida, getTargetState, retorna el estado a ser alcanzado y getAction, la lista de acciones, getLevel, el nivel de la transición	aprox. 8
Condition	test, evalúa un conjunto de fórmulas lógicas, las líneas de código dependerán del número de condiciones	
Hierarchical StateMachine	Clase que implementa la ejecución de estados y sus transiciones. update, verifica las transiciones, y ejecuta las acciones del estado actual, realiza la transición al estado objetivo	aprox. 116
SubMachine State	Combina la funcionalidad de un estado y la máquina de estados.	aprox. 14
HSMBase	Interface que permite manejar las transiciones y estados por la jerarquía propuesta	aprox. 17

Tabla 5.1: Descripción de la implementación de la HFSM para comportamientos.

La implementación anterior lleva control sobre la ejecución de la máquina de estados jerárquica, y que puede ser considerada como base para la implementación de un comportamiento. Millington lo ha considerado como parte de su core de IA. y que debe ser tomado en cuenta a la hora de programar. Luego de esta implementación, completamos el código para el primer ejemplo de un comportamiento de SBTournament.

La Figura 5-2 muestra la implementación de nuestro primer comportamiento llamado "Centro delantero". Este implementa a un jugador que siempre está en el centro del campo, pero que su comportamiento depende del marcador, de la posición del balón y su ubicación. Los distintos roles que jugará son central ofensivo o central defensivo.

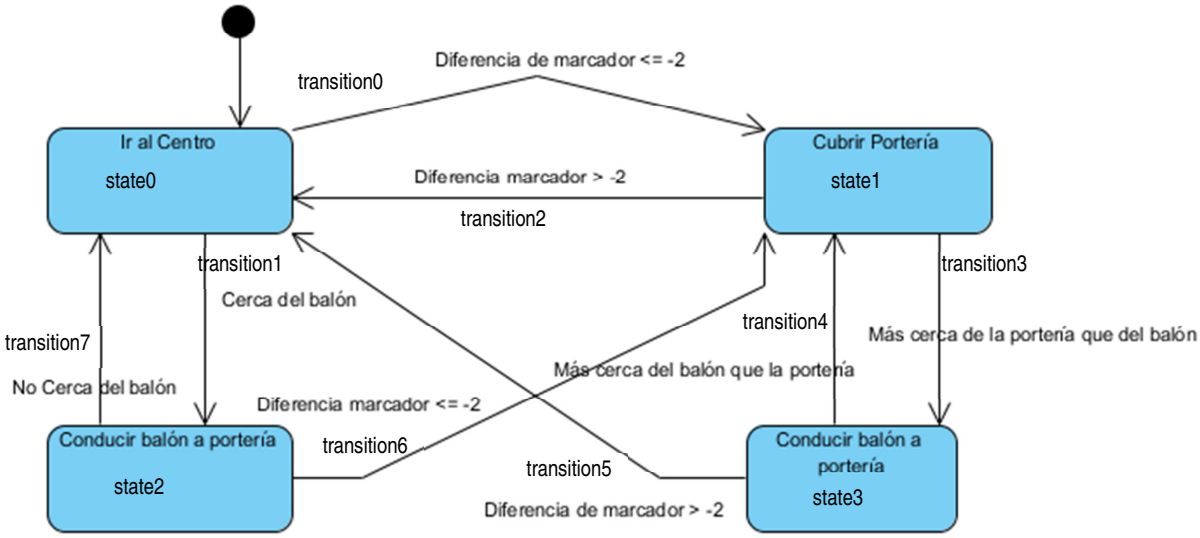

Figura 5-2: FSM para el comportamiento "Centro delantero" usando notación UML

En este ejemplo se prefiere el uso de una máquina de estados sencilla sin jerarquías con fines de demostrar la asignación de código a cada estado y transición que van a componer la máquina de estados en la implementación de HFSM propuesta al inicio de esta sección.

El primer bloque de código a mostrar es la asignación de cada estado, las acciones y sus transiciones. Por cada estado se establecen el conjunto de acciones que deben ser realizadas, estas acciones serán codificadas como llamadas a funciones que serán implementadas en cada estado. Las acciones que describen a cada estado son las siguientes, *Ir al Centro*: mueve al jugador hasta el centro del campo, *Cubrir Portería*: bloquea el balón en camino a su portería, *Conducir balón a portería*: lleva la pelota a la portería contraria.

```
//Establecer cada estado de la FSM con sus acciones
State[] states = new State[4];
states[0] = new State();
states[0].actions = "actions_state0";
states[1] = new State();
states[1].actions = "actions_state1";
states[2] = new State();
states[2].actions = "actions_state2";
states[3] = new State();
states[3].actions = "actions_state3";

//Establecer cada transicion de la FSM con sus estados destinos
Transition[] transitions = new Transition[8];
transitions[0] = new Transition();
transitions[0].targetState = states[1];
transitions[0].condition = "condition0";
transitions[1] = new Transition();
transitions[1].targetState = states[2];
transitions[1].condition = "condition1";
transitions[2] = new Transition();
transitions[2].targetState = states[0];
transitions[2].condition = "condition2";
transitions[3] = new Transition();
transitions[3].targetState = states[3];
transitions[3].condition = "condition3";
…

//Establecer las transiciones por cada estado
states[0].transitions = new Transition[2];
states[0].transitions[0] = transitions[0];
states[0].transitions[1] = transitions[1];
states[1].transitions = new Transition[2];
states[1].transitions[0] = transitions[3];
states[1].transitions[1] = transitions[4];

…
```

Listado 5-1: Bloque de código de implementación del comportamiento "Centro delantero"

Ambos, los estados y las transiciones son manejadas mediante arreglos y sus índices son los identificadores, es decir, a cada índice del arreglo de estados corresponde un estado de la Figura 5-2 y lo mismo para las transiciones, a cada índice corresponde una transición de la figura anterior.

Existen 3 bloques de código en el Listado 5-1, los cuales nos permiten obtener un patrón del número de líneas de código utilizadas. El primer bloque establece todos los estados, por cada estado hay dos líneas de código, el segundo bloque establece las transiciones tiene tres líneas por cada estado, el siguiente bloque se establecen las

transiciones por estado y el número de líneas por estado es igual al número de transiciones por estado mas una. De manera que se puede obtener la siguiente fórmula.

$$Líneas\ de\ código = (N * 2) + (E * 3) + \sum_{node-i=1}^{N} (E_{node-i} + 1)\ ;\ siendo$$

$$, N: número\ de\ estados$$

$$, E: número\ de\ transiciones$$

$$, E_{node-i}: número\ de\ transiciones\ del\ nodo\ iésimo$$

Finalmente podemos obtener una medida de número de líneas de código para la implementación de nuestro comportamiento.

Implementación	**Número de líneas de código**
Core de Máquina de estados jerárquica	116
Construcción de conjunto de estado y transiciones y relaciones entre estos (N=4,E=7,E_{node}=2)	41

Hasta aquí, solo se ha construido la máquina de estados, pero el código que realiza las acciones el motor del juego y las condiciones que producen las transiciones entre estados es difícil de cuantificar y dependen específicamente de la lógica de cada implementación. En nuestro ejemplo las acciones del estado *Cubrir la Portería* y de la condición de la transición transition0 corresponderían al código del Listado 5-2.

Del código del Listado 5-2 podemos determinar una línea adicional que corresponda a la declaración de los métodos para cada llamada, de manera que la fórmula quedaría como sigue, con lo mismo parámetros anteriores.

$$Líneas\ de\ código = (N * 3) + (E * 4) + \sum_{node-i=1}^{N} (E_{node-i} + 1)\ ;\ siendo$$

```
public void actions_state1() {
	Vec2 dest = new Vec2(myRobotAPI.getOurGoal());
	dest.add(myRobotAPI.getBall());
	myRobotAPI.setSteerHeading(dest.t);
	myRobotAPI.setSpeed(1.0);
	myRobotAPI.setDisplayString("Cover Goal");
	}

public Boolean condition0() {
	return (myRobotAPI.getDistanceToBall()<myRobotAPI.getDistanceToGoal());
}
…
```

Listado 5-2: Bloque de código de las acciones del estado state1 y de la condición condition0

5.2 Diseño de un boceto de comportamientos desde cero.

Como se ha dicho en el inicio de este capítulo es necesario definir los comportamientos básicos que servirán para la construcción de los comportamientos creados por el diseñador. Al crear un nuevo repositorio inicialmente esta librería está vacía. Basados en la implementación del API de SBTournament y el conjunto de sensores y actuadores que se encuentran en el Apéndice A y B, podemos construir los siguientes comportamientos básicos. Se han elegido los siguientes comportamientos de la Tabla 5.2 basados en la experiencia de torneos anteriores y junto con el API anterior de Soccerbots. Cada comportamiento básico se refiere a las acciones ejecutadas por un jugador de SBTournament o en el caso de Soccerbots se refiere al comportamiento de un robot del campo de juego en miniatura. Se utilizarán los nombres en inglés para tener coherencia con los diseños de la herramienta.

Puesto que, lo que se intenta es encontrar una medida de comparación entre un comportamiento creado por la asistencia de la herramienta y uno que no lo es, el siguiente ejemplo implementará la construcción de una máquina de estados para el comportamiento "*Centro delantero*" propuesto como ejemplo en la sección anterior. Suponiendo que la se cuentan con los bloques de código de los comportamientos básicos y se conoce las descripciones de estos y de cada uno de los sensores, el trabajo se centra únicamente en el diseño del comportamiento.

Comportamiento básico	Descripción
Cover Goal	Bloquea la trayectoria del balón en camino a la portería local
Kick	Patea el balón si le es posible
Lead Ball to Goal	Conduce el balón a la portería contraria
Go To Center	Se mueve hacia el centro del campo
Block Forward	Cubre al jugador más adelantado del equipo contrario
Walk Toward Goal	Se mueve hacia la portería local
Wait	Permanece en su posición.
Walk Toward Ball	Se mueve hacia el balón

Tabla 5.2: Comportamientos básicos de un repositorio dirigido a SBTournament

Partiendo de la suposición anterior, el diseñador se debe preocupar del diseño de bloques de comportamientos independientes más que de las acciones que se realizan en cada estado. Por ejemplo, el comportamiento "*Centro delantero*" de la Figura 5-2 puede ser dividido en dos comportamientos en función del marcador, "*Central defensivo*" o simplemente "*Central*" y "*Central Ofensivo*" o "*Delantero*". Este diagrama del comportamiento es mucho más sencillo de entender que el anterior. La estrategia del *Centro delantero* es ofensiva cuando el equipo va perdiendo 2 a 0, caso contrario la estrategia es defensiva (véase Figura 5-3).

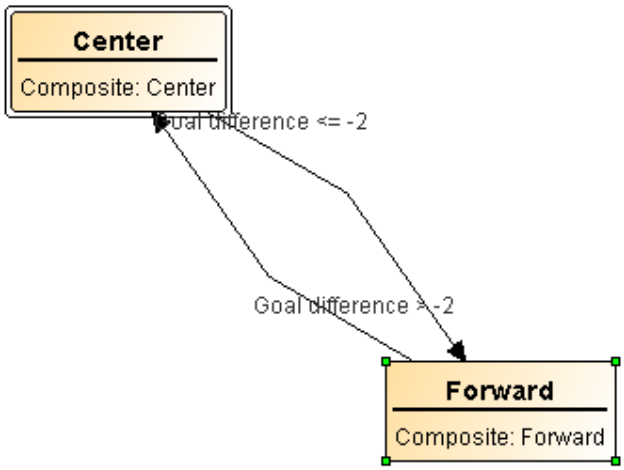

Figura 5-3: Comportamiento "Centro delantero" creado por la herramienta

El diseño de "*Centro delantero*" está compuesto de dos comportamientos con funcionalidad propia, "*Forward*" (Central ofensivo) que espera en el centro del campo hasta estar cerca de la pelota y la intenta meter en la portería contraria, "*Center*" (Central defensivo) defiende la portería local hasta estar cerca de la pelota e intenta meterla en la portería contraria.

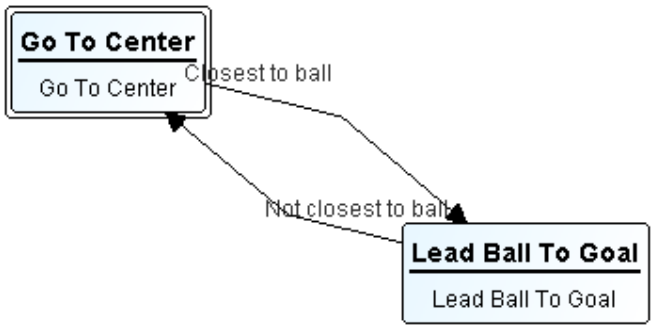

Figura 5-4: Comportamiento "*Forward*" central ofensivo

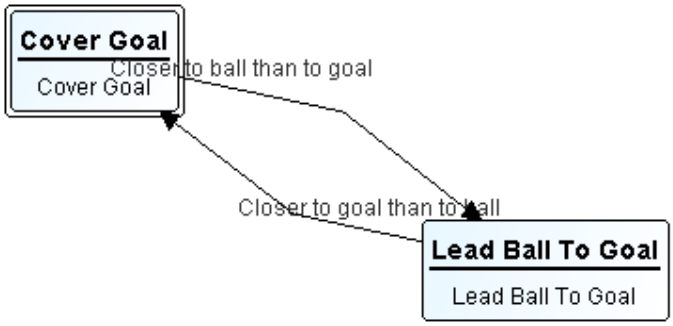

Figura 5-5: Comportamiento "*Center*" central defensivo

Básicamente, el comportamiento "*Centro delantero*" ha sido construido en 3 iteraciones distintas, una por cada figura. Además, se puede decir que se ha construido con 3 FSMs similares en estructura, es decir, con dos estados y una transición de salida y otra de retorno en cada estado (ver Figuras). El número de pasos para construir cada FSM se muestra en la Tabla 5.3, la cual consta de 4 operaciones de edición básicas de la herramienta, tales como son, la creación de nodos y asignación de transiciones entre nodos. Sin contar el establecimiento de las condiciones, es obvio que la creación de la FSM estaría completa con 4 pasos de edición. Por consiguiente, el diseño de "*Centro delantero*" puede ser diseñado a partir de una sola estructura, la cual es adaptada en 3 sub-máquinas diferentes, con comportamientos básicos distintos y transiciones distintas.

Operación de edición	Nombre
Creación de un nodo	Nodo 1: Cover Goal
Creación de un nodo	Nodo 2: Lead Ball To Goal
Creación de una transición	De nodo 1 a nodo 2
Creación de una transición	De nodo 2 a nodo 1

Tabla 5.3: Operaciones de edición para crear el comportamiento "*Center*"

Para finalizar se genera el código del comportamiento y se incorpora en el motor del juego y se consiguen los mismo resultados que con la primera implementación que se creó sin una herramienta de autoría.

5.3 Diseño asistido de comportamientos basado en bocetos

Hasta ahora, las ventajas presentadas en esta evaluación tienen relación con la facilidad de edición y con la modularidad para el diseño de comportamientos. No se ha evaluado con respecto al re-uso de comportamientos del repositorio. Basados en varios ejemplos se tratará de demostrar la importancia de tener una buena base de casos y contar con comportamientos pre-existentes para posteriores desarrollos.

El punto clave del diseño basado en bocetos es, a partir del dibujo de una estructura de una FSM de un comportamiento sin completar, el diseñador reciba las sugerencias más aproximadas de comportamientos que podrían tener la funcionalidad final deseada. A continuación en la Tabla 5.4, se listan los comportamientos dentro del repositorio y los resultados de similitud obtenidos en una búsqueda sencilla.

En la primera iteración de búsqueda se trata de encontrar la figura de la consulta 1 (ver Tabla 5.5). Obviamente este conjunto de nodos y arcos es muy común en el diseño de comportamientos y debió a esto los resultados obtenidos no son tan buenos y es necesario que el diseñador revise la lista de resultados para encontrar el más conveniente. El comportamiento objetivo ha sido encontrado en la primera consulta con un valor de 0.4 de similitud, puesto que, el resultado tiene varios comportamientos con la misma medida entonces el diseñador debe seleccionar uno de ellos.

Comportamiento básico	Descripción
ForwardBlocker	Si existe un portero en el equipo contrario, este trata de bloquearlo, caso contrario actúa como delantero.
ForwardBallhog	Delantero que siempre trata de anotar en la portería contraria.
Forward	Defiende la portería local hasta estar cerca de la pelota e intenta meterla en la portería contraria
Center	Se mantiene en centro del campo hasta que esté cerca de la pelota y la conduce a la portería contraria
Simple Goalkeeper	Cubre la portería local y despeja cuando tiene la oportunidad de disparar el balón
Defender	Previene que el oponente ataque cubriendo la portería local y bloqueando al delantero contrario
Blocker	Bloquea al que el portero contrario evitando que se mueva y anotar con más facilidad
Advanced Goalkeeper	Se mantiene en una posición de avance con respecto a un Simple Goalkeeper.

Tabla 5.4: Lista de comportamientos del repositorio inicial para SBTournament.

En la segunda consulta, también se obtiene un resultado favorable en la primera iteración del boceto. Con un resultado de 0.3076 para el comportamiento Simple Goalkeeper, y construido a partir de 3 estados y tres transiciones es el resultado más relevante de nuestra base de casos. Dado que el objetivo era diseñar ese mismo comportamiento, no se necesitarán más consultas. Pero a manera de ejemplo, se incluye una consulta Nº 3 con una iteración con más detalle del boceto 2 del comportamiento objetivo. El resultado con mayor precisión sigue siendo el mismo Simple Goalkeeper con 0.4 como medida de similitud.

Con estos simples ejemplos se pretende demostrar la asistencia por parte de la herramienta al diseñador basado en bocetos de comportamientos. Los resultados esperados en cuanto a la agilidad al momento de construir un comportamiento se sustentan en la inclusión de comportamientos o partes de ellos que ya existen con anterioridad en la base de casos.

Nº	Estructura inicial	Objetivo	Resultado
1		Diseñar el comportamiento "Centro Delantero"	Blocker: 0.4 CenterForward: 0.4 ForwardBlocker: 0.2857 Conseguido en la primera iteración: CenterForward
2		Diseñar el comportamiento "Simple Goalkeeper"	Simple Goalkeeper: 0.3076 Advanced Goalkeeper: 0.2857 Conseguido en la primera interación
3	Go to goal Walk Towards Goal	Diseñar el comportamiento "Simple Goalkeeper"	Simple Goalkeeper: 0.4 Conseguido en la segunda iteración

Tabla 5.5: Consultas por similitud estructural realizadas en la herramienta

Pueden ser incluidos otros ejemplos más complejos, pero se ha preferido incluir estos más sencillos con el fin de llegar a comprender la recuperación de comportamientos por bocetos con menos iteraciones. En futuros trabajos se obtendrá una medida en función de los resultados esperados por el comportamiento.

5.4 Resultados de la evaluación

El número de líneas de código que se escriben para construir la HFSM y su manejo, así como, la construcción de la estructura de cada comportamiento dentro de este bloque de código es una medida de utilidad para determinar el número de líneas de código que un

programador ahorra al momento de programar comportamientos de IA. De manera aproximada se ahorran 157 más el número de líneas que derivan de cada estado y transición.

La separación de los bloques básicos de código o comportamientos básicos, le permiten al diseñador ocuparse del diseño de cada comportamiento de manera general y más abstracta y no ocuparse de los detalles de programación.

Las facilidades de edición y de incluir comportamientos de manera modular también ayudan al momento de diseñar comportamientos jerárquicos construidos por otros comportamientos más simples.

La base de casos es un factor importante en la construcción de nuevos comportamientos, se obtuvieron buenos resultados, debido a que, los comportamientos objetivos o buscados ya se encontraban en el repositorio. Pero en los casos que no se encuentre el comportamiento deseado dentro de la base, la recuperación por partes será útil cuando se diseñen comportamientos de jerarquía más profunda.

6 Conclusiones

Las pruebas realizadas con la herramienta de autoría, muestran que nuestro enfoque es factible como una metodología de trabajo, y que los diseñadores y programadores podrán maximizar el trabajo en conjunto y minimizar el tiempo de producción de comportamientos de NPC destinados a un motor de simulación o videojuegos en particular. Estos supuestos, son respaldados por las pruebas realizadas en la evaluación de la metodología a través de la herramienta de autoría desarrollada.

También creemos que los objetivos planteados al principio del trabajo han sido cumplidos, podemos tomar en cuenta importantes factores de la metodología, como la representación utilizada de HFSMs, el enriquecimiento de información que se le da a los repositorios de comportamientos, el diseño asistido por similitud estructural y funcional, la construcción de bloques de código destinados a los programadores de IA. Todos y cada uno de estos factores incluidos en la metodología hacen suponer que hemos alcanzado nuestro objetivo.

Aunque la implementación actual de la herramienta no incluya algunas técnicas de usabilidad, creemos que es factible desarrollar todo el ciclo de producción de un comportamiento. Dentro de las mejoras de usabilidad podríamos destacar:

- Incluir un editor gráfico de fórmulas, una aproximación sería utilizar un lenguaje de etiquetas para representarlas y un motor gráfico de edición de.
- Incluir parámetros de entrada y de salida en la construcción de comportamientos básicos, de manera que puedan ser utilizados, en la construcción de las condiciones.
- Incluir una depuración del comportamiento del tipo ventana de ejecución de texto con valores generados, para no depender el motor de videojuegos.
- Escribir los generadores de código a través de la herramienta siguiendo algún estándar de construcción de código para motores de videojuegos.

Bibliografía

1. **Staff.** GDC Austin: An Inside Look At The Universe Of Warcraft. *Gamasutra.* [Online] Sept. 17, 2009. http://www.gamasutra.com/php-bin/news_index.php?story=25307.

2. *Head games: Double Fine's psychonautic break.* **Esmurdoc, C.** s.l. : Game Developer Magazine, 2005.

3. **Engine, Game.** Unreal Development Kit. [Online] 2012. http://www.udk.com/.

4. **Crytek.** CryENGINE 3 SDK. *Sandbox Editor.* [Online] 2010. http://mycryengine.com.

5. *BehaviorShop: An Intuitive Interface for Interactive Character Design.* **Frederick W. P. Heckel, G. Michael Youngblood, and D. Hunter Hale.** s.l. : Association for the Advancement of Artificial Intelligence, 2009.

6. **Daniel Fu and Ryan Houlettem, Stoller Henke Associates.** Putting AI in Entertaiment: An AI Autorithing Tool for Simulation and Games. s.l. : IEEE Intelligence Systems, 17(4):81–84, 2002.

7. **James, Wexler.** *Artificial Intelligence in Games: A look at the smarts behind Lionhead Studio's "Black and White" and where it can and will go in the future.* Rochester, NY 14627 : University of Rochester, 2002.

8. **Champandard, Alex J.** *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors.* s.l. : New Riders, 2003.

9. **FEAR.** Flexible Embodied Agent aRchitecture. [Online] http://fear.sourceforge.net/docs/latest/guide/.

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

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

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

13. **Kolodner, J.** *Case-based reasoning.* s.l. : Morgan Kaufmann, 1993.

14. **A. Aamodt, E. Plaza.** *Case-Based Reasoning: Foundational Issues, Methodological Variations, and System.* s.l. : AI Communications. IOS Press, 1994. Vol. 7: 1, pp. 39-59.

15. **Christopher K. Riesbeck, Roger C. Schank.** Case-based Reasoners. *Inside Case Based Reasoning.* s.l. : Lawrence Erlbaum, 1989.

16. **Ralph Bergmann, J. Kolodner, and Enrick Plaza.** *Representation in case-based reasoning.* [The Knowledge Engineering Review] United Kingdom : The Knowledge Engineering Review, Cambridge University Press, 2005.

17. **E. K. Burke, B. MacCarthy, S. Petrovic, R. Qu.** *Structured Cases in CBR – Re-using and Adapting Cases for Timetabling.* [Knowledge-Based Systems] Nottingham : aSchool of Computer Science and Information Technology, The University of Nottingham, 2000.

18. **Gonzalo Flórez Puga, María Belén Díaz Agudo.** *Edición Semiautomática de Comportamientos en Entornos de Videojuegos.* Madrid : Universidad Complutense de Madrid, 2007.

19. **Aamodt, A.** A knowledge-intensive approach to problem solving and sustained learning, Ph.D. 1991.

20. **Carbonell, J.** Derivational analogy; A theory of reconstructive problem solving and expertise acquisition. [book auth.] Morgan Kaufmann. *Machine Learning - An Artificial Intelligence Approach, Vol.II.* pp. 371-392 : s.n., 1986.

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

22. **H. Bunke y B. Messmer.** «Similarity measures for structured representations», in Topics in Case-Based Reasoning, vol. 837, S. Wess, K.-D. Althoff, y M. Richter, Eds. Springer Berlin / Heidelberg, 1994, pp. 106-118.

23. **Kaspar Riesen and Horst Bunke.** Approximate graph edit distance computation by means of bipartite graph matching, Image Vision Comput., vol. 27, nº. 7, pp. 950–959, jun. 2009.

24. *Rapid Development of Intelligent Agents in First/third-person Training Simulations via Behavior-based Control.* **George Alexander, Frederick W. P. Heckel, G. Michael Youngblood, D. Hunter Hale, Nikhil S. Ketkar.** s.l. : University of North Carolina at Charlotte, March 2010.

25. **Bonnie Lida Rogers and Barbara Chaparro.** *Breadcrumb Navigation: Further Investigation of Usage.* Lida, Hull & Pilcher, 2003.

26. **Tucker Balch,.** TeamBotsTM Domain: SoccerBots. *TeamBots(tm) software.* [Online] Georgia Tech Research Corporation (GTRC) , Copyright 1999, 2000. http://www.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots/.

27. GAIA – Group of Artificial Intelligence Applications. *SBTournament.* [Online] Universidad Complutense de Madrid, Spain. http://gaia.fdi.ucm.es/research/sbtournament.

28. **Ian Millington and John Funge.** Desicion Making. *Aritificial Intelligence for Games.* 2009.

29. **J, Kolodner.** *An introduction to case-based reasoning.* s.l. : Artificial Intelligence Review, 1992. 6(1), pp. 3-34.

30. **Börner K, Coulon CH, Pippig E and Tammer EC.** *Structural similarity and adaptation, in: Smith I and Faltings B, eds.* [Advances in Case-based Reasoning] s.l. : Springer-Verlag, Switzerland, 1996.

31. *"metric space" The Concise Oxford Dictionary of Mathematics. Christopher Clapham and James Nicholson. Oxford University Press, 2009. Oxford Reference Online. Oxford University Press. Universidad Complutense de Madrid. 3 February 2012.*

32. The Machine Learning Dictionary. *Copyright © Bill Wilson, 1998, 2011.* [Online] COMP9414. http://www.cse.unsw.edu.au/~billw/mldict.html#ID3.

33. **"authoring noun" Oxford Dictionary of English. Edited by Angus Stevenson. Oxford University Press, 2010. Oxford Reference Online. Oxford University Press. Universidad Complutense de Madrid. 5 February 2012.** [Online] http://0-www.oxfordreference.com.cisne.sim.ucm.es/views/ENTRY.html?subview=Main&entry=t140.e0049850.

34. **Gonzalo Flórez-Puga, Pedro A. González-Calero, Guillermo Jiménez-Díaz and Belén Díaz-Agudo.** Supporting Sketch-based Retrieval from a Library of Reusable Behaviours, IEEE Trans. Comput. Intellig. and AI in Games (enviado).

Apéndice A: Ficheros de configuración xml

Ejemplo de un modelo del juego para un repositorio

```
<?xml version="1.0" encoding="UTF-8"?>
<modeloJuego 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:bb="http://www.local.com/eCo/xml/BasicBehaviours.xsd" nombre="">
  <ij:interfazJuego>
    <ij:sensores>
      <ij:sensor nombre="Blocked" tipo="BOOLEAN" comando="myRobotAPI.blocked()"
categoria="NATURAL">
        <ij:descripcion>Returns if the robot is blocked
        </ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="canKick" tipo="BOOLEAN" comando="myRobotAPI.canKick()"
categoria="NATURAL">
        <ij:descripcion>Reveals whether or not the ball can be kicked.
        </ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="closestToBall" tipo="BOOLEAN"
comando="myRobotAPI.closestToBall()" categoria="NATURAL">
        <ij:descripcion>Checks if the robot is the closest one to the ball
             </ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="getMyScore" tipo="NUMERIC"
comando="myRobotAPI.getMyScore()" categoria="NATURAL">
        <ij:descripcion>Returns player's team score
             </ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="getPlayerNumber" tipo="NUMERIC"
comando="myRobotAPI.getPlayerNumber()" categoria="NATURAL">
        <ij:descripcion>Get the robot's player number, between 0 and the number of
robots in the team.
             </ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="getSteerHeading" tipo="NUMERIC"
comando="myRobotAPI.getSteerHeading()" categoria="NATURAL">
        <ij:descripcion>Get the current heading of the steering motor.
             </ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="getBall" tipo="ACTUATOR" comando="myRobotAPI.getBall()"
categoria="SINTETICO">
        <ij:descripcion>Get a Vec2 that points egocentrically from the center of
the robot to the ball.
             </ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="getBallX" tipo="NUMERIC" comando="myRobotAPI.getBall().x"
categoria="SINTETICO">
        <ij:descripcion>Get the X component of the vector returned by
getBall</ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="getBallY" tipo="NUMERIC" comando="myRobotAPI.getBall().y"
categoria="SINTETICO">
```

```
        <ij:descripcion>Get the Y component of the vector returned by
getBall</ij:descripcion>
      </ij:sensor>
      <ij:sensor nombre="isBlockingMe" tipo="ACTUATOR"
comando="myRobotAPI.isBlocking(Vec2 anotherplayer)" categoria="SINTETICO">
        <ij:descripcion>Checks if another player is blocking us.
             </ij:descripcion>
      </ij:sensor>

    </ij:sensores>
    <ij:actuadores>
      <ij:actuador nombre="avoidCollisions" parametros="0"
comando="myRobotAPI.avoidCollisions()" categoria="NATURAL">
        <ij:descripcion>Tries to avoid a collision with the closest player
             </ij:descripcion>
      </ij:actuador>

      <ij:actuador nombre="blockClosest" parametros="0"
comando="myRobotAPI.blockClosest()" categoria="NATURAL">
        <ij:descripcion>Tries to block the closest opponent
             </ij:descripcion>
      </ij:actuador>

      <ij:actuador nombre="blockForward" parametros="0"
comando="myRobotAPI.blockForward()" categoria="NATURAL">
        <ij:descripcion>Tries to block opponent's forward (opponent closest to our
goal)
             </ij:descripcion>
      </ij:actuador>

      <ij:actuador nombre="blockGoalKeeper" parametros="0"
comando="myRobotAPI.blockGoalKeeper()" categoria="NATURAL">
        <ij:descripcion>Tries to block opponent's goalkeeper (opponent closest to
their goal)
             </ij:descripcion>
      </ij:actuador>

      <ij:actuador nombre="kick" parametros="0" comando="myRobotAPI.kick()"
categoria="NATURAL">
        <ij:descripcion>If the ball can be kicked, kick it.
             </ij:descripcion>
      </ij:actuador>

    </ij:actuadores>
  </ij:interfazJuego>
  <prop:propiedades>
    <prop:propiedad nombre="Defender" min="0.0" max="5.0" valDefecto="2.0" />
    <prop:propiedad nombre="Attacker" min="0.0" max="5.0" valDefecto="3.0" />
    <prop:propiedad nombre="Mobility" min="0.0" max="5.0" valDefecto="4.0" />
    <prop:propiedad nombre="Goalkeeper" min="0.0" max="1.0" valDefecto="1.0" />
  </prop:propiedades>
  <bb:basicBehaviours>
    <bb:basicBehaviourDefinition name="Lead Ball To Goal">
myRobotAPI.alignedToBallandGoal();
myRobotAPI.kick();
myRobotAPI.setDisplayString("Lead Ball To Goal");
    </bb:basicBehaviourDefinition>
    <bb:basicBehaviourDefinition name="Block Forward">
myRobotAPI.blockForward();
```

```
myRobotAPI.setDisplayString("Block Forward");
    </bb:basicBehaviourDefinition>
    <bb:basicBehaviourDefinition name="Go To Center">
Vec2 destino = new Vec2(0.0, 0.0);
destino.sub(myRobotAPI.getPosition());
myRobotAPI.setSteerHeading(destino.t);
myRobotAPI.setSpeed(1.0);
myRobotAPI.setDisplayString("Go to center");
      </bb:basicBehaviourDefinition>

  </bb:basicBehaviours>

</modeloJuego>
```

Ejemplo de un comportamiento creado por la herramienta

```
<?xml version="1.0" encoding="UTF-8"?>
<compCompuesto xmlns="http://www.local.com/eCo/xml/Comportamiento.xsd"
nombre="simpletest" descripcion="">
  <listaPropiedades>
    <propiedad nombre="Attacker" valor="3.0" />
    <propiedad nombre="Defender" valor="2.0" />
    <propiedad nombre="Mobility" valor="4.0" />
    <propiedad nombre="Goalkeeper" valor="1.0" />
  </listaPropiedades>
  <nodos>
    <nodo id="1" inicial="true">
      <compAtomico nombre="Cubrir Portería" descripcion="Cubre la portería.">
        <listaPropiedades>
          <propiedad nombre="Defender" valor="2.0" />
          <propiedad nombre="Attacker" valor="3.0" />
          <propiedad nombre="Mobility" valor="4.0" />
          <propiedad nombre="Goalkeeper" valor="1.0" />
        </listaPropiedades>
        <basicBehaviour name="Block Goalkeeper" />
      </compAtomico>
      <vista x="103" y="95" />
    </nodo>
    <nodo id="2" inicial="false">
      <compAtomico nombre="Conducir pelota a portería" descripcion="Kick">
        <listaPropiedades>
          <propiedad nombre="Defender" valor="2.0" />
          <propiedad nombre="Attacker" valor="3.0" />
          <propiedad nombre="Mobility" valor="4.0" />
          <propiedad nombre="Goalkeeper" valor="1.0" />
        </listaPropiedades>
        <basicBehaviour name="Lead Ball To Goal" />
      </compAtomico>
      <vista x="202" y="345" />
    </nodo>
  </nodos>
  <aristas>
    <arista id="4" origen="1" destino="2" descripcion="Más cerca de la pelota que
de la portería">
      <disyuncion>
```

```
        <conjuncion>
          <llamadaSensor nombre="canKick" comparador="=" valor="true" />
        </conjuncion>
      </disyuncion>
    </arista>
    <arista id="5" origen="2" destino="1" descripcion="Más cerca de la portería
que de la pelota">
      <disyuncion>
        <conjuncion>
          <llamadaSensor nombre="canKick" comparador="=" valor="false" />
        </conjuncion>
      </disyuncion>
    </arista>
  </aristas>
</compCompuesto>
```

Apéndice B: Listado de sensores y actuadores

Listado de sensores y actuadores para SBTournament

Listado de sensores y actuadores del API de SBTournament
ACTUADOR
avoidCollisions
Tries to avoid a collision with the closest player
blockClosest
Tries to block the closest opponent
blockForward
Tries to block opponent's forward (opponent closest to our goal)
blockGoalKeeper
Tries to block opponent's goalkeeper (opponent closest to their goal)
kick
If the ball can be kicked, kick it.
normalizeZero
Normalizes an angle in range [-PI,PI]
setBehindBall
Moves the player behind the ball and pushes it.
setDisplayString
Set the String that is printed on the robot's display. For real robots, this is printed on the console. For simulated robots, this appears printed below the agent when view "Robot State" is selected.
setSpeed
Set the desired speed for the robot (translation). The speed must be between 0 and 1; where 0 is stopped and 1.0 is "full blast". It will be clipped to whichever limit it exceeds.
setSteerHeading
Set the desired heading for the steering motor.
surroundPoint
Moves behind a given point, pushes that point trying to reach another destination point.

SENSOR
alignedToBallandGoal
Checks if the robot is behind the ball and aligned with the opponent's goal
behindEverybody
Checks if the robot is the closest to our goal
blocked
Returns if the robot is blocked
canKick
Reveals whether or not the ball can be kicked.
closestToBall
Checks if the robot is the closest one to the ball
getFieldSide
Gets the field side
getID
Get the unique ID of the robot.
getJustScored
Get an integer that indicates whether a scoring event just occurred
getMatchRemainingTime
Returns the remaining time for this match in milliseconds
getMatchTotalTime
Returns the total time of this match in milliseconds
getMyScore
Returns player's team score
getOpponentScore
Returns opponents' score
getPlayerNumber
Get the robot's player number, between 0 and the number of robots in the team.
getPlayerRadius
Gets the radius of a robot
getSteerHeading
Get the current heading of the steering motor.
getTimeStamp
Returns the elapsed simulation time in milliseconds
goalkeeperBlocked
Checks if the goalkeeper is blocked
opponentBlocking
Returns if the robot is blocked by an opponent
opponentsHaveGoalkeeper
Checks if the opponent team has a goalkeeper
teammateBlocking
Returns if the robot is blocked by a teammate
SENSOR SINTÉTICO
getBall
Get a Vec2 that points egocentrically from the center of the robot to the ball.

getClosestMate
Gets a Vec2 that points egocentrically from the center of the robot to the closest teammate
getClosestOpponent
Gets a Vec2 that points egocentrically from the center of the robot to the closest opponent
getGoalkeeper
Returns the position of our goalkeeper (i.e. the closest mate to the goal)
getObstacles
Get an array of Vec2s that point egocentrically from the center of the robot to the obstacles currently sensed by the robot's sensors
getOpponents
Get an array of Vec2s that point egocentrically from the center of the robot to opponents.
getOpponentsGoal
Gets a Vec2 that points egocentrically from the center of the robot to the opponent's goal. This is the one to get the ball across to score.
getOurGoal
Gets a Vec2 that points egocentrically from the center of the robot to the team's goal. This is the one to defend.
getPosition
Get the position of the robot in global coordinates.
getSortedMates
Gets a sorted tree map with the distances to our mates
getSortedOpponents
Gets a sorted tree map with the distances to the opponent players
getTeammates
Get an array of Vec2s that point egocentrically from the center of the robot to the teammates
isBlocking
Checks if a player is blocking another one.
Checks if another player is blocking us.

Apéndice C: Código generado por la herramienta de autoría

Código generado para el motor de simulación de SBTornament

Fichero EquipoPruebasJorgeTeamManager.java

```
package equipoPruebasJorge;//nombre del paquete generado e ingresado en la ventana
de dialogo

//IMPORT SECTION
import java.util.Queue;

import EDU.gatech.cc.is.util.Vec2;

import teams.rolebased.Behaviour;
import teams.rolebased.Message;
import teams.rolebased.Message.Tipo;
import teams.rolebased.RobotAPI;
import teams.rolebased.TeamManager;
```

```java
public class EquipoPruebasJorgeTeamManager extends TeamManager {
	/**
	 * Numbers of behaviours managed
	 */
	public static int numOfBehaviours = 1;

	@Override
	public int configure() {
		// TODO Auto-generated method stub
		return 0;
	}

	@Override
	public void takeStep() {

	}
	/**
	 * This method get Behavior that corresponds to each one player of the team
	 */
	@Override
	public Behaviour getDefaultBehaviour(int id) {
		//TODO: En esta primera implementación solo existe un comportamiento
a ser asignado
		Behaviour c = null;
		if (_behaviours!=null && _behaviours.length>0) {
			c = _behaviours[0];
		}
		return c;
	}

	/**
	 * This method creates all behaviours to be used by team manager
	 */
	@Override
	public Behaviour[] createBehaviours() {
		_behaviours =new Behaviour[numOfBehaviours];
	  for (int i=0; i<numOfBehaviours; i++)
		_behaviours[i] = new EquipoPruebasJorgeBehaviour();

	  return _behaviours;
	}

}
```

Fichero EquipoPruebasJorgeBehaviour.java

```java
//NAME'S PACKAGED GIVEN FROM CODE GENERATOR
package equipoPruebasJorge;

//IMPORT SECTION
import java.lang.reflect.InvocationTargetException;
import java.util.Vector;

import teams.rolebased.Behaviour;

//NAME'S CLASS GIVEN FROM GENERATOR FOR BEHAVIOR EXTENDS
public class EquipoPruebasJorgeBehaviour extends Behaviour {
```

```
	/**
	 * Table for manage all state of the HFMS
	 */
	private Vector<State> states;
	/**
	 * Stack for manage the current state and the hierarchical child state of
the current state
	 */
	private Vector<State> stack;


	@Override
	public void OnConfigure() {
		//Call Generate the list of states from HFSM
		GenerateStates();
	}

	/**
	 * Put the code that execute on each iteration of simulation
	 */
	@Override
	public int takeStep() {
		try
		{
			this.getClass().getMethod(stack.lastElement().actions,
(Class[]) null).invoke(this, (Object[]) null);
		}
		catch (IllegalArgumentException ex) {
			ex.printStackTrace();
		} catch (SecurityException ex) {
			ex.printStackTrace();
		} catch (IllegalAccessException ex) {
			ex.printStackTrace();
		} catch (InvocationTargetException ex) {
			ex.printStackTrace();
		} catch (NoSuchMethodException ex) {
			ex.printStackTrace();
		}
		try
		{
			int positionStack = 0;
			while(positionStack < stack.size())
			{
				State e = stack.get(positionStack);
				if (e.transitions != null)
				{
					for (int j = 0; j < e.transitions.length; j++)
					{
						if ((Boolean)
this.getClass().getMethod(e.transitions[j].condition, (Class[]) null).invoke(this,
(Object[]) null))
						{
							// Remove state of the stack
							for (int k = stack.size() - 1; k >=
positionStack; k--) {
								stack.remove(k);
							}
							// Put new state over the stack
```

```
stack.add(states.get(e.transitions[j].state));
                                }
                            }
                        }
                        positionStack++;
                    }
        } catch (IllegalArgumentException ex) {
                    ex.printStackTrace();
        } catch (SecurityException ex) {
                    ex.printStackTrace();
        } catch (IllegalAccessException ex) {
                    ex.printStackTrace();
        } catch (InvocationTargetException ex) {
                    ex.printStackTrace();
        } catch (NoSuchMethodException ex) {
                    ex.printStackTrace();
        }

        return myRobotAPI.ROBOT_OK;
}

@Override
public void onInit() {
        //Initialize the stack with the first state
        stack = new Vector<State>();
        stack.add(states.get(0));
}

@Override
public void OnEnd() {
        // TODO Auto-generated method stub

}

@Override
public void onRelease() {
        // TODO Auto-generated method stub

}

//Create the table of State. It has been generated by code generator
private void GenerateStates()
{
        states = new Vector<State>();
        State e;
        // State 1 : Node 0
        e = new State();
        e.transitions = new Transition[1];
        e.transitions[0] = new Transition();
        e.transitions[0].condition = "cond_0_0";
        e.transitions[0].state = 1; //Go to node 1
        e.actions = "acc_0";
        states.add(e);

        // State 2 : Node 1
        e = new State();
        e.transitions = new Transition[1];
        e.transitions[0] = new Transition();
```

```
        e.transitions[0].condition = "cond_1_0";
        e.transitions[0].state = 0; //Go to node 0
        e.actions = "acc_1";
        states.add(e);

    }

    //Conditions was building based in sensors of the tool editor
    public Boolean cond_0_0()
    {
        return myRobotAPI.canKick() == true;
    }

    public Boolean cond_1_0()
    {
        return myRobotAPI.canKick() == false;
    }

    //Calls to Basic Behaviours build in the tool editor
    public void acc_0()
    {
        WalkTowardsBall();
    }

    public void acc_1()
    {
        Kick();
    }

    //Basic Behaviours of the tool editor from the game model the SoccerBots
    private void WalkTowardsBall()
    {
        myRobotAPI.setSteerHeading(myRobotAPI.getBall().t);
        myRobotAPI.setSpeed(1);
        myRobotAPI.setDisplayString("Walk towards ball");
    }

    private void Kick()
    {
        myRobotAPI.kick();
        myRobotAPI.setDisplayString("Kick");
    }

    /**
     * Strut to define a Transition of the HFMS
     * @author Jorge
     *
     */

    private class Transition {
        String condition;
        int state;
    }
    /**
     * Struct to define a State of the HFMS
     * @author Jorge
     *
     */
    private class State {
```

```
            Transition[] transitions;
            String actions;
        }
}
```