UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Ingeniería del Software e Inteligencia Artificial

TESIS DOCTORAL

Herramientas de asistencia al diseño de comportamientos inteligentes

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Gonzalo Flórez Puga

Directores

Pedro Antonio González Calero
Mª Belén Díaz Agudo

Madrid, 2013

Herramientas de Asistencia al Diseño de Comportamientos Inteligentes

TESIS DOCTORAL

Gonzalo Flórez Puga

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

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

Herramientas de Asistencia al Diseño de Comportamientos Inteligentes

Memoria que presenta para optar al título de Doctor en Informática

Gonzalo Flórez Puga

Dirigida por:
Pedro Antonio González Calero
Mª Belén Díaz Agudo

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

A mis abuelos

Agradecimientos

En primer lugar, agradecer a mis directores de Tesis, Belén y Pedro, su apoyo, colaboración y, sobre todo, su paciencia. Soy consciente de que ha sido un camino duro, tanto para mi como para ellos, pero siempre he tenido la seguridad de poder contar con su consejo y su comprensión.

Un agradecimiento muy especial a Carmen Chamizo, ya que ella fue quien me dio el empujón definitivo que me impulsó a tomar la decisión de seguir una carrera académica. Por el mismo motivo quería hacer extensivo este agradecimiento a Matilde Santos y Javier Crespo.

Marco y Pedro Pablo, gracias a los cuales incorporé la idea de los BTs a mi trabajo. A Guille, por esa magia que sólo el sabe hacer con R. A Almudena porque siempre se lee los agradecimientos. A mis restantes compañeros de despacho: Javi, David, Antonio, Lara, Chema, Virgina, Sammer, Manu, Fede y Raquel. Todos ellos siempre han tenido una palabra de ánimo cuando la he necesitado. Y, por supuesto, a todos los restantes miembros del Departamento de Ingeniería del Software e Inteligencia Artificial, que han sido grandes compañeros y han sabido alimentar mi curiosidad en las charlas del café y de las comidas.

A Boris, Vanessa y Pablo por ofrecerme su compañía, sus experiencias y su amistad, por participar en algo que a ellos ni les iba ni les venía, no solo desinteresadamente, sino de manera entusiasta, y por aportar un punto de vista fresco a mi trabajo desde su propia perspectiva. Y a todos los amigos y compañeros que he encontrado por el camino: Jesús, Jose, Antonio, Mito, Javi, Saúl, Toscano, Santi, Jacinto, María, Marta, Isabel, César, Virginia, Quique, Eva, Sergio, Desi, Bea, Manu, Andrés, Miguel, Elizabeth, Dimitri y unos cuantos más. Gracias a ellos, para bien o para mal, soy la persona que soy.

Quiero agradecer también el apoyo que me ha prestado mi familia desde los primeros años de estudiante hasta ahora. Especialmente a mis padres, que han sufrido de cerca los ratos malos sin apenas compartir los buenos (que también los ha habido).

Y dejo para el final lo más importante. Gracias a ti, Isabel, por mantenerte siempre a mi lado a pesar de todo, por pensar en mi las 24 horas del día y demostrármelo, por aceptarme, por quererme, por cambiarme, por

animarme. Y por todo aquello para lo que el lenguaje se queda demasiado corto.

Gracias, de corazón, a todos.

Este trabajo ha sido financiado por la Universidad Complutense de Madrid a través de la convocatoria de Becas de Formación de Personal Investigador.

Acerca de este documento

Esta Tesis Doctoral se presenta como compendio de publicaciones editadas, de acuerdo con el epígrafe 4.4 de la Normativa de desarrollo de los artículos 11, 12, 13 y 14 del Real Decreto 56/2005, de 21 de Enero, por el que se regulan los estudios universitarios oficiales de postgrado de la Universidad Complutense (Aprobado en Consejo de Gobierno con fecha 13 de Junio de 2005 y publicado en el BOUC con fecha de 5 de Julio de 2005).

Los artículos que se aportan como parte de la Tesis Doctoral son los siguientes:

- Gonzalo Flórez Puga, David Llansó, Marco Antonio Gómez Martín, Pedro Pablo Gómez Martín, Belén Díaz Agudo y Pedro Antonio González Calero. *Empowering Designers with Libraries of Self Validated Query-enabled Behaviour Trees.* En Artificial Intelligence for Computer Games. Springer. Marzo, 2011. P. 55–82. ISBN 978-1-4419-8187-5 (edición impresa) ISBN 978-1-4419-8188-2 (edición electrónica)

- Gonzalo Flórez Puga y Belén Díaz Agudo. *Semiautomatic Edition of Behaviours in Videogames.* Proceedings of AI2007, 12th UK Workshop on Case-Based Reasoning. Cambridge. Diciembre, 2007.

- Gonzalo Flórez Puga, Pedro Antonio González Calero, Guillermo Jiménez Díaz y Belén Díaz Agudo. *Supporting sketch-based retrieval from a library of reusable behaviours.* Expert Systems with Applications, Volume 40, Issue 2. Agosto 2012, P. 531–542. Factor de impacto: **2.203** Journal Citation Reports Science Edition 2011 (22 de 111 en Computer Science, Artificial Intelligence)

- Gonzalo Flórez Puga, Belén Díaz Agudo y Pedro Antonio González Calero. *Similarity Measures in Hierarchical Behaviours from a Structural Point of View.* Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference. Daytona Beach, Florida, 19-21 Mayo, 2010. P. 330–335.

- Gonzalo Flórez Puga, Belén Díaz Agudo y Pedro Antonio González Calero. *Experience-Based Design of Behaviors in Videogames.* Pro-

ceedings of the 9th European Conference on Advances in Case-Based Reasoning. Trier. 1-4 Septiembre, 2008. P. 180–194.

- Gonzalo Flórez Puga, Marco Antonio Gómez Martín, Pedro Pablo Gómez Martín, Belén Díaz Agudo y Pedro Antonio González Calero. *Query-Enabled Behavior Trees.* IEEE Transactions on Computational Intelligence and AI in Games. Volume 1, Issue 4. Diciembre, 2009. P. 298–308. Factor de impacto: **1.617** Journal Citation Reports Science Edition 2011 (16 de 103 en Computer Science, Software Engineering)

- Gonzalo Flórez Puga, Belén Díaz Agudo y Pedro González Calero. *Evaluating Sketch-based Retrieval Speed-up for Behaviour Design in Soccerbots.* 25th International Florida Artificial Intelligence Research Society Conference. En proceso de revisión.

Este documento, según la normativa vigente, expone una introducción al trabajo realizado, acompañada de una revisión del estado del arte del campo. También se describe el planteamiento de los objetivos del trabajo y se incluye una discusión integradora sobre el contenido de los artículos presentados. Finalmente, se incluyen las conclusiones del trabajo y una sección de referencias bibliográficas que integran y complementan las ya incluidas en los artículos que componen el núcleo de esta Tesis Doctoral.

Resumen

La creación de comportamientos inteligentes para las entidades que habitan en un videojuego no es una tarea trivial. Se trata en muchos casos de una labor difícil y tediosa. En primer lugar, es necesario identificar qué entidades del juego necesitan mostrar un comportamiento inteligente y qué tipo de comportamiento debe ser. A continuación hay que diseñar cada uno de los comportamientos y, por último, implementarlo e integrarlo en el juego. A esto hay que sumar el impedimento de que, en muchas ocasiones, los diseñadores del juego no tienen los conocimientos técnicos necesarios para programar los comportamientos dentro del mismo, por lo que es necesario establecer un canal de comunicación bidireccional con los programadores, que serán los encargados de integrar los comportamientos en el juego.

El conjunto de comportamientos creados para un videojuego a lo largo del proceso de desarrollo puede ser bastante amplio. Además, en muchos casos, dentro de un mismo videojuego o de diferentes videojuegos de un mismo género, encontraremos comportamientos muy parecidos o con fragmentos en común. Los diseñadores podrían beneficiarse de este hecho, reutilizando comportamientos previamente creados para construir comportamientos nuevos. Sin embargo, la ausencia de herramientas para gestionar repositorios de comportamientos obliga al usuario a revisar de manera manual todos los comportamientos que lo componen y comprobar si se ajustan a sus necesidades.

En esta Tesis se presentan diferentes técnicas para la asistencia al diseño de comportamientos inteligentes en videojuegos, siendo el punto común de todas ellas la reutilización de comportamientos previamente diseñados. Nuestra propuesta consiste en mejorar la reutilización de comportamientos proporcionando al usuario técnicas para realizar consultas que sean capaces de recuperar comportamientos. De esta manera, el proceso de revisión manual del repositorio se convierte en un proceso automático en el que el sistema ofrece al usuario diferentes alternativas para recuperar comportamientos similares al que desea construir.

Además, las consultas también se pueden realizar mientras el comportamiento está siendo ejecutado en el juego. De esta manera, el diseñador puede definir un comportamiento de alto nivel sin necesidad de entrar a definir los

detalles de nivel más bajo, sino especificando en su lugar una consulta que recupere los comportamientos de bajo nivel.

Como prueba del correcto funcionamiento de las técnicas propuestas se describe un editor de comportamientos, *eCo*, que hace uso de ellas, y una serie de experimentos que demuestran su efectividad.

Índice

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

La creación de un videojuego es un complicado proceso creativo y de ingeniería, que involucra, en general, a un grupo numeroso de personas con diferentes perfiles colaborando durante un periodo más o menos largo de tiempo.

Desde que aparecieran los primeros videojuegos hasta la fecha actual, se ha recorrido un largo camino. Las mejoras tecnológicas en ordenadores y consolas han permitido pasar de mostrar a los usuarios simples puntos de luz en una pantalla de televisión en blanco y negro a recrear inmensos mundos tridimensionales con física realista, de interactuar con potenciómetros a hacerlo con todo el cuerpo o de jugar solos a jugar con cientos de personas que pueden encontrarse a miles de kilómetros entre sí.

Esta evolución en la calidad de los juegos y en la forma de jugar ha ido acompañada también de un crecimiento de la exigencia de los jugadores. En una industria cada vez más competitiva, estos ya no se conforman con las mismas fórmulas aplicadas una y otra vez. Cada nuevo título que sale al mercado debe aportar suficientes innovaciones en los diversos aspectos que lo componen como para hacerlo atractivo para los jugadores potenciales. A partir de la década de los 90, gracias a la introducción de hardware más potente en procesadores y tarjetas gráficas, fue la componente visual la que marcó la diferencia. Hoy por hoy, la industria se enfrenta con un público más exigente, acostumbrado a los grandes alardes visuales mostrados en el cine y que busca una experiencia de juego más completa. Es por este motivo que la IA (Inteligencia Artificial) se ha convertido en un factor de gran importancia dentro del diseño de un videojuego.

Para abordar esta competitividad se ha hecho necesaria la especialización de los roles participantes en el desarrollo de un videojuego. Mientras que en el pasado eran los programadores quienes se ocupaban también del diseño y, en algunos casos incluso del arte, la separación entre los perfiles de programador

y de diseñador se ha hecho cada vez más patente con el paso del tiempo.

En el ámbito del desarrollo de un videojuego, el diseñador es el responsable del contenido creativo del juego. Su rol consiste en idear y desarrollar las reglas del juego, la estructura, la jugabilidad, la historia de trasfondo, los comportamientos de los personajes, etc. Aunque la creación de la IA ha sido una tarea de la que se han encargado tradicionalmente los programadores, tiene gran influencia en el diseño del juego. Mediante la IA, el diseñador puede construir diferentes comportamientos para los NPC (*Non Playing Character*, Personaje No Jugador) que permitan, por ejemplo, ayudar a contar la historia, aportar realismo al juego o, en definitiva, mejorar la experiencia de juego del jugador.

La creación de comportamientos inteligentes para los NPC no es una tarea trivial. Se trata en general de una labor difícil y en muchos casos repetitiva. En primer lugar, es necesario identificar qué entidades del juego necesitan mostrar un comportamiento inteligente y qué tipo de comportamiento debe ser (si debe ser agresivo con el jugador, debe ayudarle, huir de él, etc.). A continuación se debe diseñar cada uno de los comportamientos, implementarlo e integrarlo en el juego. Por último, es necesario realizar las pruebas correspondientes para verificar que funcionan de acuerdo a lo deseado, tanto individualmente como en conjunto, cuando el NPC interactúa con los restantes elementos del juego. A esto hay que sumar el impedimento de que, en muchas ocasiones, los diseñadores del juego no tienen los conocimientos técnicos necesarios para programar los comportamientos dentro del mismo, por lo que es necesario que comuniquen sus ideas a un programador para que realice este paso. Por lo tanto, la creación de la IA para un videojuego exige la colaboración entre los programadores y los diseñadores.

Para dar una idea más aproximada de la magnitud de esta tarea, según Hocking (2009), para el desarrollo de un juego como Far Cry 2 fue necesario el trabajo de una media de 150 personas que colaboraron durante 43 meses. Haciendo una estimación (bastante conservadora, por otra parte) de que el 20 % de ellos fueran diseñadores, da un total de 30 diseñadores trabajando durante tres años y medio para crear el contenido de un videojuego de acción.

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

Esta característica se puede aprovechar para ahorrar trabajo a la hora de construir un nuevo comportamiento, utilizando los comportamientos más simples como bloques de construcción para construir los comportamientos más elaborados. De esta manera, se facilita el proceso de desarrollo ya que, por un lado los comportamientos nuevos no se deben construir desde cero, ahorrando tiempo y esfuerzo. Además, utilizando las técnicas existentes para la representación de comportamientos, los diseñadores no tienen la necesidad de escribir código para crearlos: pueden combinar los comportamientos y realizar la generación de manera automática.

Para aliviar los problemas derivados de la complejidad de la creación de comportamientos y de la colaboración entre diseñadores y programadores, en esta Tesis Doctoral se presenta un nuevo modelo de creación de comportamientos inteligentes para las entidades de videojuegos que está basado en la reutilización de comportamientos y fragmentos de comportamientos almacenados en una biblioteca.

1.2. Objetivos

A grandes rasgos, el objetivo principal de esta Tesis es aportar los medios para facilitar a los diseñadores la tarea de creación de comportamientos para los NPC de un videojuego.

Como mencionábamos anteriormente, la creación de comportamientos para los NPC es una tarea compleja que requiere la colaboración de diseñadores y programadores. La solución que presentamos a este problema, y que ha supuesto el centro de esta investigación, es la utilización de una biblioteca con comportamientos y fragmentos de comportamientos reutilizables. Utilizando las técnicas que explicaremos más adelante los diseñadores pueden realizar búsquedas dentro de esta biblioteca y recuperar comportamientos previos que se ajusten a sus necesidades. De esta manera, en lugar de "reinventar la rueda" cada vez que necesitan un comportamiento, los diseñadores pueden reutilizar el conocimiento previo creado por ellos mismos o por otros expertos.

Como paso previo a la consecución de este objetivo, en primer lugar se ha realizado un estudio del proceso de desarrollo de videojuegos, con la finalidad de identificar cuáles son las limitaciones de este proceso y cómo se pueden mitigar. Por otro lado, también se ha llevado a cabo una revisión de las diferentes técnicas utilizadas en la industria de los videojuegos para la creación y representación de comportamientos de los NPC. El objetivo de este estudio ha sido dar con las técnicas más utilizadas en la industria y, dentro de ellas, discriminar cuáles son las que pueden aportar más ayuda a los diseñadores. Los resultados de estos estudios previos quedan plasmados en el Capítulo 2 de esta Tesis.

Partiendo de las conclusiones obtenidas en estos estudios previos se ha

desarrollado un modelo de creación de comportamientos inteligentes basado en la reutilización de comportamientos completos y fragmentos de comportamientos almacenados en una biblioteca. Este modelo ha ido evolucionando a lo largo del tiempo de vida de la investigación. En el Capítulo 3 se describe en detalle el modelo.

La consecución del objetivo planteado se ha abordado desde diferentes frentes:

- Propuesta de diferentes mecanismos de representación de comportamientos y de medidas de similitud asociadas a ellos que permitan la recuperación eficiente de comportamientos de la biblioteca.
- Propuesta de diferentes técnicas para la construcción de comportamientos complejos a partir de otros creados previamente.
- Diseño de herramientas que soporten el modelo propuesto.
- Demostración experimental de la validez de las técnicas y herramientas propuestas.

Estos puntos y las contribuciones realizadas a lo largo de la investigación en cada uno de ellos están desarrollados en el Capítulo 4. En el Capítulo 5 se exponen las conclusiones de esta Tesis junto con un resumen de las líneas de trabajo futuro. Finalmente, en el Capítulo 6 se encuentran recopiladas las publicaciones que se aportan como parte de esta Tesis Doctoral en su formato original.

1.3. Resumen de las Contribuciones

Las principales contribuciones de esta Tesis están ligadas al modelo antes mencionado. En primer lugar, como se mencionaba al comienzo de esta sección, se ha realizado un estudio acerca del proceso de desarrollo de videojuegos y de las distintas técnicas para representar comportamientos para los NPC.

En lo referente al modelo en sí mismo, se han establecido los requisitos que debe cumplir una biblioteca de comportamientos reutilizables. Estos requisitos se recogen en la publicación (Flórez-Puga et al., 2011).

Por otro lado, se ha propuesto un conjunto de técnicas de reutilización diferentes que se aplican a los comportamientos en la biblioteca. Para hacer posible el funcionamiento de estas técnicas, también se ha propuesto un conjunto de funciones de similitud de diferentes tipos, adecuadas para los comportamientos almacenados en la biblioteca.

En primer lugar se encuentra la *recuperación basada en funcionalidad*, que permite recuperar los comportamientos de acuerdo a una descripción

realizada por el usuario de la funcionalidad deseada en los mismos. Esta técnica se describe en la publicación (Flórez-Puga y Díaz-Agudo, 2007), donde también se describen las funciones de similitud empleadas, y se refina posteriormente en (Flórez-Puga et al., 2012a)[1].

La segunda de las técnicas propuestas es la *recuperación basada en bocetos*, una novedosa aproximación a la recuperación de comportamientos que permite realizar búsquedas en la biblioteca a partir de una representación aproximada (un boceto) de los comportamientos. Esta técnica se describe en detalle en (Flórez-Puga et al., 2012a). La recuperación basada en bocetos utiliza funciones de similitud estructural para llevar a cabo la recuperación de los comportamientos. En (Flórez Puga et al., 2008) se comparan diferentes funciones de similitud estructural y su aplicación a los comportamientos. En (Flórez-Puga et al., 2010) se realiza una comparación de las funciones de similitud propuestas y se realiza un estudio de su aplicabilidad y su eficiencia.

Por último, la *recuperación en ejecución* permite aplicar estas técnicas cuando el comportamiento está siendo ejecutado en el juego, en lugar de durante el diseño del mismo, mediante los *nodos consulta*. Esta aproximación se recoge en la publicación (Flórez-Puga et al., 2009)[2].

Además, se ha realizado una serie de experimentos demostrando la validez de las técnicas empleadas. En concreto, en (Flórez-Puga et al., 2012a) se evalúa la correlación existente entre la similitud estructural y la similitud funcional en los comportamientos, de manera que podemos afirmar que los comportamientos recuperados mediante *recuperación basada en bocetos*, no solo van a poseer una estructura similar, sino que también van a mostrar un comportamiento similar a la hora de ser ejecutados. Por otra parte, en los experimentos desarrollados en (Flórez-Puga et al., 2013) se demuestra que las técnicas desarrolladas realmente suponen una mejora en cuanto al tiempo que es necesario emplear para la construcción de comportamientos para las entidades de un videojuego.

Para terminar, siguiendo las directrices del modelo se ha desarrollado una herramienta de edición de comportamientos en la que se han incorporado las diferentes técnicas de recuperación investigadas. Esta herramienta, llamada *eCo*, ha permitido evaluar con usuarios reales la eficiencia y la efectividad de las técnicas propuestas y ha sido de gran ayuda en la realización de los experimentos. En la publicación (Flórez-Puga y Díaz-Agudo, 2007) se presenta una versión inicial de la herramienta haciendo énfasis en la recuperación basada en funcionalidad. En la publicación posterior (Flórez-Puga et al., 2012a) se presenta la última versión de *eCo*, donde se incide en mayor

[1] Publicado en *Expert Systems With Applications*, factor de impacto: 2.203 Journal Citation Reports Science Edition 2011 (22 de 111 en Computer Science, Artificial Intelligence)

[2] Publicado en *IEEE Transactions on Computational Intelligence and AI in Games*, factor de impacto: 1.617 Journal Citation Reports Science Edition 2011 (16 de 103 en Computer Science, Software Engineering)

medida en la recuperación basada en bocetos.

Capítulo 2

Estado del Arte

2.1. Proceso de Desarrollo de un Videojuego

Según Arévalo (2005), los videojuegos son "obras de creación que exigen de la combinación de talento, técnica y creatividad". El desarrollo de un videojuego es una actividad multidisciplinaria que puede involucrar a profesionales de diversos campos técnicos o artísticos. Dentro del ámbito artístico se incluyen todos los elementos estéticos que se muestran al jugador, tales como las texturas, el diseño gráfico de los personajes, o el sonido y la música. Pero también se encuentra en este apartado el diseño del juego: el conjunto de características que dan forma al juego, tales como la jugabilidad, las reglas que lo conforman, el guion, etc. Dentro del ámbito técnico se incluye la construcción del motor del juego y de las herramientas que utilizará el resto del equipo para crear la experiencia de juego del jugador.

Los roles principales que participan en el desarrollo de un videojuego son, por tanto:

- **Diseñadores**

  Los diseñadores se encargan de crear el concepto inicial y de desarrollarlo hasta alcanzar un juego completo, concibiendo las reglas del juego y su estructura. Son los que aportan la visión global acerca del juego, especificando, por ejemplo, la historia, los personajes, el contexto o los diferentes escenarios que tienen que existir. Entre sus atribuciones destacan la creación de las mecánicas básicas del juego, el diseño de los controles, la creación del guion y los diálogos, etc.

- **Grafistas**

  Los grafistas se encargan de crear el arte que se mostrará en el juego, dando forma a los personajes y al universo en el que se mueven. Algunas de las tareas de las que se encargan son, por ejemplo, la creación del arte de concepto, el modelado de personajes, la creación de animaciones

o la creación de las texturas.

- **Programadores**

  La principal tarea de los programadores es convertir en realidad aquello que han imaginado los diseñadores, creando e integrando el motor del juego, las herramientas de creación de contenidos, la lógica, etc. Deben evaluar los riesgos técnicos de las diferentes partes del diseño para decidir qué se debe implementar antes o qué partes se deben implementar y en cuáles se puede recurrir a tecnología externa (por ejemplo, si desarrollar el motor gráfico o usar un librería).

A estos roles hay que añadir también otros que realizan tareas que no están tan relacionadas con el desarrollo en sí, pero que son de gran importancia para el buen término del mismo. Por ejemplo, los roles de gestión, como el de los productores, que se encargan de hacer de intermediarios entre las partes implicadas en el juego (generalmente, entre el editor y el desarrollador), gestionar el desarrollo y el equipo, asegurarse del cumplimiento de plazos, etc, o los *testers*, cuya misión es asegurarse que el producto final esté libre de errores de cualquier tipo.

En general, un videojuego se desarrolla en varias fases. Aunque en distintos textos encontramos diferente separación en fases, la mayoría de las diferencias entre ellos están en cómo se agrupan las tareas que conforman las fases y no en las tareas en sí mismas. Por ejemplo, en algunos textos (Chandler, 2005), la fase de preproducción incluye también las tareas que en otros se realizan durante la fase de diseño (Bates, 2004). Podemos distinguir las siguientes fases:

- **Concepto**

  El objetivo de esta fase es decidir de qué trata el juego, definiendo la idea principal del juego. Este concepto inicial suele ser muy simple, sólo una o dos frases que concentran las ideas principales y que serán refinadas y desarrolladas posteriormente. Durante esta fase también se pueden decidir los elementos del juego o esbozar el estilo artístico y la historia.

- **Preproducción**

  Partiendo del concepto se elabora el documento de diseño ampliado, en el que se recogen los detalles del juego. Típicamente, el documento de diseño contendrá datos acerca de la historia de trasfondo del juego, los distintos escenarios, las mecánicas de juego, las habilidades y tipos de personaje que aparecerán (tanto del bando del jugador como de los enemigos), las armas o hechizos, los objetos de inventario, los controles, la interfaz de usuario, distintos modos de juego, etc. En el documento de diseño, además, se incluirá una sección acerca de la IA. En ella se

detallan los aspectos relevantes para la IA del juego, tales como las propiedades de los NPC que serán utilizadas por la IA (por ejemplo, el campo de visión o la velocidad de movimiento), cómo se realiza el movimiento (*pathfinding*) y los diferentes comportamientos que puede mostrar cada NPC.

Dentro del propio documento de diseño, o en otro documento externo, también se define el plan de producción artístico, en el cual se detalla cuál es el aspecto general que se desea que tenga el juego y se crea el arte de concepto, que servirá de referencia a todos los artistas para crear los escenarios, los personajes y los restantes elementos que aparecerán en el juego. En este plan también se recoge el camino que seguirá cada pieza del arte del juego para pasar de arte de concepto a ser utilizada dentro del juego, detallando las herramientas, programas y procesos de exportación o transformación necesarios.

Sobre la extensión y el contenido del documento de diseño tampoco existe un acuerdo en la industria. Mientras que en algunos casos, se recogen todos los detalles acerca del juego, convirtiéndose en lo que algunos llaman la *biblia* de diseño del juego, en otros se opta por un documento más escueto y manejable.

Es también durante la fase de preproducción, durante la que se explora cuáles son las características del juego que aportan la diversión. Es, por tanto, habitual durante esta fase la construcción de prototipos mediante los que se prueben las diferentes ideas de diseño, descartando algunas y añadiendo otras nuevas.

Por último, durante esta fase también se crean las herramientas que se utilizarán en producción para desarrollar la aplicación final, como las herramientas de diseño de niveles, los exportadores que utilizarán los artistas para incorporar sus modelos al juego, el motor gráfico, etc.

- **Producción**

  En la fase de producción es en la que el equipo produce todos los contenidos de la aplicación final, que darán forma a la experiencia de juego del jugador. Durante esta fase es cuando el equipo de desarrollo alcanza su tamaño máximo, ya que es necesario incorporar a un amplio equipo de artistas para generar los gráficos, modelos y texturas, diseñadores de niveles que construirán los entornos donde se desarrollará el juego y programadores que se encargarán de construir el motor de ejecución y de refinar las herramientas creadas en la preproducción para adaptarlas a las necesidades de los diseñadores y grafistas. Tras esta etapa se obtiene una versión completa del juego, con todas sus fases, niveles, menús y herramientas.

- **Postproducción**

  En esta fase se pule el contenido creado realizando el control de calidad en busca de érrores. Para ello, un equipo de *testers* (que puede ser el propio equipo de desarrollo si el juego es pequeño, aunque no es lo deseable) debe probar concienzudamente cada uno de los aspectos del juego: niveles, sonido, localización, etc.

  En esta fase, además, se cierra el proyecto, generando una serie de documentos que pueden servir como experiencia para futuros desarrollos.

- **Mantenimiento**

  Con el final del proceso de desarrollo se obtiene una versión del juego que será la que verán los jugadores. Pero, en la mayoría de los casos y a pesar del cuidado que se pueda tener en la detección de errores, es habitual que en la versión final existan algunos errores del juego (*bugs*) o que los desarrolladores deseen realizar alguna modificación para mejorarlo (por ejemplo, de equilibrado). Estas modificaciones se realizan a posteriori, durante la fase de mantenimiento del juego, mediante la publicación de parches.

A pesar de que hemos separado el proceso de desarrollo en diferentes fases, estas fases no suelen darse aisladas, sino que se solapan unas con otras. Por ejemplo, aunque el concepto del juego se fija al principio, este suele refinarse a lo largo de todo el proceso de desarrollo.

2.1.1. La Inteligencia Artificial en el Proceso de Desarrollo de un Videojuego

Como se ha discutido en la sección anterior, el objetivo de un juego es que el jugador disfrute de su experiencia durante el juego, que se divierta. La IA debe ser un ingrediente más hacia la consecución de este objetivo. En este sentido, el enfoque de la IA para juegos difiere del de la IA puramente académica. En este caso, no se trata de optimizar, no se trata de conseguir que los NPC superen al jugador, sino de que el juego sea un reto para el jugador. Para ello es posible que sea necesario reducir las capacidades de la IA de los NPC. Por ejemplo, en el caso de un FPS (*First Person Shooter*), si los soldados enemigos son capaces de acertar al jugador en la cabeza nada más verle, el jugador no sólo perderá interés dada la dificultad del juego, sino que se sentirá frustrado, puesto que, a pesar de enfrentarse contra un NPC de aspecto humano, sus capacidades son sobrehumanas.

Aunque tradicionalmente la tarea de crear la IA de los NPC ha sido una tarea de programación, es innegable que tiene una componente importante de diseño. Mediante la IA, el diseñador indica a los NPC qué es lo que deben hacer en cada momento. Esta tarea creativa puede servir para conseguir otros objetivos de diseño (Rouse y Ogden, 2005):

- Gracias a una IA nivelada, el juego puede convertirse en un reto para el jugador. Sin una buena IA el juego será demasiado fácil y predecible o, por el contrario, demasiado difícil y frustrante para el jugador.

- La IA influye en la jugabilidad. Diseñando una IA que se comporte de diferente manera en cada partida, ofrecemos al jugador un comportamiento más realista de los NPC y hacemos que cada experiencia de juego sea diferente, dando al juego el valor añadido de la posibilidad de jugarlo varias veces. Un comportamiento no predecible, además, puede formar parte del reto que supone el juego. Evidentemente, esta impredecibilidad debe ir acorde con el juego, el escenario o el tipo de personaje. No tendría mucho sentido, por ejemplo, que cuando un soldado se sintiera amenazado, comenzara a cacarear como una gallina, a no ser que el propio juego así lo precise.

- La IA puede ser una herramienta útil para contar historias. Además de usar *cutscenes*, vídeos o textos, el diseñador puede apoyarse en los comportamientos que muestran los NPCs para contar la historia del juego.

- La IA puede aportar realismo al universo donde se desarrolla el juego. El juego puede incluir personajes con los que el jugador no interactúe directamente (personas que se desplazan a su trabajo o, si el juego se desarrolla en el campo, pequeños animales), pero que se comporten de una manera realista dentro del juego. De esta manera, la ilusión de que el jugador está interactuando con un mundo real es mucho mayor.

2.2. Inteligencia Artificial en Videojuegos

La utilización de técnicas de inteligencia artificial ha estado ligada a los videojuegos prácticamente desde el origen de estos, hacia los años 70. Los primeros videojuegos en incorporar IA utilizaban conjuntos de reglas simples y acciones, combinadas con toma de decisiones aleatoria para conseguir comportamientos menos predecibles. A partir de entonces, la aplicación de técnicas de inteligencia artificial ha acompañado en su evolución a los videojuegos, tímidamente al principio, convirtiéndose con el paso del tiempo en un componente fundamental y diferenciador.

Generalmente, la ejecución de un juego consiste en la ejecución de un bucle, que recibe el nombre de bucle de juego. Este bucle se repite continuamente, hasta que el usuario decide salir del juego. En cada iteración se realizan todas las tareas necesarias para el desarrollo del juego, como la captura de la entrada de los periféricos de control (ratón, teclado, mando de juego...), el cálculo de la influencia del modelo físico en las entidades del juego, el render (la representación gráfica por pantalla del entorno de juego)

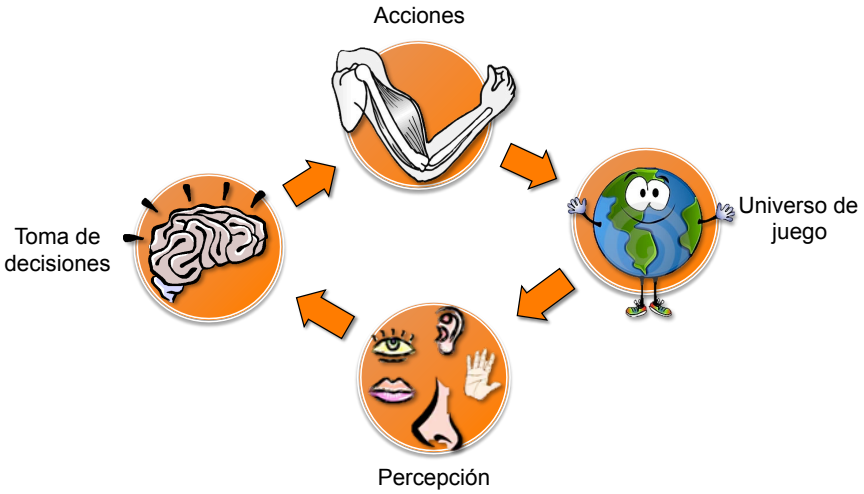


Figura 2.1: Ciclo de comportamiento de un NPC

o la ejecución de los comportamientos de las entidades del juego. En cuanto a la estructura del bucle, existen varias aproximaciones que ofrecen distintas ventajas e inconvenientes (Val, 2005), si bien, queda fuera del alcance de este trabajo profundizar en las características de cada una de ellas.

Un comportamiento se puede definir como un conjunto de acciones o reacciones que realiza una entidad, generalmente en relación con su entorno. Esto es particularmente cierto en el ámbito de los videojuegos. En términos generales, una entidad dentro de un videojuego recopila información sobre el entorno mediante una serie de sensores, equiparables a los sentidos de los seres vivos, y, dependiendo de ésta, realiza determinadas acciones. Para llevar a cabo estas acciones utiliza los actuadores. Los actuadores de las diferentes entidades provocan cambios en el estado del mundo donde se desarrolla el juego. La información sobre estos cambios será, de nuevo, percibida mediante los sensores, dando comienzo de nuevo al ciclo. Esta idea se puede ver de manera más gráfica en la Figura 2.1.

En los sistemas sensoriales de los juegos, la inteligencia artificial observa periódicamente el entorno en el que habita, en contraste con los sentidos reales, que son estimulados independientemente de si se desea o no (no podemos dejar de oír o de ver a voluntad). De esta manera se pueden emular las capacidades sensoriales limitando la cantidad de recursos utilizada para ello (Leonard, 2003).

El módulo de toma de decisiones es el responsable de decidir qué actuadores es necesario activar en cada momento para un determinado estado de los sensores. Es en este módulo donde tendrán lugar las decisiones que guiarán el comportamiento. La interfaz que utiliza la inteligencia artificial para comunicarse con el juego será precisamente el conjunto de sensores, a

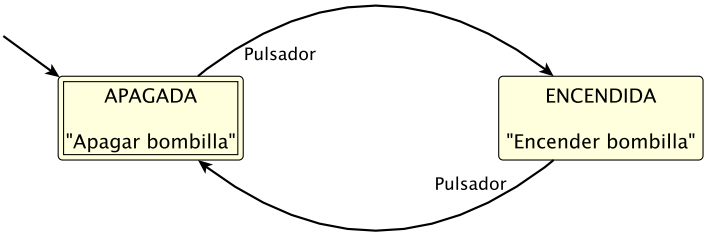


Figura 2.2: Máquina de estado finito correspondiente al comportamiento de una lámpara

través de los que recibe la información acerca de su entorno, y el conjunto de actuadores, que le permiten llevar a cabo las acciones necesarias.

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

2.2.1. Máquinas de Estado Finito

Una máquina de estado finito es un modelo de computación con una cantidad limitada de memoria, que recibe el nombre de estado. Cada máquina tiene un número finito de estados posibles y una función de transición que determina cómo cambia el estado de acuerdo a las entradas (Champandard, 2003). Formalmente, una máquina de estado finito queda definida por 6 elementos $A = \{\Sigma, Q, \delta, q_0, O, \lambda\}$, donde:

- Σ es el alfabeto de entrada, es decir, el conjunto de símbolos que pueden encontrarse como entradas a la máquina de estado y que serán los que provoquen los cambios de estado,
- Q es el conjunto de estados,
- $\delta : Q \times \Sigma \rightarrow Q$ es la función de transición. Esta función asigna a cada par formado por un estado de origen y un símbolo de entrada (q, v) un estado de destino q', de manera que cuando está en el estado q y lee el símbolo v en la entrada, el estado pasa a ser $\delta(q, v) = q'$,
- q_0 es el estado inicial,
- O es el alfabeto de salida,
- λ es la función de salida. En general, en las máquinas de estado finito aplicadas al control de entidades de un videojuego, la función de salida sólo depende del estado actual $\lambda : Q \rightarrow O$.

Por ejemplo, supongamos que tenemos una lámpara controlada por un interruptor de pulsador. Al pulsar el interruptor, si la lámpara está encendida, se apagará, y si está apagada se encenderá. Podemos modelar el comportamiento de la lámpara mediante una máquina de estado finito como la de la figura 2.2. La lámpara puede encontrarse en dos estados diferentes: $Q = \{\text{encendida}, \text{apagada}\}$. El alfabeto de entrada, Σ, sólo tiene un símbolo, el correspondiente a la activación del pulsador. Cada pulsación en el interruptor hace que cambie de estado, por lo que la función de transición, δ, se define mediante la siguiente tabla:

$$\begin{array}{rcl} (\text{encendida}, \text{pulsador}) & \rightarrow & \text{apagada} \\ (\text{apagada}, \text{pulsador}) & \rightarrow & \text{encendida} \end{array}$$

El estado inicial q_0 será apagada (suponiendo que la lámpara se encuentre inicialmente en este estado). Por último, el alfabeto de salida O estará compuesto por las acciones necesarias para controlar la lámpara en cada estado. En este caso $O = \{\text{encender bombilla}, \text{apagar bombilla}\}$. Mediante la función λ asignamos los valores del alfabeto de salida a cada uno de los estados:

$$\begin{array}{rcl} \text{encendida} & \rightarrow & \text{encender bombilla} \\ \text{apagada} & \rightarrow & \text{apagar bombilla} \end{array}$$

Dentro de nuestro dominio de estudio, la representación de comportamientos para NPC de videojuegos, podemos acotar esta definición. En concreto, el alfabeto de entrada debe recoger el conjunto de símbolos que pueden hacer que la máquina abstracta cambie de estado. En el caso de un NPC, los cambios de estado se producirán cuando perciba determinadas condiciones en su entorno. Por lo tanto, el alfabeto de entrada estará formado por condiciones definidas usando los sensores del NPC. Por ejemplo, serían condiciones válidas `salud < 10%` o `distancia_objetivo > 25`. Lo mismo sucede con el alfabeto de salida. En este caso, el alfabeto de salida estará formado por el conjunto de acciones que puede realizar el NPC, es decir, el conjunto de actuadores.

Como hemos visto en los ejemplos anteriores, una máquina de estado se puede representar como un grafo dirigido, en el que cada estado se corresponde con un nodo y cada transición con una de las aristas del grafo. Cada uno de los nodos del grafo estará etiquetado con la acción correspondiente al estado que representa. De la misma forma, las aristas se etiquetarán con las condiciones que las activan.

Las máquinas de estado proporcionan un modelo estructurado para la creación de comportamientos que facilita la tarea de los diseñadores de videojuegos. No en vano han sido el estándar *de facto* en la industria de los videojuegos durante mucho tiempo y aún siguen siendo ampliamente utilizadas. Además, el hecho de tener una representación gráfica hace que su

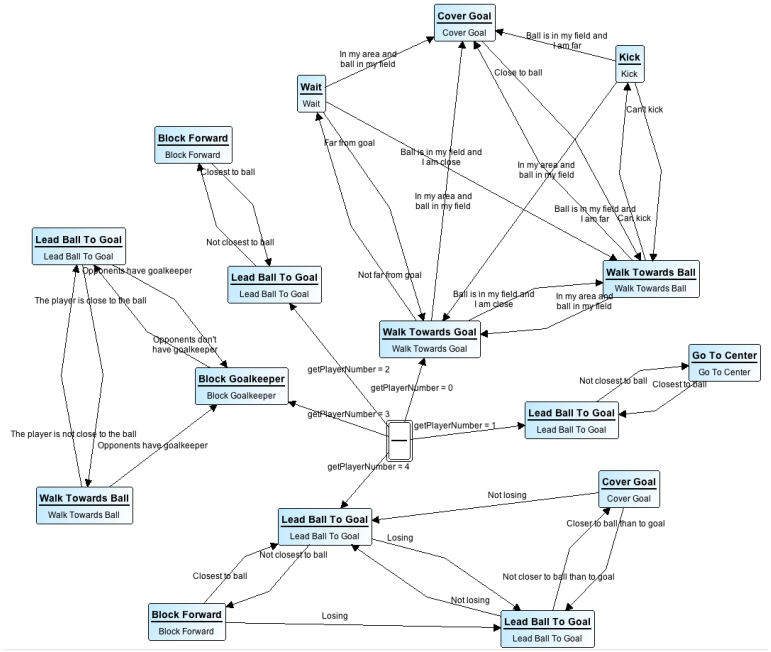


Figura 2.3: El mantenimiento de una FSM se puede complicar al aumentar el número de nodos y aristas

diseño sea más intuitivo. Mediante la utilización de herramientas, los diseñadores pueden centrarse únicamente en el diseño de los comportamientos en un entorno visual más amigable, delegando otras tareas que pueden ser ajenas a sus conocimientos, como la generación del código correspondiente al comportamiento, en la herramienta.

El problema principal de las máquinas de estado es su mala escalabilidad. Al añadir nuevos estados y transiciones a una máquina de estado, su diseño y mantenimiento se pueden volver muy complicados. En la Figura 2.3 podemos ver, por ejemplo, una máquina de estado con 16 estados y 34 aristas. Una solución a este problema, como veremos a continuación, es el uso de HFSM (*Hierarchical Finite State Machine*, Máquina de Estado Finito Jerárquica).

2.2.1.1. Máquinas de Estado Finito Jerárquicas

El concepto de HFSM se basa en los *statecharts*, un formalismo para describir estados y transiciones de manera modular, permitiendo concurrencia y agrupamiento (Harel, 1987). La idea consiste en agrupar los estados en estructuras llamadas *superestados*, que pueden compartir las transiciones. Así, un superestado puede ser origen de una transición a otro estado, indicando que todos los estados contenidos en él comparten esta transición.

En muchos casos, esta idea se refina aún más haciendo que cada superestado funcione como un estado normal y que, en lugar de solamente agrupar

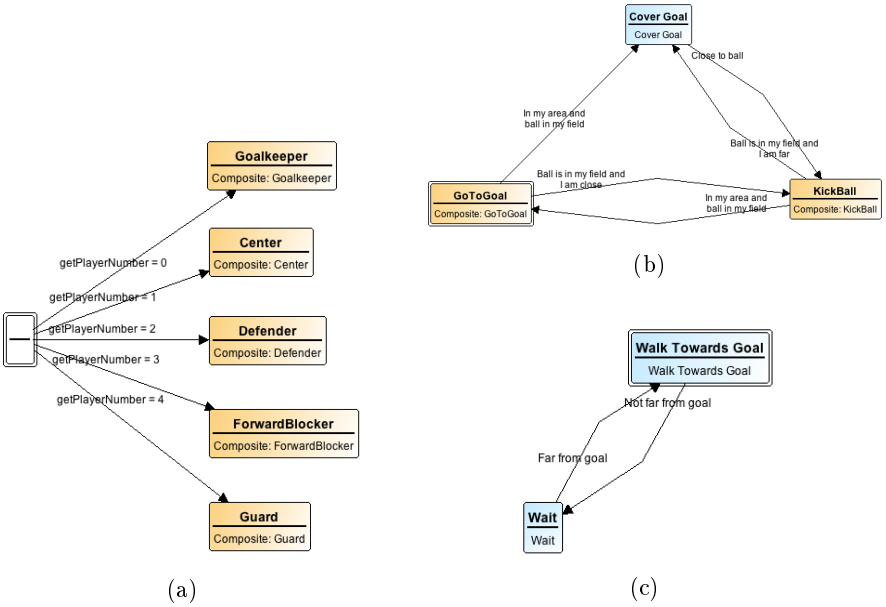


Figura 2.4: Representación jerárquica del comportamiento de la Figura 2.3. En (a) se muestra el comportamiento de más alto nivel, en (b) el contenido del supernodo Goalkeeper y en (c) el de Go To Goal

estados, pueda contener otra HFSM completa. Aunque ambas definiciones se aplican al concepto de HFSM (Champandard, 2007), cuando hagamos referencia en el texto a este concepto lo haremos refiriéndonos a la segunda de ellas.

La figura 2.4 muestra un ejemplo HFSM que se ajusta a esta definición. Esta máquina de estado es una versión "jerarquizada" del ejemplo de la figura 2.3. En la figura se muestra el nivel superior de la jerarquía y las máquinas de estado contenidas en los superestados *Goalkeeper* y *Go To Goal*. Como se puede ver, la introducción de la jerarquía simplifica la representación de la HFSM facilitando su mantenimiento.

La ejecución de una HFSM no difiere mucho de la ejecución de una FSM (*Finite State Machine*, Máquina de Estado Finito) simple. De hecho, cualquier HFSM puede ser "aplanada" para convertirla en una FSM siguiendo un sencillo algoritmo. Igual que en una FSM, la ejecución comienza en el estado inicial. Cuando se hace cierta alguna de sus transiciones, se cambia de estado al destino de la transición. La principal diferencia entre ambas la encontramos en la ejecución de los estados. Si el estado a ejecutar es un estado simple, como los que encontramos en una FSM, se procede de la misma forma, ejecutando los actuadores asociados al estado. Pero si se trata de un superestado, la ejecución pasa al nodo inicial de la máquina de

estado contenida en el superestado. A partir de este momento, es necesario comprobar las transiciones de los estados que están siendo ejecutados en todos los niveles de la jerarquía, de manera que las transiciones se puedan producir a todos los niveles.

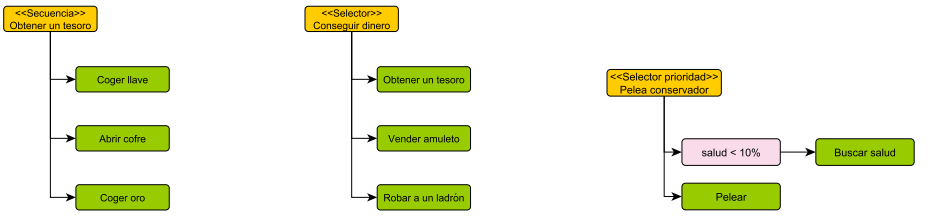
Por ejemplo, supongamos que una entidad está ejecutando el comportamiento de la Figura 2.4. La ejecución comenzaría en el estado inicial de 2.4(a), marcado en la figura con un doble contorno. Este estado no tiene ninguna acción asociada, por lo que la entidad no realizará acciones mientras se encuentre en él. Supongamos que se activa la primera transición. El nuevo estado será *Goalkeeper*. Como se trata de un superestado, en lugar de ejecutar una acción se ejecuta la máquina de estado contenida en él (2.4(b)). De nuevo, se comienza la ejecución en el estado inicial *Go To Goal* y de nuevo se trata de un superestado, por lo que el estado a ejecutar pasa a ser *Walk Towards Goal*, el estado inicial de la máquina de estado correspondiente, representada en la Figura 2.4(c). Si en algún momento se activa la transición *Not far from goal*, el nuevo estado pasará a ser *Wait*. Pero si se activa la transición del estado a algún nivel más alto de la jerarquía también debe ser tenida en cuenta. Así, por ejemplo, si se hace cierta la condición *In my area and ball in my field*, independientemente de si el estado actual es *Walk Towards Goal* o *Wait*, el nuevo estado pasará a ser *Cover Goal*.

Otra ventaja de las HFSM es que la jerarquía proporciona diferentes niveles de abstracción. Así, en los niveles más altos se representarán las tareas más abstractas, descomponiendo estas en tareas más simples según se baja en la jerarquía. De esta manera, cada superestado puede funcionar como una caja negra, ocultando los detalles del nivel inferior. La abstracción, además, permite diseñar el comportamiento siguiendo un enfoque top-down que resulta mucho más intuitivo para los diseñadores, creando en primer lugar los estados con comportamientos de más alto nivel y descomponiéndolos recursivamente en subcomportamientos más simples.

2.2.2. Behaviour Trees

Los BT (*Behaviour Tree*, Árbol de Comportamiento) conforman una nueva tecnología para el control de NPC en videojuegos, cuyo uso está cada vez más extendido en juegos de diferentes géneros, a juzgar por el número de publicaciones que han aparecido sobre ellos, por ejemplo, en la prestigiosa serie AI Game Programming Wisdom (Rabin, 2006, 2008), así como de diversos autores pertenecientes a la industria de los videojuegos (Isla, 2005, 2008; Pillosu, 2009).

Un BT se representa como un árbol. La ejecución comienza en la raíz; los nodos intermedios son los encargados de decidir el flujo de ejecución, mientras que las hojas contienen los comportamientos básicos con las acciones del juego que ejecutará el NPC. Todos los nodos ejecutados, tanto los nodos intermedios como las hojas, al terminar su ejecución devuelven un estado



(a) Nodo secuencia (b) Nodo selector (c) Nodo selector con prioridad

Figura 2.5: Distintos ejemplos de Behaviour Trees

de finalización, que puede ser ÉXITO o FALLO. El nodo padre utiliza esta información de finalización para decidir el flujo de ejecución de sus hijos, para comprobar si ha terminado su ejecución y para devolver su propio estado de finalización a su nodo padre.

Aunque no existe un acuerdo sobre cuál debería ser el conjunto de nodos de decisión, existen diferentes propuestas en la literatura . Para todos ellos, el orden en el que se definen los hijos es importante. A continuación describimos los nodos de decisión más comunes:

- **Secuencias**

  El nodo secuencia ejecuta la lista de hijos en el orden en que se han definido, uno tras otro. Cuando uno de los hijos de la secuencia termina con éxito, el siguiente hijo se lanza inmediatamente. Si alguno de los hijos falla, se considera que el comportamiento falla, con lo que el fallo se pasa hacia arriba en la jerarquía. La idea intuitiva de este tipo de nodos es que el comportamiento que se plantea se puede resolver descomponiéndolo en un conjunto de comportamientos subordinados que deben ser ejecutados en orden. Por lo tanto, el objetivo sólo se resuelve si todos los hijos tienen éxito. Por ejemplo, si el objetivo de un comportamiento es `Obtener un tesoro` que se encuentra dentro de un cofre cerrado, el árbol de comportamiento correspondiente podría estar formado por un nodo secuencia con tres hijos: `Coger llave`, `Abrir cofre` y `Coger oro` (Figura 2.5(a)). Esto significa que para conseguir el tesoro es necesario ejecutar los tres comportamientos en orden.

- **Selectores**

  El funcionamiento del nodo selector es complementario al del nodo secuencia. En este caso también se comienza ejecutando el primero de los hijos pero, al contrario que en el anterior, si termina con éxito, la ejecución del nodo selector se considera que también ha terminado con éxito. Por el contrario, si falla se intenta ejecutar el siguiente de los hijos. Únicamente se considera que el selector falla si fallan todos sus

hijos. En este caso, la idea es que un selector ofrece distintas alternativas para resolver un comportamiento, una por cada uno de sus hijos. Para que falle, todas las posibles alternativas tendrán que fallar. El orden de los hijos indica la prioridad de una de las alternativas. En el comportamiento `Conseguir dinero` de la Figura 2.5(b) vemos un ejemplo de un nodo selector. Para conseguir dinero podemos `Obtener un tesoro`, `Vender amuleto` o `Robar a un ladrón`. Cada uno de los comportamientos se intenta por orden de prioridad, siendo más prioritario el primero (`Obtener un tesoro`). En el momento en que alguno de estos tres comportamientos termine con éxito, no será necesario ejecutar ninguno de los restantes, ya que habremos conseguido el dinero, por lo que el comportamiento `Conseguir dinero` también terminará con éxito.

Dependiendo de la literatura consultada, los hijos de los nodos selectores pueden estar guardados, además, por una condición que indica en qué circunstancias ese hijo puede ser seleccionado para resolver el comportamiento del padre (Pillosu, 2009). Antes de intentar ejecutar el hijo correspondiente se hace una comprobación sobre la condición. Si esta no se cumple, el nodo no se ejecuta, sino que se pasa al siguiente como si hubiera fallado su ejecución. Es importante señalar que no se trata de precondiciones, puesto que las condiciones de los hijos no garantizan la ejecución completa del comportamiento al que guardan, sino que acotan las condiciones bajo las que se puede ejecutar. El comportamiento de un nodo hijo puede fallar aún cuando su condición sea cierta.

- **Selectores con prioridad**

  Los selectores con prioridad guardan mucha semejanza con los selectores simples. De nuevo ejecutan cada uno de los hijos hasta que alguno de ellos tiene éxito. En este caso, consideraremos que cada uno de los hijos de un selector con prioridad está guardado por una condición. La diferencia principal con el anterior es que si, mientras se está ejecutando uno de los hijos, la guarda de otro hijo más prioritario que el actual se hace cierta, se interrumpe la ejecución del hijo actual y se comienza a ejecutar el de mayor prioridad. Por ejemplo, en la Figura 2.5(c), el comportamiento `Pelea conservador` tiene dos hijos: `Buscar salud`, guardado por la condición `salud <10 %`, y `Pelear`. Supongamos que el comportamiento empieza a ejecutarse en un momento en el que el NPC tiene un nivel de salud alto. La guarda fallaría y pasaría al siguiente comportamiento `Pelear`. Si el comportamiento fuera un selector simple, el NPC seguiría peleando hasta que terminara la pelea con éxito o con fallo. Al ser un selector con prioridad, si en algún momento se hace cierta la guarda del primer hijo, es decir, si su salud baja del 10 %, el

hijo más prioritario `Buscar salud` empezará a ejecutarse. Cuando este hijo falle, volverá a ejecutarse el comportamiento `Pelear`.

Al igual que en el caso anterior, en la literatura existen otras definiciones de este tipo de nodos en las que los hijos no tienen condiciones definidas explicitamente. Cada ciclo de ejecución se intentan ejecutar todos los hijos hasta el que se está ejecutando actualmente. Si alguno de ellos tiene éxito se cancela el hijo actual. En realidad, lo que se hace es cambiar la evaluación de las condiciones por la evaluación del estado de Éxito o Fallo de un comportamiento. Por término general se acaban definiendo comportamientos básicos que chequean una determinada condición, y se ponen como guardas, dentro de un nodo secuencia, en las ramas del árbol que lo necesitan, que suelen ser la mayoría. La consecuencia última de este cambio es que los árboles aumentan su complejidad considerablemente.

- **Paralelo**

  Este tipo de nodo ejecuta a todos sus hijos en el mismo ciclo de juego de manera que parece que se ejecutan concurrentemente. Habitualmente falla cuando lo hacen todos sus hijos y tiene éxito si alguno de ellos lo tiene.

- **Decoradores**

  El nombre de Decoradores se refiere a toda una familia de nodos de decisión que se caracterizan por tener un solo hijo y cuya función es modificar su comportamiento original o añadirle nuevas funcionalidades. Los decoradores más comunes son los modificadores, que cambian el valor devuelto por su hijo, invirtiéndola o forzando a que sea siempre un valor fijo, por ejemplo, y los filtros, que modifican su comportamiento. Entre los filtros usados más comúnmente se encuentra el Repetidor, que repite un comportamiento un número determinado de veces mientras tenga éxito, o el Temporizador, que impone una pausa entre dos ejecuciones de un comportamiento para evitar que se ejecute muy a menudo.

2.3. Herramientas de Autoría de IA en Videojuegos

2.3.1. Unreal Kismet

Unreal Kismet[1] forma parte del UDK (*Unreal Development Kit*). Kismet permite a usuarios sin conocimientos de programación crear scripts de manera visual para controlar las diferentes facetas de jugabilidad de un nivel

[1] Kismet: `http://www.unrealengine.com/features/kismet/`

del juego y, en concreto, permite crear la inteligencia artificial que controlará a los NPC. Los comportamientos creados con Kismet sólo pueden ser utilizados en juegos desarrollados usando el UDK.

Los comportamientos se especifican como *Secuencias*, que son colecciones de nodos, o *Sequence Objects* en la terminología de Kismet, que se conectan entre sí para formar construcciones complejas. Usando los nodos se pueden realizar acciones tales como cambiar la maya asignada a una entidad, cambiar su velocidad, modificar la cámara, cargar un nivel, etc.

Además, los *Sequence Objects* pueden tener un conjunto de puertos de entrada y de salida por los que pueden recibir y emitir señales de distinto tipo (booleanas, numéricas, referencias a objetos, etc.). Su funcionamiento es como el de una caja negra: cuando se activan determinadas entradas, el *Sequence Object* realizará su acción correspondiente y activará a su vez las salidas necesarias. Las salidas de unos pueden conectarse a las entradas de otros para formar así construcciones complejas.

Por ejemplo, el objeto *Timer* se utiliza para contabilizar el tiempo entre dos eventos. En este objeto encontramos dos entradas, Start y Stop, y dos salidas, Time y Out. Cuando el objeto recibe una señal de activación a través de Start empieza a contabilizar el tiempo transcurrido, actualizando el valor de la salida Time. Cuando recibe una señal a través de Stop deja de contabilizarlo. La señal de salida Out simplemente sirve para retransmitir las señales de activación a los siguientes objetos de la secuencia.

Podemos distinguir cuatro categorías de *Sequence Objects*:

- Acciones (Actions): son las acciones que puede realizar cada una de las entidades del juego.
- Eventos (Events): estos objetos informan cuando se produce un determinado suceso en el juego. Todos los eventos tienen una salida Out, que se activa al producirse el suceso asociado. Normalmente, la ejecución de una secuencia comienza con la activación de un evento.
- Condiciones (Conditions): las condiciones afectan al control de flujo de la *Secuencia*.
- Variables: almacenan información que puede ser usada posteriormente por otros objetos.

Un punto débil de Kismet es la escalabilidad. El editor no permite que los diseñadores asignen *Secuencias* a entidades específicas, sino que todas las *Secuencias* correspondientes a un nivel tienen que estar contenidas en el mismo documento. De esta manera podemos llegar a *Secuencias* prácticamente imposibles de seguir y mucho menos de mantener. Un buen ejemplo de ello son las mostradas en la Figura 2.6.

En cuanto a la modularidad, permite organizar las secuencias de manera jerárquica usando "subsecuencias" que actúan a modo de caja negra. De esta

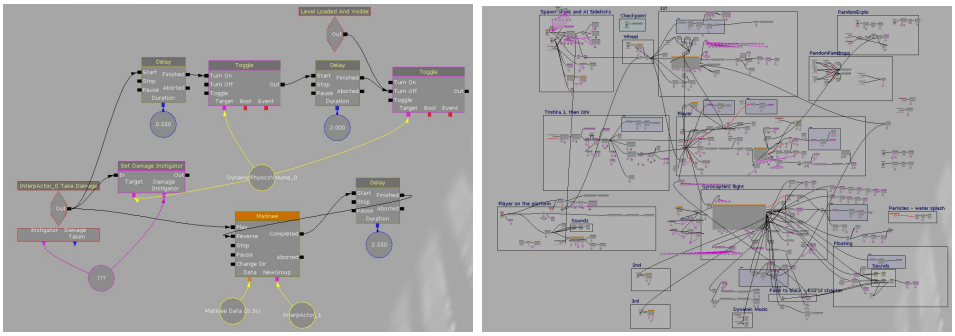


Figura 2.6: Diferentes *Secuencias* de Kismet que muestran el grado de complejidad que pueden alcanzar

manera, a una secuencia de alto nivel podemos añadir otras subsecuencias que funcionan de la misma forma que las Acciones, pero cuyo contenido ha sido diseñado por el usuario como una secuencia más.

Desde el punto de vista de la gestión de colecciones de comportamientos, Kismet no ofrece ninguna asistencia al usuario. El conjunto de secuencias se muestra en la interfaz de usuario como un árbol que lista todas las secuencias existentes junto con sus subsecuencias. El único tipo de búsqueda permitido es buscar un *Sequence Object* por nombre dentro de una secuencia.

2.3.2. Flow Graph Editor

Flow Graph Editor[2] es una herramienta que forma parte del Sandbox Editor del CryENGINE 3 SDK. Al igual que Kismet es una herramienta visual de scripting que permite construir comportamientos para las entidades de los niveles que han sido creados con el Sandbox Editor.

La filosofía del Flow Graph Editor es muy similar a la de Kismet. En este caso, los comportamientos se representan en forma de *Flow Graphs*, que están compuestos de nodos y aristas. Los nodos representan componentes (acciones) y entidades, y tienen puertos de entrada y salida que pueden conectarse entre sí mediante las aristas. Además, existe un tipo especial de componentes (los llamados *módulos*) que se puede usar para invocar la ejecución de otro grafo. Utilizando los módulos podemos crear una jerarquía de Flow Graphs.

En la Figura 2.7, junto con la interfaz de usuario del Flow Graph Editor, se muestra un ejemplo de Flow Graph. Este grafo de ejemplo se activa cuando el jugador entra en un *trigger*. En ese momento, se comprueba el nivel de salud del jugador. Si está por debajo de 50 unidades hace aparecer dos enemigos, mientras que si está por encima hace aparecer tres.

La diferencia más importante entre ambos editores es que en este caso

[2]Flow Graph Editor: `http://goo.gl/Y6a7i`

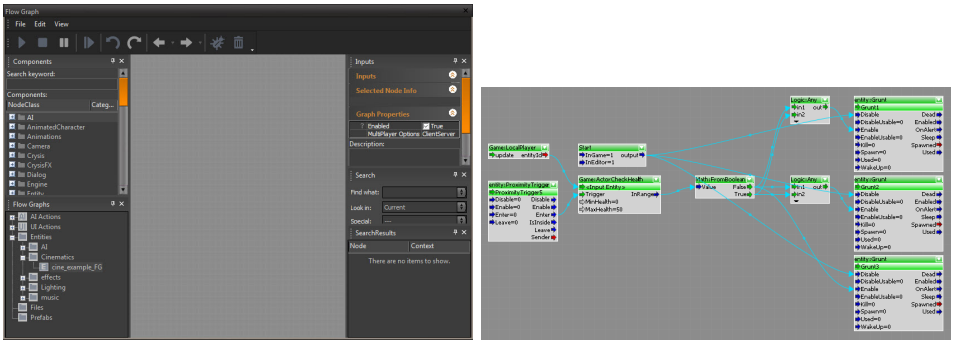


Figura 2.7: *Flow Graph Editor*

cada entidad puede tener su propio Flow Graph. De esta manera se mejora la escalabilidad cuando el número de entidades aumenta. Además, desde el Flow Graph de una entidad se puede hacer referencia a las restantes entidades del nivel, por lo que es posible coordinar el comportamiento de varias entidades. Los Flow Graphs pueden exportarse a ficheros XML y reutilizarse, de esta manera, en distintas entidades o niveles del juego.

Flow Graph Editor permite crear colecciones de grafos y organizarlas en grupos, dentro de un nivel. Además incluye una funcionalidad de búsqueda, pero sólo permite buscar nombres de nodos o valores en el grafo o el nivel actual.

2.3.3. Unity 3D

Unity[3] es una herramienta de autoría para crear juegos 3D para diferentes plataformas. La herramienta está compuesta por un entorno de edición para crear y diseñar los juegos y un motor para ejecutarlos. La versión actual de Unity (3.5.2) no incorpora ninguna herramienta visual que ayude a los diseñadores en la tarea de creación comportamientos inteligentes para los NPC. Es necesario programarlos en alguno de los lenguajes de script que soporta el editor. No obstante, existen plugins desarrollados por terceros que pueden ser utilizados para esta tarea. A continuación revisaremos algunos de ellos.

Behave[4] es un plugin de Unity que permite a los usuarios diseñar gráficamente los BT para controlar las entidades de un juego. El uso de Behave es bastante intuitivo y sólo son necesarios ciertos conocimientos generales sobre los BT. Aunque el proceso de edición de comportamientos es visual, es necesario utilizar *scripts* para asignar los BT a las entidades del juego y también para programar los comportamientos básicos que se ejecutarán en las hojas de los árboles.

[3] Unity: `http://unity3d.com/`
[4] Behave: `http://eej.dk/angryant/behave/`

En Behave todos los BT creados por el usuario se añaden a una colección. Las colecciones se usan para organizar los BT y para acceder a los árboles de los *scripts*. Behave permite reutilizar comportamientos creados previamente añadiendo referencias a ellos como nodos dentro de otro BT, pero para encontrar un comportamiento los usuarios tienen que revisar de manera manual las colecciones, ya que el editor no incluye ninguna herramienta de búsqueda.

RAIN{one}[5] y **Playmaker**[6] son dos plugins también para Unity. Ambos son comerciales y ambos permiten a los usuarios diseñar gráficamente comportamientos, usando BT en el caso de RAIN{one} y FSM en el de Playmaker. Para definir las acciones básicas, los usuarios tienen la opción de definir scripts adecuados a las acciones que necesiten, o bien, de utilizar un conjunto de acciones básicas predefinidas. De esta manera, usuarios sin conocimientos de programación pueden crear comportamientos simples.

Ninguna de las dos herramientas ofrece la posibilidad de crear comportamientos jerárquicos, a pesar de que ambas representaciones lo permiten. Tampoco dan la opción de realizar búsquedas de comportamientos.

2.3.4. BehaviorShop

BehaviorShop (Alexander et al., 2010; Heckel et al., 2009, 2010) es un editor de comportamientos basado en la arquitectura de subsunción propuesta en (Brooks, 1986).

Los comportamientos se definen como un conjunto de capas, compuestas por condiciones y acciones que toman la forma de reglas condicionales (`si <condición>entonces <acción>`). Cada capa tiene asociada una prioridad, de manera que las capas de mayor prioridad pueden inhibir el comportamiento de aquellas con una prioridad más baja.

En BehaviorShop, los comportamientos pueden organizarse de manera jerárquica: una única capa puede estar compuesta por dos o más capas subordinadas. Aún así, el diseño del editor no permite administrar colecciones de comportamientos.

2.3.5. En Conclusión

Como se puede ver, existen aproximaciones muy diferentes a la edición de comportamientos para videojuegos. La mayoría de ellas intenta facilitar el diseño de los comportamientos utilizando diferentes técnicas de representación (FSM, BT, reglas...). Por otro lado, el uso de técnicas para la reutilización de comportamientos editados con anterioridad no está muy extendida. La mayoría de los editores revisados no proveen facilidades para la gestión de colecciones de comportamientos y, aquellos que lo hacen, únicamente permiten añadir o eliminar comportamientos a la colección. En aquellos casos

[5] RAIN{one}: `http://rivaltheory.com/product`
[6] Playmaker: `http://hutonggames.com/index.html`

en los que se permite al usuario realizar búsquedas se limitan, en general, a permitir búsquedas textuales usando el nombre del comportamiento.

A nuestro entender, la edición de comportamientos se puede beneficiar en gran medida de la reutilización, pero para gestionar grandes colecciones de comportamientos, es necesario que los usuarios dispongan de herramientas para realizar búsquedas más complejas, de acuerdo, por ejemplo, a la funcionalidad o la estructura deseada en el comportamiento a recuperar.

2.4. Razonamiento Basado en Casos

El CBR (*Case Based Reasoning*, Razonamiento Basado en Casos) es un paradigma de resolución de problemas utilizado para el desarrollo de sistemas basados en el conocimiento. La idea fundamental que se emplea para solucionar un problema consiste en reutilizar soluciones a problemas parecidos ya resueltos en el pasado, adaptándolas a las condiciones del nuevo problema. Para ello se necesita una colección de experiencias previas, que reciben el nombre de casos y se almacenan en una base de casos. En términos generales, un caso se compone de la descripción del problema y la solución aplicada para resolverlo (Aamodt y Plaza, 1994).

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

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

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

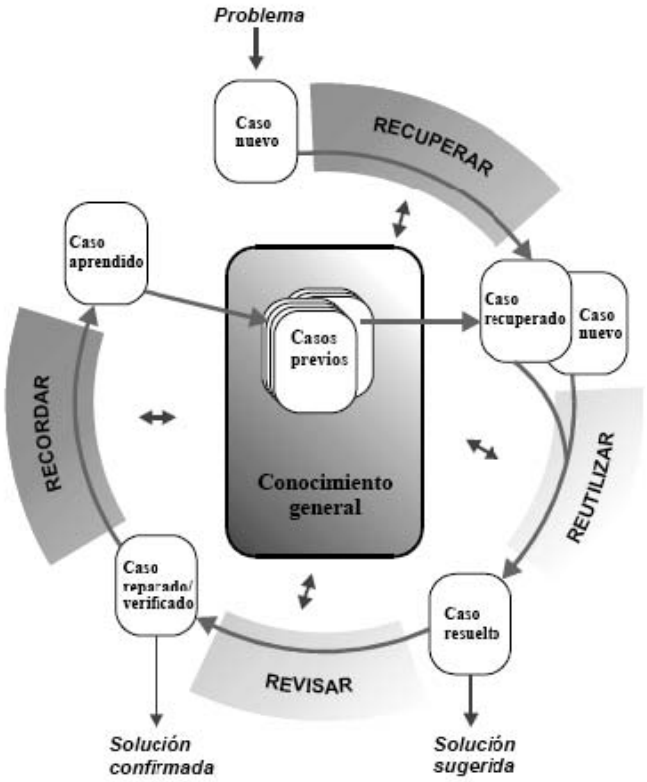


Figura 2.8: El ciclo CBR

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

2.4.1. El Ciclo CBR

De forma general, un problema se puede resolver mediante CBR aplicando el ciclo que se muestra en la Figura 2.8. En él se observa que, dado un problema en forma de nuevo caso (consulta) se procede a recuperar casos ya resueltos con anterioridad para elegir el mejor de ellos. A continuación se revisa la solución para comprobar que esta es correcta y, opcionalmente, se aprende el nuevo caso, junto con su solución, añadiéndolo a la base de casos.

2.4.1.1. Representación de los casos

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

para que la reutilización se realice de forma eficiente.

Por regla general, un caso está compuesto por:

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

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

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

- Representación plana: en este caso, todos los casos se representan utilizando los mismos atributos.
- Representación estructurada: al utilizar esta representación, distintas entidades del dominio pueden tener distintos atributos.

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

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

- **Árboles de decisión** A partir del conjunto de casos y fijando una serie de atributos, que actúan como índice, se puede obtener un árbol de decisión mediante el algoritmo ID3 o alguna de sus variantes, de manera que se pueda recuperar en un número mínimo de pasos los casos más relevantes para una consulta dada.
- **Árboles k-d** Son estructuras de datos que provienen del campo de la informática gráfica. Mediante ellos se pueden obtener los k puntos más próximos a uno dado. Los árboles k- d se pueden obtener automáticamente aplicando un algoritmo, que divide el espacio en volúmenes que contienen un número homogéneo de puntos. Aplicados a la recuperación en CBR, se sustituyen puntos por casos, que en lugar de ocupar

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

Figura 2.9: Funciones de similitud local

un lugar en el espacio tridimensional, ocupan un lugar en el espacio de atributos escogidos como índice. El proceso de recuperación consiste en realizar un recorrido en profundidad del árbol, buscando los k vecinos más próximos a la consulta dada.

2.4.1.2. Recuperación

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

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

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

En general se suele distinguir entre medidas de similitud local y global. Las primeras se utilizan para obtener la similitud entre atributos pertenecientes a diferentes casos, mientras que las segundas combinan los resultados de similitud local de todos los atributos para obtener un solo valor que mida la similitud entre los casos. A continuación se señalan algunas de las aproximaciones más comunes para ambas medidas:

- **Similitud local** En este caso hay que considerar qué tipo de atributos

se están comparando. Atendiendo a este criterio se puede distinguir entre atributos numéricos, símbolos y estructuras más complejas. Para los dos primeros existen algunas funciones estándar que se detallan en la Figura 2.9, distinguiendo además si cada atributo tiene un sólo valor (univaluado) o puede tener varios (multivaluado). Donde O es el conjunto de valores posibles, long (l) es el tamaño del intervalo l y ac y bc son los puntos centrales de los intervalos a y b.

En el caso de los atributos simbólicos, si existe una ordenación entre ellos y se puede calcular, tanto su posición en el orden como el número total de valores posibles, se les pueden aplicar las mismas funciones que a los atributos numéricos. Otras estrategias para el cálculo de la similitud contemplan el uso de tablas de similitudes, en las que se refleja explícitamente el valor de similitud entre cada par de valores de un atributo, o de ontologías, en cuyo caso se tiene en cuenta la distancia entre los individuos dentro de la ontología.

- **Similitud global** La similitud global resume los valores de similitud local de cada uno de los atributos del caso en uno sólo, que representa cuál es la similitud entre el caso y la consulta. Para ello puede emplearse diferentes operadores, siendo el más común de ellos la media aritmética.

2.4.1.3. Adaptación

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

- Adaptación transformacional: consiste en reutilizar la solución al caso recuperado aplicando determinados operadores de transformación sobre ella. No trata cómo se resuelve el problema, sino la equivalencia de soluciones. Requiere un fuerte modelo de dominio, los operadores de transformación, y un mecanismo de control para gestionar su aplicación.

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

plan de solución recuperado, pero en el contexto del caso introducido como consulta.

2.4.1.4. Revisión

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

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

2.4.1.5. Aprendizaje

Uno de los puntos fuertes del CBR es el aprendizaje, ya que, a un nivel elemental, es muy fácil de llevar a cabo. Se basa, tan sólo, en añadir los nuevos casos consultados a la base de casos. La finalidad del aprendizaje consiste en mejorar el rendimiento del sistema. Existen dos factores principales que miden el rendimiento: la competencia, que es el rango de problemas que pueden resolverse satisfactoriamente, y la eficiencia, el coste computacional de resolver cada problema. Atendiendo únicamente a la competencia, esta mejorará según crezca la base de casos dado que:

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

Los datos que se suelen conservar de un caso son:

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

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

el uso de taxonomías, o el aprendizaje de conocimiento de adaptación, que busca mejorar el proceso de adaptación.

Según lo visto hasta ahora, parece que lo mejor es tener una base de casos lo más grande posible, para mejorar así la competencia. Esto no siempre es así. Si la base de casos crece mucho, las búsquedas en ella son cada vez más lentas, es decir, aparece una degradación de la eficiencia. Es lo que se conoce como el problema de la utilidad (Smyth y Cunningham, 1996).

Para resolverlo hay que buscar un compromiso entre la eficiencia y el tamaño de la base de casos, especialmente en sistemas que necesitan bases de casos de gran tamaño. Por ello, es conveniente evitar incluir casos que no aporten información nueva al sistema. Existen distintas políticas de mantenimiento cuyo objetivo es refinar la base de casos para reducir el problema de la utilidad sin disminuir la competencia del sistema.

Capítulo 3

Objetivos y Planteamiento del Trabajo

3.1. Hacia un Modelo de Diseño de Comportamientos Basado en la Reutilización

En el Capítulo 1 se planteaba como objetivo general de la presente Tesis Doctoral facilitar a los diseñadores la tarea de creación de comportamientos para los NPC de un videojuego mediante la reutilización de comportamientos y fragmentos de comportamientos creados previamente. Para lograr este objetivo se ha desarrollado un modelo de creación de comportamientos basado en la reutilización, descrito en la siguiente sección.

Con la finalidad de llegar al modelo propuesto, este fue abordado desde los siguientes frentes:

- Propuesta de diferentes mecanismos de representación de comportamientos y de medidas de similitud asociadas a ellos que permitan la recuperación eficiente de comportamientos de la biblioteca.

  Se ha realizado una revisión de diferentes técnicas para la representación de comportamientos existentes en la literatura con el fin de encontrar las más adecuadas para construir un modelo basado en la reutilización. Dentro de las técnicas revisadas, las más adecuadas para nuestro modelo son las HFSM y los BT por diversas razones:

  - Ambas técnicas son intuitivas y simples de implementar, diseñar y visualizar. De un solo vistazo podemos capturar la idea general del comportamiento.
  - La teoría subyacente está formalizada en el caso de las HFSM. En el caso de los BT no es así, pero existe una formalización *de facto* aceptada en la industria.

- Ambas técnicas han sido ampliamente probadas para diseñar la IA de numerosos juegos, con buenos resultados. Además, esto nos asegura que estas técnicas son familiares para la mayoría de diseñadores de videojuegos.
- Tienen una representación visual que simplifica, como hemos dicho, el diseño de comportamientos a los usuarios que no tienen conocimientos acerca de cómo programarlos. Además, el paso del diseño a la implementación se puede automatizar.
- Tanto las HFSM como los BT son modulares, lo que permite reutilizar, no sólo los comportamientos completos, sino también fragmentos de comportamientos con una funcionalidad específica.

Junto con estas técnicas, se hace necesario para completar el modelo especificar cómo se realiza la recuperación, definiendo las funciones de similitud correspondientes.

En nuestro modelo proponemos dos tipos de recuperación:

- Recuperación basada en funcionalidad. En este caso, la búsqueda se realiza especificando la funcionalidad esperada del comportamiento que se desea recuperar. Este tipo de recuperación utiliza funciones de similitud basadas en atributos, que se caracterizan porque a cada comportamiento de la biblioteca se le asocia un conjunto de pares `(atributo, valor)`, cada uno de los cuales describe una característica concreta del comportamiento. Para recuperar un elemento de la biblioteca se crea una consulta, también formada por pares `(atributo, valor)`, y se recupera el elemento de la biblioteca cuyo conjunto de pares se parezca más al de la consulta.
- Recuperación basada en bocetos. Para especificar el comportamiento que se desea recuperar se utiliza un boceto del comportamiento final, es decir, una representación aproximada del mismo pero que difiera en algunas de sus partes. Como hemos visto anteriormente, tanto las HFSM como los BT pueden representarse como grafos. En este caso, lo que se compara es la estructura del comportamiento, es decir, cómo están interconectados entre sí los nodos y las aristas que los forman, para encontrar dentro de la biblioteca el que guarde más semejanza con la consulta. Utilizamos, por tanto, funciones de similitud estructural para grafos.

- Propuesta de técnicas para la construcción de comportamientos complejos a partir de otros creados previamente.

  Mediante las técnicas de recuperación referidas anteriormente es posible recuperar comportamientos de la biblioteca de acuerdo a una

consulta. Estos comportamientos pueden ser comportamientos completos, utilizables por el diseñador para asignarlos a un NPC del juego directamente o tras ser modificados, o fragmentos de comportamientos, pequeños bloques de comportamiento que cumplen una funcionalidad específica. Combinando varios de estos bloques el diseñador puede construir comportamientos más complejos.

Como parte del desarrollo del modelo hemos estudiado posibles técnicas para sistematizar el proceso de creación de comportamientos complejos a partir de otros más simples. El resultado ha sido la incorporación al modelo de los *nodos consulta*. Un nodo consulta es un nodo de una HFSM o BT que contiene en su interior una consulta a la biblioteca. En esta consulta se especifica el comportamiento que deberá ejecutarse cuando, durante el juego, se intente ejecutar este nodo. De esta manera, el diseñador no necesita conocer el conjunto completo de bloques que realizan una función y los programadores pueden añadir nuevos bloques incluso después de que el comportamiento que incluye el nodo consulta esté terminado, ya que la selección del comportamiento específico se aplaza hasta el momento de la ejecución.

- Diseño de herramientas que soporten el modelo propuesto.

  Paralelamente a la construcción del modelo se ha implementado un conjunto de herramientas que sirven de apoyo al mismo. De entre ellas, la más importante es *eCo*, un editor visual de comportamientos que lleva a la práctica lo propuesto en el modelo. El objetivo principal del editor es comprobar de manera práctica la viabilidad de las ideas que se plantean en el modelo. Al mismo tiempo, ofrece a los usuarios interesados en el modelo una implementación que les permite comenzar a utilizarlo rápidamente. Además, el editor nos ha servido como banco de pruebas de cara a la evaluación de las diferentes técnicas utilizadas. Por un lado, hemos podido evaluar las técnicas en acción y por otro, ha servido para generar datos tales como comportamientos y trazas de uso que han sido utilizadas posteriormente en diferentes experimentos.

- Demostración experimental de la validez de las técnicas y herramientas propuestas.

  Con el objetivo de cuantificar las mejoras que aporta el modelo se han realizado varios experimentos. A través de ellos, se busca demostrar que la utilización del modelo de desarrollo de comportamientos propuesto ahorra tiempo a los diseñadores, reduciendo el número de pasos que tienen que dar para conseguir llegar a un comportamiento completo para un NPC.

3.2. Descripción del Modelo de Diseño

Dentro del proceso de desarrollo de la IA de un videojuego la reutilización puede resultar de gran utilidad en diversos escenarios. A continuación enumeramos a modo de ejemplo algunos escenarios en los que la reutilización puede resultar útil:

- Como parte de la creación de un comportamiento complejo, un diseñador necesita crear un subcomportamiento simple que realice una tarea concreta. El diseñador puede buscar en la biblioteca de comportamientos creados previamente un comportamiento simple que se adapte a sus necesidades.

- El diseñador necesita crear un nuevo comportamiento para una entidad. Para no empezar desde cero, busca en la biblioteca de comportamientos algún comportamiento con una funcionalidad parecida al que quiere crear.

- Mientras está diseñando un comportamiento, el diseñador desea recibir sugerencias acerca de comportamientos similares que hayan sido creados con anterioridad. Estas sugerencias deben basarse en el comportamiento que está creando en ese momento.

- Desde hace alrededor de 20 años existe una práctica común entre los usuarios de determinados videojuegos que recibe el nombre de "modding" (Kücklich, 2005) y consiste en desarrollar modificaciones sobre un juego original, dando lugar a nuevos escenarios, personajes, mecánicas, IA, etc. En los últimos años, han surgido varios juegos en los que la creación de contenido para el juego y la posibilidad de compartirlo con otros jugadores se ha convertido en una mecánicas más. Es lo que se ha llamado "Gaming 2.0" , siendo quizá su ejemplo más conocido el videojuego "Little Big Planet"[1]. En este escenario, los diseñadores son los propios usuarios, quienes estarían interesados en realizar búsquedas de comportamientos entre el contenido creado previamente por otros usuarios para añadirlo a su propio nivel o para jugar a un nivel que contenga ese contenido concreto.

- En lugar de definir un subcomportamiento explícito para una entidad, el diseñador desea que este se decida durante el juego, según determinados factores de la partida. En este caso, es necesario un sistema que permita definir el tipo de comportamiento deseado durante el diseño y que sea capaz de encontrar un comportamiento adecuado dentro del repositorio durante la partida.

[1] Little Big Planet (Media Molecule, 2008): `http://www.littlebigplanet.com/`

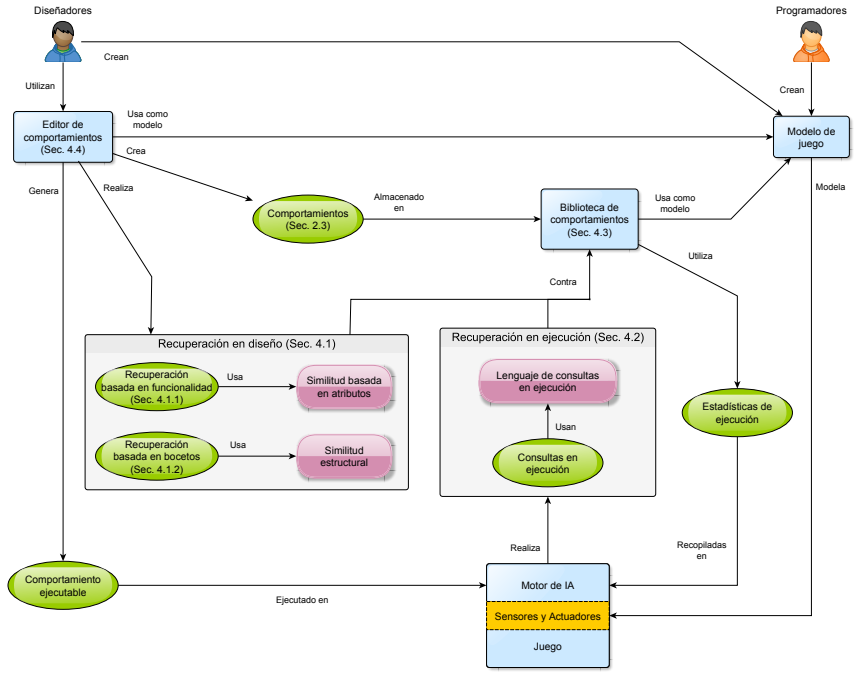


Figura 3.1: Proceso de diseño de comportamientos basado en la reutilización

El principal problema es que, como hemos visto en los ejemplos anteriores, la reutilización supone, en líneas generales, realizar una búsqueda del comportamiento más adecuado a las necesidades del diseñador. Para ello, es necesario recorrer la biblioteca de comportamientos creados y analizar cada uno de ellos para comprobar si es el más adecuado. Cuando el número de comportamientos es grande o estos son complejos, realizar estas tareas puede ser demasiado trabajoso si no contamos con las herramientas adecuadas.

Para facilitar el proceso de diseño basado en la reutilización proponemos utilizar una metodología, siguiendo el esquema de la Figura 3.1, que toma ideas del CBR.

Antes de construir los comportamientos, tanto los diseñadores como los programadores deben ponerse de acuerdo en el conjunto de sensores y actuadores que van a utilizar. Por un lado, los diseñadores deben indicar cuál es el conjunto que necesitarán para construir los comportamientos para el juego. Por otro, los programadores añadirán los sensores y actuadores propuestos al juego, construyendo la interfaz de comunicación entre la IA y el resto de sistemas del juego. De esta interacción surge lo que hemos llamado el *modelo de juego*, que modela las propiedades que son específicas del juego desde el punto de vista de la IA.

Utilizando los sensores y actuadores especificados en el *modelo de juego*, los diseñadores comienzan a construir los comportamientos. Un problema

clave en este punto es que los diseñadores, en muchos casos, no tienen los conocimientos técnicos necesarios para programar los comportamientos e integrarlos en el juego. Para solucionar este problema, proponemos la utilización de un editor visual de comportamientos, que permita a los diseñadores crear comportamientos ejecutables de manera sencilla sin necesidad de programarlos.

El editor no sólo debe permitir completar el diseño de los comportamientos, sino que también tiene que ayudar a los diseñadores a pasar del comportamiento diseñado a un *comportamiento ejecutable* sin que sea necesario programarlo. En este caso estamos utilizando el término "ejecutable" en un sentido amplio. No nos referimos a que el editor deba generar una versión compilada del comportamiento diseñado, sino que el comportamiento debe poder ser ejecutado en el juego directamente. En este sentido, un comportamiento ejecutable puede ser, por ejemplo, un fichero de código fuente que será compilado junto con el juego, un script que será ejecutado por el juego o un fichero de datos que el juego interpretará.

Los comportamientos terminados se pueden ejecutar en el juego, probándolos en varias situaciones. De esta manera, se recopilan estadísticas de su funcionamiento que servirán para decidir en qué situaciones es más adecuado reutilizarlos, como veremos más adelante.

En la Sección 4.4 se describe *eCo*, una herramienta visual de creación de comportamientos basada en la reutilización que hemos diseñado y que incorpora las técnicas de reutilización que describimos a continuación.

Cada uno de los comportamientos creados por los diseñadores se almacena en la *biblioteca de comportamientos*. De cara a la reutilización, es conveniente que la biblioteca contenga, no sólo los comportamientos completos, sino también fragmentos de comportamientos con una funcionalidad específica de manera que puedan ser reutilizados posteriormente como parte de nuevos comportamientos. Por ejemplo, si estamos construyendo comportamientos para un juego de fútbol, *Delantero agresivo* podría ser un comportamiento completo almacenado en la biblioteca, mientras que *Ir hacia la pelota* sería un fragmento de comportamiento reutilizable.

Cuando en la biblioteca exista una cantidad de comportamientos adecuada, los diseñadores podrán realizar consultas para, en lugar de diseñar un comportamiento desde cero, poder utilizar como base un comportamiento creado anteriormente.

Dentro del ciclo CBR la biblioteca cumple la función de *base de casos* y los comportamientos almacenados en ella son los *casos* que podemos reutilizar. Cada caso está compuesto por la descripción del problema, que se corresponde en nuestro modelo con la descripción del comportamiento, y la solución al caso, es decir, aquello que el usuario busca recuperar al realizar una consulta, que será el comportamiento en sí mismo.

Para construir una consulta el diseñador debe crear una nueva descripción del problema que desea resolver, es decir, del comportamiento que quiere recuperar. Utilizando una función de similitud adecuada se compara la consulta con las descripciones de cada uno de los comportamientos de la biblioteca. El método para seleccionar los comportamientos recuperados por la consulta es el de los kNN (*k Nearest Neighbours*, k vecinos más cercanos), donde se recuperan los k comportamientos más parecidos. El número k de comportamientos recuperados lo selecciona el usuario.

En nuestro modelo permitimos dos tipos de recuperación durante el diseño: recuperación basada en funcionalidad y recuperación basada en bocetos. Cada una de ellas utiliza una descripción de los casos diferente y, por lo tanto, una función de similitud diferente.

En la recuperación basada en funcionalidad, cada caso de la biblioteca está descrito por un conjunto de pares `(atributo, valor)` que especifican la funcionalidad del comportamiento. Estos atributos dependen principalmente del juego, por lo que se especifican en el *modelo de juego*. Por ejemplo, en un juego de fútbol tendrían sentido atributos como *Defensor*, *Atacante* o *Agresividad*, mientras que en un *shooter* podrían ser adecuados la *Puntería* o también la *Agresividad*.

Para construir una consulta el diseñador asigna valores a los distintos atributos, creando una descripción del comportamiento que desea recuperar. A continuación se compara la consulta con las descripciones de los comportamientos de la biblioteca, utilizando una función de similitud que calcula en primer lugar la similitud local, es decir, la similitud entre los valores del mismo atributo en la consulta y en el caso. A continuación se calcula un valor de similitud global combinando los valores locales de cada atributo. La recuperación basada en funcionalidad se cubre en detalle en la Sección 4.1.1.

El principal problema al utilizar este tipo de consultas es la adquisición del conocimiento. Al añadir un comportamiento a la biblioteca, el diseñador tiene, además, que asignar valores a todos los atributos que forman su descripción. Esta es una tarea tediosa para el diseñador y sujeta, en muchos casos, a su subjetividad. A esto hay que sumarle que, dado un conjunto de atributos, no siempre es fácil asignar valores a todos ellos para describir un comportamiento, especialmente en el caso de los comportamientos más complejos.

Para solucionar este problema proponemos automatizar la asignación de valores a los atributos: en lugar de usar atributos arbitrarios, se trata de recopilar estadísticas de los comportamientos ejecutándolos en diferentes situaciones. Los valores de esas estadísticas son utilizados como atributos para caracterizar a los comportamientos de la biblioteca. Esta técnica se explica en mayor detalle en la Sección 6.3.

Otro escenario habitual dentro del ámbito de la reutilización es el siguiente: un diseñador que está en el proceso de crear un comportamiento quiere

encontrar comportamientos similares al que está construyendo. De esta manera puede utilizarlos como modelo o, si el comportamiento recuperado es igual que el que quiere construir, se evita tener que hacerlo desde cero o añadir comportamientos duplicados a la biblioteca. Esto requiere construir una consulta que represente el comportamiento actual.

Usando estadísticas es posible automatizar la asignación de valores a los atributos que describen los casos. Sin embargo, no podemos utilizar esta aproximación para crear consultas, porque es necesario tener un comportamiento terminado, que se pueda ejecutar en el juego, para recopilar los valores de los atributos. Para recopilar los atributos, además, hay que ejecutar los comportamientos en el juego, lo que haría que el tiempo requerido para construir la consulta usando esta técnica fuera demasiado largo.

Por lo tanto, el usuario tendría que construir la consulta asignando valores manualmente a los atributos propuestos, ya sean atributos arbitrarios o estadísticas observables. A este problema hay que añadir que, como hemos dicho anteriormente, no siempre es trivial describir un comportamiento dando valores a sus descriptores.

La recuperación basada en bocetos, descrita en la Sección 4.1.2, puede ayudarnos a resolver este problema. En el dominio de la recuperación de imágenes la recuperación basada en bocetos consiste en encontrar una imagen más o menos compleja utilizando como consulta una representación aproximada de ella (un boceto). Esta idea puede trasladarse al dominio de la recuperación de comportamientos, donde podemos considerar un boceto como una representación parcial de un comportamiento (por ejemplo, una FSM a la que le faltan algunas aristas o para la que el comportamiento en algunos de sus nodos no se ha definido). En la recuperación de comportamientos basada en bocetos buscamos en un repositorio los comportamientos que son similares al que el usuario está "dibujando" y hacemos sugerencias acerca de cómo puede completarlo. Para realizar la búsqueda se compara la consulta con las descripciones de los comportamientos candidatos, es decir, con la estructura de los comportamientos almacenados en la biblioteca.

Para comparar la estructura de los comportamientos necesitamos una función de similitud adecuada a la representación de los comportamientos en la biblioteca. En el modelo que proponemos estamos considerando que los diseñadores construyen los comportamientos usando un editor visual que les permita dibujar los comportamientos utilizando técnicas tales como HFSM o BT. Ambas técnicas coinciden en que la representación subyacente es la de un grafo. Por lo tanto podemos usar alguna de las funciones de similitud de grafos presentes en la literatura para comparar entre sí dos HFSM o BT.

Uno de los problemas de estas funciones de similitud estructural es que su complejidad computacional es muy alta. Para poder realizar las consultas hemos optado por utilizar una heurística que permita calcular la solución al problema en un tiempo razonable a costa de devolver una solución subópti-

ma. En (Flórez-Puga et al., 2012b) exponemos la función de similitud y la heurística propuesta. Como se demuestra en la Sección 6.3, los comportamientos recuperados en nuestro dominio de estudio usando esta heurística, no sólo tienen una estructura similar, sino que también su comportamiento es parecido al de la consulta.

Una característica destacable de la recuperación basada en bocetos es que no es necesaria la intervención del usuario para crear una consulta, ya que la consulta es el propio comportamiento que está editando. Esto permite automatizar todo el proceso de recuperación lanzando consultas automáticamente. Utilizando estas ideas, hemos incorporado al editor un sistema de sugerencias, cuyo funcionamiento detallaremos más adelante.

Como hemos visto, la reusabilidad y la modularidad son dos características importantes de los comportamientos en las que se basa la reutilización. Durante la fase de diseño y desarrollo del juego se crean distintos comportamientos independientes que pueden ser ensamblados para crear nuevos comportamientos más complejos. Cada comportamiento representa una abstracción que puede utilizarse como una "pieza" para crear un nuevo comportamiento.

Las herramientas de búsqueda que hemos descrito nos proporcionan una manera "estática" de reutilizar los comportamientos: las consultas se realizan sobre el contenido actual de la biblioteca y los subcomportamientos, una vez que son recuperados y añadidos a otro comportamiento, quedan almacenados en la biblioteca. Pero durante el desarrollo del juego, tanto los diseñadores como los programadores pueden añadir nuevos comportamientos o fragmentos reutilizables, que podrían ser adecuados para las búsquedas realizadas con anterioridad. Por lo tanto, para que el proceso de reutilización sea consistente y útil, el diseñador debería volver atrás y revisar los comportamientos existentes que puedan ser relevantes para el nuevo comportamiento añadido. Este proceso de chequeo de consistencia requiere un esfuerzo extra y en muchos casos no es llevado a cabo, lo que implica que los comportamientos añadidos a la biblioteca en fases posteriores de desarrollo no se tienen en cuenta para crear los comportamientos incluidos al principio.

Para solucionar este problema proponemos en nuestro modelo lo que hemos llamado *recuperación durante la ejecución*, que consiste en realizar consultas durante la ejecución del juego para encontrar el subcomportamiento más adecuado dentro de la biblioteca. De esta manera, el sistema siempre trabajará con la biblioteca actualizada y podrá recuperar el comportamiento más idóneo de acuerdo a una consulta usando toda la colección de comportamientos diseñados, pero a la vez evitando el coste extra de comprobar la consistencia para nuevos comportamientos.

Para ello, añadimos a los comportamientos un nuevo tipo de nodo, el *nodo consulta*. Cuando este nodo se ejecuta, lanza una consulta contra la biblioteca de comportamientos y ejecuta el comportamiento recuperado por

ella. Una ventaja de este tipo de consultas es que, al lanzarse durante la ejecución del juego, disponemos de información acerca del estado actual del juego en el momento de ejecutarlas. Así pues, podemos describir los comportamientos de la biblioteca, no sólo por su funcionalidad o sus objetivos, sino también usando atributos que describan en qué estado debe encontrarse el juego cuando pueden ser utilizados.

Esta idea se verá más clara con un ejemplo. Podemos tener en nuestra biblioteca dos subcomportamientos llamados *Tirar a puerta* y *Pasar a compañero desmarcado*. En la descripción del primero de ellos existe un atributo llamado *Heurística de tiro*. Este atributo tiene un valor entre 1 y 0 que indica si el jugador tiene una buena posición para tirar a portería. En el caso del comportamiento *Tirar a puerta* este atributo tendría que tener un valor alto, para indicar que este comportamiento será utilizado cuando el jugador tenga una buena posición de tiro. De la misma forma, el comportamiento *Pasar a compañero desmarcado* podría estar descrito por un atributo *Compañero desmarcado*, que también tendría un valor alto.

A continuación construimos un nuevo comportamiento para un delantero. En este comportamiento añadimos un *nodo consulta* que se ejecute cuando el jugador está cerca de la portería contraria. Cuando se ejecute el comportamiento, al llegar al *nodo consulta*, se obtienen del estado del juego los valores actual de *Heurística de tiro* y de *Compañero desmarcado* y se utilizan para realizar la recuperación. De esta manera, si el jugador que ejecuta el nodo consulta está en buena posición de tiro, se recuperará y ejecutará el comportamiento *Tirar a puerta*. Si es *Compañero desmarcado* el que tiene un valor más alto en el estado actual del juego, se ejecutará *Pasar a compañero desmarcado*.

La recuperación en ejecución aporta, además, una manera de construir comportamiento emergente y variable. Los comportamientos que utilizan nodos consulta no están prefijados, sino que pueden variar según el resultado de la consulta. En la Sección 4.2 se estudia el proceso de recuperación durante la ejecución, describiendo el lenguaje de consultas desarrollado para construir los nodos consulta.

Capítulo 4

Discusión de las Contribuciones de los Artículos

4.1. Reutilización de Comportamientos Durante el Diseño

4.1.1. Recuperación Basada en Funcionalidad

La manera más intuitiva de recuperar un comportamiento de una biblioteca con una funcionalidad específica consiste en asignar descriptores a cada comportamiento de la biblioteca. Cuando se añade un nuevo comportamiento a la biblioteca, el usuario asigna valores a estos descriptores, consiguiendo así una descripción de su funcionalidad. Para construir la consulta, se procede de la misma forma: el usuario debe asignar a cada descriptor un valor que describa la funcionalidad del comportamiento que desea recuperar.

Para realizar la búsqueda se compara la consulta con los descriptores de cada comportamiento de la biblioteca usando una función de similitud. Los comportamientos más similares son mostrados al usuario para que escoja el más adecuado.

Siguiendo estas ideas, en 6.2 (Flórez-Puga y Díaz-Agudo, 2007) se detalla cómo se lleva a cabo la recuperación basada en funcionalidad. Además de los descriptores de funcionalidad, en nuestra aproximación se añaden dos parámetros que permiten describir los comportamientos con mayor precisión. Por un lado, una descripción textual que permite al usuario especificar con un grano más fino determinados aspectos que considere relevantes sobre el comportamiento y que no estén cubiertos por los descriptores predefinidos. Por otro, se puede añadir a las consultas una lista de comportamientos subordinados. De esta manera, el usuario puede recuperar comportamientos que incluyan un determinado conjunto de subcomportamientos.

Las mismas ideas también se pueden aplicar a BTs, como se describe

en 6.1 (Flórez-Puga et al., 2011). En este caso, el conjunto de descriptores es diferente para ajustarlo de manera más adecuada al juego que se está modelando.

El principal problema de esta aproximación está en el cuello de botella que supone la adquisición de conocimiento. Al añadir un comportamiento a la biblioteca, el diseñador tiene, además, que asignar valores a todos los atributos que forman su descripción. Esta es una tarea tediosa para el diseñador y sujeta, en muchos casos, a su subjetividad. A esto hay que sumarle que, dado un conjunto de atributos, no siempre es fácil asignar valores a todos ellos para describir un comportamiento, especialmente en el caso de los comportamientos más complejos.

Para solucionar este problema proponemos automatizar la asignación de valores a los atributos: en lugar de usar atributos arbitrarios, proponemos recopilar estadísticas de los comportamientos ejecutándolos en diferentes situaciones. Los valores de esas estadísticas son utilizados como atributos para caracterizar a los comportamientos de la biblioteca. Esta técnica se explica en detalle en 6.3 (Flórez-Puga et al., 2012a).

4.1.2. Recuperación Basada en Bocetos

En 6.3 (Flórez-Puga et al., 2012a) se detalla una nueva aproximación a la recuperación de comportamientos, a la que hemos dado el nombre de *recuperación basada en bocetos*. En el dominio de la recuperación de imágenes, la recuperación basada en bocetos consiste en encontrar una imagen más o menos compleja utilizando como consulta una representación aproximada de ella (un boceto). Esta idea puede trasladarse al dominio de la recuperación de comportamientos, donde podemos considerar un boceto como una representación parcial de un comportamiento (por ejemplo, una FSM a la que le faltan algunas aristas o para la que el comportamiento en algunos de sus nodos no se ha definido). En la recuperación de comportamientos basada en bocetos buscamos en la biblioteca los comportamientos que son similares al que el usuario está "dibujando" y hacemos sugerencias acerca de cómo puede completarlo. Para realizar la búsqueda se compara la consulta con las descripciones de los comportamientos candidatos, es decir, con la estructura de los comportamientos almacenados en la biblioteca.

Para comparar la estructura de los comportamientos es necesaria una función de similitud adecuada a la representación de los comportamientos en la biblioteca. En el modelo propuesto en la Sección 3.2 estamos considerando que los diseñadores construyen los comportamientos usando un editor visual que les permita dibujar los comportamientos utilizando técnicas tales como HFSM o BT. Ambas técnicas coinciden en que la representación subyacente es la de un grafo. Por lo tanto podemos usar alguna de las funciones de similitud de grafos presentes en la literatura para comparar entre sí dos HFSM o BT. En concreto, hemos propuesto y evaluado el uso de la distancia

de edición para grafos (Bunke y Messmer, 1994).

Uno de los problemas de las funciones de similitud estructural es que su complejidad computacional es muy alta. Para poder realizar las consultas hemos optado por utilizar una heurística que permita calcular la solución al problema en un tiempo razonable a costa de devolver una solución subóptima.

La recuperación basada en bocetos nos plantea una pregunta importante: ¿dos grafos que son similares a nivel estructural representan comportamientos similares? De no ser así, la recuperación basada en bocetos pierde gran parte de su utilidad. En este artículo se demuestra mediante resultados experimentales que existe una correlación entre la similitud estructural y la similitud basada en atributos. Es decir, que si dos comportamientos tienen una estructura similar, existe también una similitud en su funcionalidad.

Otra pregunta de investigación fundamental relacionada con la recuperación basada en bocetos es si, al utilizarla, se obtiene una mejora en cuanto al tiempo de edición y si esta mejora es significativa. En 6.7 (Flórez-Puga et al., 2013) se demuestra de manera experimental que esto es así, con tasas que rondan el 50 % de ahorro en el número de pasos requeridos para construir un comportamiento.

En 6.5 (Flórez Puga et al., 2008) se hace una revisión de algunas de las funciones de similitud estructural presentes en la literatura y de cómo se pueden adaptar a las HFSM. Por otro lado, en 6.4 (Flórez-Puga et al., 2010) se comparan los resultados de estas funciones con los de la función de similitud basada en atributos. También se demuestra experimentalmente que, en el dominio de estudio, no existen diferencias sustanciales entre los resultados de las funciones estudiadas. Tanto la función de similitud empleada en esta publicación como la heurística se explican con más detalle en (Flórez-Puga et al., 2012b).

4.2. Reutilización de Comportamientos Durante la Ejecución

Si en las secciones anteriores hemos hablado de cómo recuperar comportamientos durante el diseño para asistir al diseñador de IA, en 6.6 (Flórez-Puga et al., 2009) se aborda el tema de la recuperación de comportamientos para ser reutilizados durante la ejecución. Esta técnica se ejemplifica en este caso con BT, pero puede aplicarse sin realizar grandes cambios a HFSM.

En esta publicación se propone extender los BT con un nuevo tipo de nodo: el nodo consulta. Cuando el flujo de ejecución llega a un nodo consulta se ejecuta la consulta que contiene, recuperando de la biblioteca un comportamiento. El comportamiento recuperado sustituye al nodo consulta, pasando a ser ejecutado en su lugar.

Al ser lanzada en ejecución, la consulta puede utilizar parámetros relativos al estado del juego o del jugador en el momento de ejecutarse. Los comportamientos de la biblioteca están descritos mediante un conjunto de restricciones sobre los parámetros que conforman el estado del juego. Estas restricciones indican en qué circunstancias del juego el comportamiento descrito es aplicable. En el momento de ejecutarse la consulta, se comparan las restricciones con el estado del juego en ese instante. De esta manera, el diseñador puede construir comportamientos variables, pues el comportamiento recuperado será diferente en función del estado del juego.

Una ventaja de esta aproximación es que el número de comportamientos en la biblioteca puede crecer durante todo el proceso de desarrollo y, aún así, los diseñadores pueden asegurarse de que los nuevos comportamientos serán utilizados en comportamientos complejos más antiguos. Sin nodos consulta, al añadir un nuevo comportamiento a la biblioteca, es necesario revisar todos los comportamientos contenidos en ella para comprobar si alguno se podría beneficiar de él, algo que sólo es aceptable cuando el tamaño de la biblioteca es pequeño. Con los nodos consulta, esta comprobación se hace automáticamente: cuando se ejecuta la consulta, si el nuevo comportamiento es adecuado se ejecutará en el nodo, mejorando así la escalabilidad de la biblioteca de comportamientos.

4.3. Una Biblioteca de Comportamientos Reutilizables

Para poder llevar a cabo la recuperación de comportamientos es necesario contar con una biblioteca en la que los comportamientos diseñados previamente se encuentren indexados. En 6.1 (Flórez-Puga et al., 2011) se describen las características que debe tener esta biblioteca.

Por un lado se describe el modelo del dominio de la biblioteca, especificando las restricciones que se imponen sobre los comportamientos en ella almacenados. Así pues, el modelo de dominio contempla la utilización de comportamientos parametrizables, con un contexto en el que se pueden almacenar, durante la ejecución, datos relativos al mundo de juego, al jugador, al NPC que ejecuta el comportamiento, etc.

La biblioteca almacena, no sólo los comportamientos, sino también un conjunto de metadatos asociados a cada uno de ellos, que serán utilizados durante la recuperación. Entre los metadatos se encuentran, por ejemplo, los objetivos del comportamiento, los parámetros que requiere y un conjunto de descriptores que especifican características acerca del comportamiento. Para poder razonar con estos datos, se propone la utilización de dos ontologías: una con conocimiento acerca de los comportamientos y otra sobre las diferentes entidades del juego. Aunque estas ontologías son propias de cada juego y tienen que ser construidas *ad-hoc*, es posible reutilizarlas parcialmente,

especialmente en juegos del mismo género.

Por otro lado, se define un lenguaje de consultas que permite construir consultas para ser ejecutadas tanto durante el diseño como durante la ejecución de los comportamientos. El lenguaje de consultas utiliza las ontologías mencionadas anteriormente para formular restricciones.

Por último, se muestra cómo se integra el modelo propuesto dentro de una arquitectura de juego orientada a componentes.

4.4. El Editor de Comportamientos *eCo*

Una parte fundamental del trabajo que se presenta en esta Tesis es una herramienta de autoría de comportamientos llamada *eCo*[1], desarrollada siguiendo las directrices del modelo propuesto. *eCo* es un editor visual altamente configurable, que permite la creación de comportamientos para prácticamente cualquier juego o entorno de simulación. El editor ha servido para un doble propósito: por un lado, ha sido un banco de pruebas para comprobar la efectividad de los algoritmos y las técnicas presentadas en este trabajo y, por otro, ha facilitado en gran medida el proceso de adquisición del conocimiento, ya que los casos con los que trabajamos son comportamientos que han sido creados utilizando el editor.

eCo es una pieza fundamental dentro del modelo de diseño de comportamientos basado en la reutilización ya que, no sólo se encarga de la edición visual de comportamientos; también incluye las herramientas necesarias para gestionar las bibliotecas de comportamientos creadas y para realizar en ellas las búsquedas descritas anteriormente.

Este editor ha evolucionado junto con el trabajo de investigación. En 6.2 (Flórez-Puga y Díaz-Agudo, 2007) se describe una versión preliminar del editor y se enuncian los tres objetivos principales que le han servido como base:

- Facilidad de uso: al ser un editor orientado a los diseñadores de comportamientos, el usuario no necesita tener conocimientos técnicos sobre la creación de los comportamientos para el juego. Para lograrlo, el editor utiliza formatos intermedios de representación de los comportamientos en lenguajes visuales, tales como las HFSM o los BT. Además, el editor es capaz de generar el código correspondiente al comportamiento diseñado. De esta manera no es necesaria la intervención de los programadores en ningún punto del proceso de desarrollo.

- Aplicable a diferentes juegos: el editor es capaz de generar comportamientos para las entidades de diferentes juegos o entornos de simulación. Según se explica en el artículo, el editor ha sido probado con

[1] *eCo*: `http://gaia.fdi.ucm.es/research/eco-behaviour-editor`

éxito en entornos tan diferentes como SoccerBots o para controlar a un robot Aibo.

- Asistencia al usuario: el objetivo principal presentado en este trabajo es facilitar a los diseñadores la tarea de creación de comportamientos mediante la reutilización. Esto queda reflejado en el editor mediante las diferentes herramientas de recuperación de comportamientos. En esta versión del editor sólo incorpora la recuperación basada en funcionalidad, pero las restantes se han incorporado en posteriores versiones.

Durante el desarrollo de esta Tesis, el editor *eCo* ha evolucionado incorporando más funcionalidades de las descritas en el modelo de la Sección 3.2. En 6.3 (Flórez-Puga et al., 2012a) se introduce la última versión del editor. Esta versión sigue siendo fiel a los principios con los que se creó *eCo*. Además, incorpora diversas mejoras de usabilidad y en cuanto a la gestión de las bibliotecas y de los comportamientos. La mejora más importante introducida en esta versión es la recuperación basada en bocetos. Gracias a esta técnica, el usuario puede realizar búsquedas de comportamientos parecidos al que está "dibujando". Además, el editor es capaz de sugerir comportamientos similares sin que sea necesario que el usuario construya una consulta, utilizando el comportamiento actual.

En el Apéndice A, se puede encontrar una descripción más detallada del editor de comportamientos.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Principales Aportaciones

El objetivo perseguido en esta Tesis Doctoral es facilitar el proceso de creación de comportamientos inteligentes para los NPC de un videojuego. Hemos observado que este proceso puede facilitarse mediante la reutilización de comportamientos construidos previamente. En esta línea, se ha desarrollado un modelo de creación de comportamientos que se basa precisamente en la reutilización.

En esta sección se resumen las principales aportaciones realizadas para conseguir este objetivo, en base a la discusión mantenida en los capítulos previos.

- Estudio de diferentes técnicas de representación de comportamientos en el ámbito de los videojuegos.

  Un primer paso hacia la construcción del modelo ha sido el análisis de las distintas técnicas existentes para la representación de comportamientos de los NPC. A raíz de este análisis hemos observado que, en primer lugar, los comportamientos son modulares, es decir, que los comportamientos complejos pueden descomponerse en comportamientos más simples. Además, en la mayoría de los casos también se observa que los comportamientos más simples son comunes a varios comportamientos complejos.

  El modelo propuesto se basa en explotar estas propiedades ofreciendo a los diseñadores los medios para reutilizar un conjunto de comportamientos creados previamente y almacenados en una biblioteca.

- Descripción de las características de la biblioteca de comportamientos reutilizables.

  Una parte fundamental del modelo propuesto es la utilización de una biblioteca en la que se mantienen los comportamientos indexados para

su recuperación. Como hemos visto, en 6.1 (Flórez-Puga et al., 2011) se describe la estructura de la biblioteca, así como un lenguaje de consultas para llevar a cabo la recuperación.

- Propuesta de un conjunto de funciones de similitud para comportamientos.

  La tercera aportación de esta Tesis ha sido proponer diferentes funciones de similitud para poder llevar a cabo la recuperación de los comportamientos almacenados en la biblioteca. Las funciones se dividen en dos grupos:

  - Basadas en atributos: cada comportamiento se describe mediante un conjunto de pares `(atributo, valor)`. Para realizar la búsqueda se compara el conjunto de pares de cada comportamiento con los de la consulta. Este tipo de funciones está descrito en 6.2 (Flórez-Puga y Díaz-Agudo, 2007). El principal problema de esta aproximación se encuentra en el cuello de botella de la adquisición de conocimiento. Para solventarlo, en 6.3 (Flórez-Puga et al., 2012a) se propone la automatización de la asignación de valores a los atributos.
  - Basadas en estructura: en este caso se utiliza la estructura de los comportamientos (la manera en la que están conectados los nodos y las aristas) para compararlos. Una de las ventajas de esta técnica es que, a diferencia de lo que ocurre con la similitud basada en atributos, no se necesita más conocimiento que el propio comportamiento almacenado en la biblioteca, con lo que se solventa el problema de adquisición de conocimiento. Además, las consultas se construyen "dibujando" un comportamiento, por lo que el usuario no necesita aprender un lenguaje de consultas específico. En 6.5 (Flórez Puga et al., 2008) hemos propuesto diferentes funciones de similitud estructurales.

- Propuesta de diferentes técnicas de recuperación de comportamientos de una biblioteca.

  En esta Tesis se han propuesto diferentes técnicas que permiten recuperar comportamientos de la biblioteca. Podemos distinguir entre:

  - Recuperación durante el diseño.
    - Recuperación basada en funcionalidad.
      Cada comportamiento de la biblioteca se asocia con un conjunto de pares que describen determinados aspectos de su funcionalidad. Para llevar a cabo la recuperación se utiliza

una función de similitud basada en atributos. Una primera versión de esta técnica puede encontrarse en 6.2 (Flórez-Puga y Díaz-Agudo, 2007) y una versión más avanzada en 6.3 (Flórez-Puga et al., 2012a).

- Recuperación basada en bocetos.
  Imitando ideas planteadas en el dominio de la recuperación de imágenes, hemos propuesto una nueva técnica de recuperación de comportamientos basada en utilizar como consulta un boceto del comportamiento que se desea recuperar. Utilizando una función de similitud estructural se recuperan de la biblioteca aquellos comportamientos cuya estructura es similar al boceto planteado. Esta técnica nos permite utilizar como consulta el propio comportamiento que el diseñador está editando pudiendo realizar recomendaciones sobre comportamientos similares que se encuentren en la biblioteca sin que sea necesario que el usuario genere una consulta de manera explícita. Los detalles de esta aproximación se han descrito en 6.3 (Flórez-Puga et al., 2012a).

- Recuperación durante la ejecución.
  En 6.6 (Flórez-Puga et al., 2009) proponemos la utilización de las técnicas de recuperación mencionadas anteriormente, no sólo mientras el comportamiento está siendo diseñado, sino también mientras está siendo ejecutado dentro del juego. Para ello, incorporamos a los comportamientos un nuevo tipo de nodo llamado *nodo consulta*. Este nodo tiene una consulta asociada. Cuando llega el momento de ser ejecutado, el nodo recupera un comportamiento de la biblioteca utilizando la consulta y lo ejecuta en su lugar.

- Implementación de una herramienta visual de creación de comportamientos basada en el modelo propuesto.

  Siguiendo el modelo propuesto se ha implementado una herramienta visual de creación de comportamientos para NPC a la que hemos dado el nombre de *eCo*. Esta herramienta aborda tanto los aspectos de autoría como de reutilización de comportamientos. En 6.2 (Flórez-Puga y Díaz-Agudo, 2007) hemos presentado una versión preliminar de *eCo* y, posteriormente, en 6.3 (Flórez-Puga et al., 2012a) se puede ver una versión más avanzada que incorpora la mayoría de características del modelo.

- Demostración experimental de la validez de las técnicas propuestas.

  Mediante la realización de varios experimentos hemos demostrado que las técnicas utilizadas en el modelo son válidas y adecuadas para conseguir el objetivo propuesto.

En la publicación 6.3 (Flórez-Puga et al., 2012a) demostramos que existe una correlación entre la similitud estructural y la similitud funcional, de modo que los comportamientos que se comportan de manera parecida, también muestran un alto grado de similitud estructural. Por lo tanto, podemos valernos de técnicas como la recuperación basada en bocetos, que utiliza similitud estructura, para recuperar comportamientos que al ser ejecutados reaccionen de manera parecida.

En la publicación 6.7 (Flórez-Puga et al., 2013) se demuestra experimentalmente que el uso de la recuperación basada en bocetos permite obtener una reducción significativa en el número de pasos que tiene que realizar el usuario para llegar al comportamiento deseado, reduciendo así el tiempo de desarrollo.

5.2. Trabajo Futuro

Para concluir el capítulo, se presentan algunas líneas futuras de investigación:

- Investigación de nuevas funciones de similitud para mejorar la recuperación basada en bocetos.

  Como se ha expuesto anteriormente, en la recuperación basada en bocetos, para poder calcular la similitud estructural entre comportamientos grandes es necesario utilizar una heurística que reduzca la complejidad del proceso. Se considera interesante estudiar otras funciones de similitud para obtener unos resultados más precisos. En concreto, resultan prometedoras las funciones de similitud de grafos basadas en *kernels* (Harchaoui y Bach, 2007).

- Integración en el modelo de funciones de similitud estructural específicas para árboles.

  Dentro de la recuperación basada en bocetos no se hace ninguna distinción entre HFSM y BT. Sin embargo, la estructura de los BT es diferente, ya que estos son árboles. Otra línea de investigación interesante es la búsqueda de funciones de similitud estructural específicas para árboles. De esta manera sería posible comparar los resultados con las funciones utilizadas hasta ahora y comprobar si se obtiene alguna ganancia.

- Aplicación del modelo a otros dominios.

  Tanto los experimentos como los ejemplos mostrados en esta Tesis se han realizado con dominios de juego relativamente sencillos, como puede ser Soccerbots. Sería interesante aplicar el modelo a otros dominios

de juegos del "mundo real". De esta manera se podrían comprobar de primera mano cuáles son los beneficios y los problemas que surgen.

En la misma línea, también sería interesante explorar la aplicación del modelo a otros dominios fuera del de los videojuegos. Un dominio que parece adecuado es el de los *workflows científicos* (Gil et al., 2009). Las técnicas de recuperación propuestas en nuestro modelo pueden ser utilizadas para recuperar workflows de grandes colecciones como myExperiment[1].

- Completar *eCo* y dotarlo de mayor usabilidad

  eCo, el editor de comportamientos desarrollado como parte de esta Tesis, se ideó como un prototipo para demostrar la viabilidad de las técnicas propuestas en el modelo. Dado que se ha demostrado la utilidad de estas técnicas, resultaría interesante someter al editor a un proceso de refactorización y mejora para acercarlo a las necesidades reales de los usuarios finales. Un punto de partida pueden ser los resultados de las encuestas realizadas a los estudiantes durante el experimento descrito en 6.3 (Flórez-Puga et al., 2012a).

  El objetivo sería conseguir una herramienta estable que se pueda distribuir de manera pública a fin de que pueda ser utilizada por una amplia comunidad de usuarios.

[1] `http://www.myexperiment.org/`

Capítulo 6

Artículos Presentados

A continuación se incluyen los artículos editados que se aportan como parte de esta Tesis Doctoral.

6.1. Empowering Designers with Libraries of Self-validated Query-enabled Behaviour Trees

Cita completa

Gonzalo Flórez Puga, David Llansó, Marco Antonio Gómez Martín, Pedro Pablo Gómez Martín, Belén Díaz Agudo y Pedro Antonio González Calero. *Empowering Designers with Libraries of Self Validated Query-enabled Behaviour Trees.* En Artificial Intelligence for Computer Games. Springer. Marzo, 2011. P. 55–82. ISBN 978-1-4419-8187-5 (edición impresa) ISBN 978-1-4419-8188-2 (edición electrónica)

Resumen original de la contribución

Building the behaviour for non-player characters (NPC) in a game is a collaborative effort between AI designers and programmers. Programmers provide to the designers with the building blocks for specifying behaviours in the game, and designers use some combination of state machines, scripting and visual languages to build complex behaviours by composing the basic pieces the programmers provide.

Behaviour Trees (BTs) are the technology of choice for AI programmers to build NPC behaviour. Although BTs can be naturally built using visual languages that require no programming, in general, they are considered too complex for being built by designers without a programming background. In this chapter we propose a number of techniques for facilitating the collaborative work of behaviour design through BTs. We provide tools for creating and managing a library of reusable fragments of BTs, intended for both programmers and designers. Such library is accessed through retrieval mechanisms that also support the definition of query nodes in BTs that can be expanded at run-time. In order to harness such an expressive power in behaviour design, we also propose an extension to the component-based architecture that supports a number of sanity checks to validate BTs, both at design and run-time.

Referencia de citas bibliográficas

Ubisoft Montreal Studios (2008), Hocking (2009), Esmurdoc (2005), Games (2009), Crytek (2010), Rabin (2006, 2008), Isla (2005), Isla (2008), Krajewski (2009), Millington (2006), Atkin et al. (2001), Tai (1979), Zhang y Shasha (1989), Wang et al. (1998), Flórez Puga et al. (2008), Ierusalimschy (2006), Lakos (1996), Szyperski (1997), West (2006), Rene (2005), Buchanan (2005), Garcés (2006), Llansó et al. (2009), Flórez-Puga et al. (2008)

Empowering Designers with Libraries of Self-validated Query-enabled Behaviour Trees

G. Flórez-Puga, D. Llansó, M.A. Gómez-Martín, P.P. Gómez-Martín, B. Díaz-Agudo, and P. González-Calero

Abstract Building the behaviour for non-player characters (NPC) in a game is a collaborative effort between AI designers and programmers. Programmers provide to the designers with the building blocks for specifying behaviours in the game, and designers use some combination of state machines, scripting and visual languages to build complex behaviours by composing the basic pieces the programmers provide. *Behaviour Trees* (BTs) are the technology of choice for AI programmers to build NPC behaviour. Although BTs can be naturally built using visual languages that require no programming, in general, they are considered too complex for being built by designers without a programming background. In this chapter we propose a number of techniques for facilitating the collaborative work of behaviour design through BTs. We provide tools for creating and managing a library of reusable fragments of BTs, intended for both programmers and designers. Such library is accessed through retrieval mechanisms that also support the definition of query nodes in BTs that can be expanded at run-time. In order to harness such an expressive power in behaviour design, we also propose an extension to the component-based architecture that supports a number of sanity checks to validate BTs, both at design and run-time.

Gonzalo Flórez-Puga
Complutense University of Madrid, Spain e-mail: gflorez@fdi.ucm.es

David Llansó
Complutense University of Madrid, Spain e-mail: llanso@fdi.ucm.es

Marco A. Gómez-Martín
Complutense University of Madrid, Spain e-mail: marcoa@fdi.ucm.es

Pedro P. Gómez-Martín
Complutense University of Madrid, Spain e-mail: pedrop@fdi.ucm.es

Belén Díaz-Agudo
Complutense University of Madrid, Spain e-mail: belend@sip.ucm.es

Pedro González-Calero
Complutense University of Madrid, Spain e-mail: pedro@fdi.ucm.es

Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03)

Key words: behaviour trees, ontologies, software architecture, authoring tools

1 Introduction

Building the behaviour for non-player characters (NPC) in a game is a collaborative effort between AI designers and programmers. Programmers provide to the designers with the *building blocks* for specifying behaviour in the game, as a collection of parametrized systems, entity types and actions those entities may execute. Designers use some combination of state machines, scripting, visual languages and map editors to build complex behaviours by composing the basic pieces the programmers provide. Just to give a hint about the magnitude of the task, developing a game such as Far Cry 2 [22], according to [9], required an average number of 150 people (including testers) during 43 months, which results, making a conservative assumption of a 20% of designers, in 30 designers working for three years in creating game play content for a shooter.

Ideally, a detailed design document should serve as the specification contract between designers and programmers: before entering into production stage, it should be perfectly clear which building blocks the programmers should build and what building blocks the designers would count on for designing the game levels. However, in actual development, the design of the game usually becomes a moving target, with designers coming up with new requirements for programmers as new mechanics are explored. Furthermore, programmers overwhelmed by their current tasks can feel tempted to let designers use their dubious scripting skills to implement such additions, what, later on, will probably result in the programmer debugging a designer's script during crunch time.

A key problem in this process is that a good game designer may not have programming skills but nevertheless what a designer is actually doing most of the time is building portions of a software system. A possible solution for this problem is to hire designers who know how to program, which actually some companies do (Double Fine fired the whole level design department in the mid of the development of Pshyconauts, and hired fresh college graduates from Computer Science departments to script the levels [4]). Another approach, also used in industry, is to let designers use visual languages that are supposed to facilitate the process, by hiding the formal syntax of the programming language, and controlling through a GUI the sentences that can be built with the visual language. UnrealKismet, integrated in the Unreal Development Kit game editor [7], and Flow-Graph Editor, integrated in the Sandbox Editor of CryENGINE 3 SDK [3], are two of such visual scripting tools, that let designers model the gameplay of a level without touching a single line of code through some variation of data flow diagrams.

For AI programmers, according to the number of papers dedicated to the subject in the editions 3 and 4 of the AI Game Programming Wisdom book series [17, 18], Behaviour Trees (BTs) are the technology of choice for programming the AI of NPCs in different game genres. BTs have been proposed as an evolution for hier-

archical finite state machines (HFSM) intended to solve FSM scalability problems by emphasizing behaviour reuse [11]. In BTs instead of explicit transitions from one state to another, each node defines procedurally how to traverse its children. BTs are goal structures that represent how a high level goal can be decomposed into lower level ones until reaching the leaves of the tree, which contain primitive goals that can be achieved by available actions. In this chapter we propose a number of techniques for facilitating the collaboration between AI programmers and designers through the collaborative construction of BTs.

Although BTs can be naturally built using visual languages that require no programming, in general, they are considered too complex for being built by designers without programming skills. The use of different levels of abstraction, implicit transitions and arbitrarily complex control structures for composite nodes make BTs as expressive as general purpose programming languages, and therefore not convenient for designers to use. Nevertheless, BTs have been successfully used by professional game designers in released commercial games, by focusing designers on building BTs for high level strategic behaviour which relies on lower level reactive behaviour that programmers provide, typically also as BTs [12, 13]. Building upon this idea of BT fragments at different levels of abstraction, we provide tools for creating and managing a library of reusable fragments of BTs, intended for both programmers and designers. The library is equipped with an authoring tool that promotes to build new BTs by composing other BTs already in the library. Notice that such a library supports collaborations between different roles. Between programmers, that have an easy access to low level BTs designed by other programmers, between designers accessing high level BTs designed by other designers, and for designers to build high level BTs reusing those that programmers designed. Considering the number of people involved and the duration of the process, as hinted above, having a principled way of accessing somebody else's BTs can become crucial to avoid a situation where BTs become a new form of spaghetti code that only its author, if anybody, dares to modify.

A library of reusable fragments of BTs requires a query language and a retrieval mechanism that returns BT fragments relevant for a given need. The query language that we propose is based on a declarative representation of the game world, a *domain model* that names and classifies the types of entities available in the game, along with their properties, available actions and goals. The same language will be used to annotate BT fragments with the intended goal, as well as the restrictions on the type of entities that can execute the BT or the parameter values it can receive.

The possibility of retrieving BT fragments from a library, naturally leads to a second contribution of the work presented here. BTs can be extended to include *query nodes* that specify queries that will be executed at run time, resulting in the substitution of the query node with the retrieved BT fragment. This mechanism provides a controlled form of emergent behaviour, as well as an easy way to introduce variability in the responses of an NPC, and will also allow for high level BTs to automatically incorporate new BT fragments as they are incorporated into the library.

Having designers build BT fragments with parameters and query nodes may easily result in unusable BTs. This may also be the case for BTs with query nodes even

when designed by programmers, since BTs generated on the fly through this mechanism could be impossible to execute. Thus, in order to harness such an expressive power in behaviour design, we also propose an extension to the component-based architecture that supports a number of sanity checks to validate BTs, both at design and run-time, through *reflective components* that are able to validate a given behaviour tree.

The rest of the chapter runs as follows. Section 2 presents the *Behaviour Tree* model that will be extended in later sections. Section 3 presents the mechanisms of a library of reusable BT fragments, and shows how this naturally leads to extend BTs with query nodes. Section 4 presents the main ideas of a component-based architecture and how this can be extended to validate BTs. The chapter ends with a clarifying example and some conclusions.

2 Behaviour Trees

Finite-state machines (FSMs) are the most used technology for AI on games, easy to understand, deterministic and fast. Designers are also used to them, and they can be defined using simple (even graphical) tools. Unfortunately when they are used to define complex behaviours, FSMs require more and more states that can become the FSM hard to control.

A way to scale up FSMs is to consider that a state can hide another FSM to decide its actions. Instead of having a flat set of states, they are arranged in different levels, creating a hierarchical finite state machine (HFSM). Apart from adding more structure to the states, they ease the reuse of low-level FSM and provide different views of the HFSM depending on the detail they are observed, which facilitates their comprehension.

HFSMs expand the complexity of the AI of the NPCs that can be implemented with this technology but, obviously, they also suffer of their own threshold that makes them too complex. Curiously, the bottleneck in the FSMs and HFSM scalability are not the states, but transitions. Transitions grow much faster than states, and they become uncontrollable sooner.

A way to overcome this problem is to completely remove transitions. The resulting structure is not a (H)FSM anymore but it is useful anyway. Without transitions, an AI of an NPC is defined using a “cloud of states”, and a *procedural* way to choose which one is the active one. An AI of an NPC is not in a state anymore, but *executing a behaviour*. The selection mechanism that picks up the current behaviour hides the old nasty transitions, and plays the role of a referee. It can use any information about the virtual environment to arbitrate between them.

This new scheme is enriched with a new ingredient: behaviours (the old states) *can end*. Although a behaviour could last many game cycles, eventually it could decide that it has finished its labour and a new behaviour selection should be triggered. Even better, behaviours can inform about the success or failure of their execution,

information that enriches the decision-making process done by the selection mechanism choosing the new behaviour.

We can go even further considering the selection mechanism *as a behaviour* with sub-behaviours as *children*. With this fresh perspective, hierarchy comes to the surface: a behaviour could easily be implemented as a new low-level selection mechanism with its own sub-behaviours. This new decision structure is called *Behaviour Tree* (BT). This step is similar to that taken when moving from FSMs to HFSMs. Now we have compound behaviours that are decomposed on sub-behaviours.

BTs can be drawn using a tree representation, that could be confused with the FSM classical representation. Keep in mind that each edge in FSMs represents a *transition*, but BTs edges represent parent-child relationships; an internal "decision node" chooses among all its children which one should be executed next; all "transitions" between behaviours are decided by those selector nodes, not by behaviours themselves as was done by states in a FSM.

Notice that, depending on the context, nodes in a BT can be seen as states, behaviours, or actions. In this context, "behaviour" is a synonym of (transitionless) "state", while "action" corresponds to a primitive behaviour that can only appear as a leaf in a BT.

The literature is full of proposals for different decision nodes; for the goals of this chapter, we only require three of them: sequences, static priority list and dynamic priority list. For all of them, the child order is important: children nodes are not a *set* of behaviours, but a *list*.

Sequences are simple composite behaviours that execute their children in the order they are defined. Keep in mind that behaviours *end*, so sequence nodes wait until the current active child ends *with success* to launch the next one. If any child *fails*, the sequence also immediately fails, throwing the problem up in the hierarchy. Sequences end with success when their last child does.

To introduce static and dynamic priority list, a new concept must be first presented. Children behaviours can be guarded by *conditions*, indicating when that child can be chosen. Keep in mind that these conditions are *not* preconditions, because a valid candidate child (which condition is true) could, after all, fail: true conditions do not guarantee the complete correct execution of the guarded behaviour.

With conditions in mind, a *static priority list* node evaluates its children conditions in order, and activates the first one whose condition is true. The child order represents a *behaviour priority*, with the first child having a higher priority than the next ones. A *dynamic priority list* is similar, but it continuously *reevaluates conditions* of prior nodes to the active one, and switches to a higher priority node whenever possible, as soon as its condition becomes true. In contrast to sequences, priority lists *fail* if all of their children fail. If any child ends successfully the priority list also ends with *success*.

Although they are not important for this chapter, BTs usually provide with a second family of internal nodes known as *decorators*. Decorators have only one child, and they add or modify the original child behaviour. Examples of decorators are *control modifiers* (negating the child result, or forcing a concrete one) or filters (repeating the child behaviour while it succeeds, avoiding it to be fired too often

using a timer, etc.). Decorators bring into BTs the expressive power of a general purpose programming language [16].

Apart from the lack of transitions, other crucial aspect to overcome the scalability problem in FSMs is considering nodes as *behaviours* instead of *states*. This new point of view introduces the idea of a *goal* for every behaviour and, with this vision, design is simpler because hierarchy let designers think in terms of goals and subgoals instead of states and substates. Most actions have a primary goal along with a number of additional goals that depend on the action context [1]. For example, the primary goal of the action "move-to" is to change location from *x* to *y*, but in an urban fight scenario we can be moving to get under cover from enemy fire or to assist a fallen comrade. Having actions focus only on their primary goal can sometimes lead to unintelligent behaviour. For example, if an agent is moving to a destination and is attacked, it will continue to move, even when it would be totally destroyed by doing so. Instead of adding conditional statements to every action that specify all the exceptions to normal behaviour we can handle multiple goals and make them part of a hierarchy, which prioritizes goals higher up in the hierarchy, i.e., staying alive is more important than moving to point *y*, so if some condition higher up in the behaviour tree becomes activated for self protection, the whole branch being executed can be pruned.

Hierarchy also supports reusability, because a BT fragment can be seen as a black box that provides a specific behaviour that can be attached to more complex BTs as a child. Throughout the game production, more and more behaviours (general and enough reusable BTs) will be available for the designers' team, saving time from reinventing the wheel.

2.1 A Domain Model for Behaviour Trees

For reusability becoming true, reusable BTs should allow some kind of parametrization. For example, designers may build a BT for an enemy that attacks using an available weapon and picks an item up afterwards. Although the concrete weapon and item could be hard-coded in the BT, this spoils nearly all opportunity for reusing it, so an elaborate mechanism to specify parameters should be available. The system should be good enough to let parameters be bound both in design and runtime, depending on the circumstances.

Keep in mind that both FSMs and BTs are *static* structures used to model *NPC AI*. In runtime, the same FSM could be used for multiple NPC simultaneously, each of them storing the current state and other information needed to "run" the FSM. Something similar occurs for BTs, where each NPC should keep track which *behaviours* are activated, which ones have failed, and so on. For parameter passing between nodes, the NPC runtime structure is enriched with an "execution context" (or *blackboard*) specific for each NPC, where behaviours read information (attribute-value pairs) to be used in the decision-making process using the guards (conditions). The set of attributes in the context is the portion of the game state that can be accessed

by the NPC. Values will be specified by designers during development (for example to force an NPC to pick up a *concrete* weapon), or written by some actions (leaf behaviours) in runtime (for example the treasure found by a search behaviour).

In order to be able to reason with BTs independently of the underlying game engine, we need to model the context and parameter passing mechanisms. Furthermore, we need to specify the collection of goals and the restrictions on the type of entities that can execute the BT or the parameter values it can receive. We propose the use of ontologies to represent both, the knowledge and the entities. Ontologies are a standard mechanism for knowledge representation, based on conceptual hierarchies, defined using the *is-a* relation where abstract concepts are located on the top of the taxonomy whilst specific concepts are located in its leaves.

To model the knowledge on our domain we use a *behaviour ontology*, which provides different classes used to categorize the behaviours in terms of the *goals* they fulfill. In the ontology we can find behaviour classes like *Attack Behaviours*, *Defend Behaviours* or *Resource Gathering Behaviours*. Each class can have several instances, that represent the different behaviours for that goal. For instance, in the class *Resource Gathering Behaviours* we can find `Steal Resource From Weakest Player` (Figure 1) or in the class *Attack Behaviours* we can find behaviours like `Long Range Stealth Attack` or `Hand To Hand Stealth Attack`.

To classify the entities that form the context in which behaviours are executed, we use an *entity ontology*. In the top of the entity ontology we can find, for instance, classes like *Alive*, that represent the alive creatures. Going down through the ontology, we will have subclasses like *Monster* and *Player* that are alive creatures. *Player*, in turn, subsumes the *Human* and *Computer* categories that respectively represent the player's avatar and an AI controlled avatar.

Additionally, a set of relations exists between behaviours and entities. These relations are used to express the restrictions on the parameters of the behaviours.

Parameters are referenced in two places in the BT:

- The set of parameters that will be used in the BT is declared in the root of the tree. Each declaration consists in three elements: the relation between the behaviour and the entity in the parameter, the class from the entity ontology that will be the parent of the entity and the name that will be used to reference the parameter later in the BT.
  For instance, in the `Steal Resource` BT in Figure 1, we have the parameter declaration `(hasTarget, entity: PLAYER)`. This means that an input parameter is declared for the relation `hasTarget`. The entity type of the input parameter is `PLAYER`, which means that the *target* of this behaviour can only be a PLAYER (resources can only be stolen from players, whether they are human or AI controlled). To reference this input parameter inside the BT the identifier `entity` should be used.
- In the invocation of other BTs or leaf behaviours, parts of the execution context are bound to the input parameters of the invoked behaviour in the parameter passing mechanism.
  This is the case of the invocation of the leaf `Search` by the BT `Steal Resource From Weakest Player`. In this case, the value of the parameter `entity` from

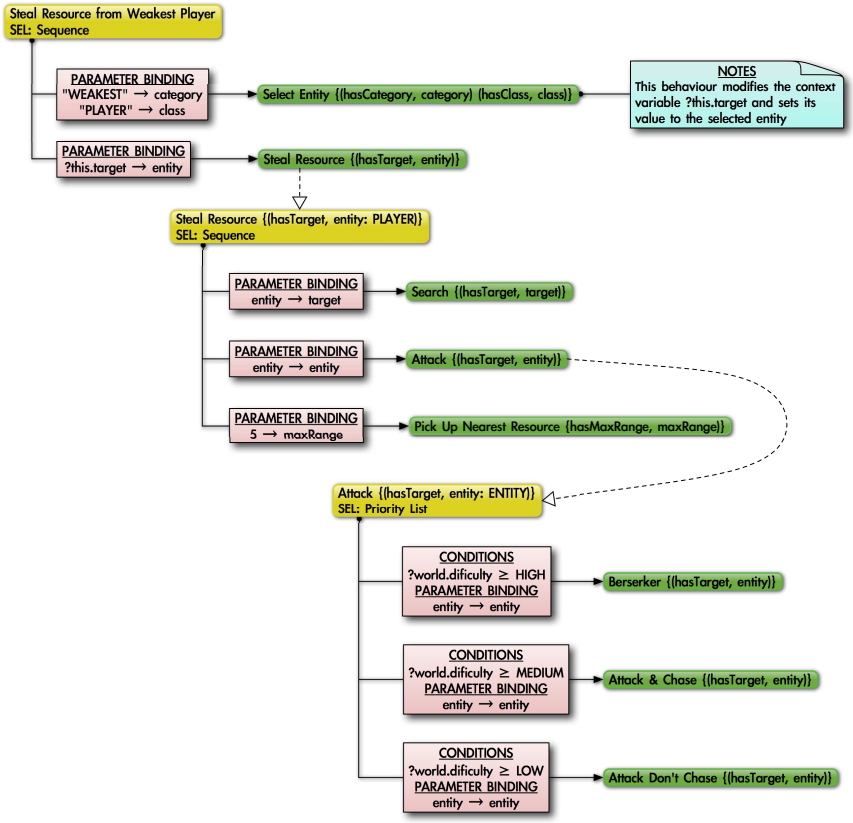


Fig. 1: Behaviour Tree for *Steal Resource From Weakest Player*

`Steal Resource From Weakest Player` is bound to the input parameter `target` of `Search`.

The NPC context provides a storage structure similar to that one found in object-oriented programming languages. For example, `?this` will refer to the NPC executing the BT (with information such as `?this.health` or `?this.aggressive`), `?world` will refer to the virtual environment state (`?world.time`) and `?target` will refer to the game entity target for the behaviour (`?target.distance`). As a conclusion, NPC context provides a way to consult the *game state*, both of the virtual environment and the NPC state itself.

Using this notation we can represent a tree such as the one shown in Figure 1

3 A Library of Reusable Behaviour Trees

One of the main advantages of using Behaviour Trees for the AI design is the reusability they provide. The main reusability components are basic actions provided by programmers, but BTs combining several nodes could also become reusable behaviours to be chained into more complex BTs. For example, a programmer could create a `StealthWalking` BT using simple actions that look for dark zones and walk through them. Once it is available, other designers could use it to create behaviours such as `SurpriseAttack` or `Spy`.

BT reusability is possible because of two features that are common in most of everyday videogames. First of all, modularity in behaviours: complex behaviours can be decomposed into simpler behaviours that are somehow combined. Second, simpler behaviours tend to recur within complex behaviours of the same game, or even in different games of the same genre. For instance, in an action game, a `Hand to hand attack` could be a complex behaviour that is composed of two simpler behaviours like `Go to (enemy)` and `Attack with knife`; on the other hand, `Long range attack` could be composed of `Go to (cover)` and `Shoot ray gun`. Both features are useful to build new complex behaviours based on simple behaviours as the reusable building blocks.

Programmers and designers should keep an eye on BT reusability in two aspects. They should create BTs trying to make them general enough to be later reused. And, at the same time, they should try to reuse previously made BTs, instead of reinventing the wheel creating the same basic behaviours again and again. This is quite important because, although BTs make easier the creation of behaviours for NPCs, it still takes a lot of time to wire them up because of the large number of behaviours that can be involved in the process (Halo 2 had an average of 60 different behaviours arranged in 4 layers [11]).

To assist game designers in the creation and edition of BTs we have developed the *eCo Behaviour Editor*. The *eCo* editor is an authoring tool that provides the users with a graphical interface which allows them to manually create or modify behaviours just by "drawing" them. It includes tools for loading, saving and importing the behaviours from disk, drawing and erasing nodes and edges from the trees, and specifying their content. Once the behaviour is complete, it is possible to use the included code generation tool to generate the source code corresponding to the behaviour.

Nevertheless, the more outstanding feature of the *eCo* editor is BT reusability. Every manually designed behaviour is *stored and indexed* in a *database* that allows easy *BT retrieval* of previously stored behaviours. We use techniques imported from the Case Base Reasoning (CBR) area, where data (cases) are stored in such a way that search becomes more than only matching.

CBR is based on the intuition that new problems are often similar to previously encountered problems, and therefore, that past solutions may be reused, directly or through adaptation, in other situations. CBR systems typically apply retrieval and matching algorithms to a case base of past problem-solution pairs. Another very important feature of CBR is its coupling to learning. A strong effort has been done

in the CBR community to solve the problems of similarity and adaptation in different contexts, with different approaches to case representation, organization and storage, and amount of knowledge, from knowledge intensive to data intensive approaches.

CBR is specially well suited to deal with the modularity and reuse properties of the behaviours; it assists the user in the reuse of behaviours by allowing her to query a case base. Each case of the case base represents a behaviour. By means of these queries, the user can make an approximate retrieval of behaviours previously created, which will have similar characteristics and satisfy some conditions. The retrieved behaviours can be reused, modified and combined to get the required behaviours.

Although the more important component of each case is the BT itself (the behaviour that want to be retrieved), they also store *metainformation* that is used in the search process. We use XML files to store all this information, that is defined by the following attributes:

1. **Header**: includes the *case number*, used to identify the the case in the case base, and a *textual description* that describes in natural language the behaviour represented by the case.
2. **Goals**: this attribute enumerates the list of goals from the *behaviour ontology* satisfied by this behavior.
3. **Parameters**: is the set of parameters received by the behaviour (for example the enemy to attack, or the weapon to use), along with the restrictions of type of each one of them. The type is built from the classes in the ontology which an individual belongs to.
4. **Descriptors**: is a set of restrictions declared over the game state (context variables such as `?this`, `?target` or `?world` mentioned previously). The values of the descriptors can be either symbolic or numeric. The descriptors specify under which circumstances of the game state is appropriate to run the behaviour.

As an example, Table 1 shows the set of behaviors that satisfy the goal *Attack*.

When a designer creates a new BT, she must enrich it with all this information. Although this could be seen as tedious and useless, they could be used it later while retrieving previously stored BTs to be mixed with new ones.

We distinguish between two types of queries: functionality based queries and structure based queries. In the former, the user provides a set of *descriptors* to specify the desired functionality of the searched behaviour. In the latter, a behaviour is retrieved whose composition of nodes and edges is similar to the one specified in the query.

3.1 Functionality Based Retrieval

The most common usage of the CBR system in the editor is when the user wants to obtain a behaviour similar to a query in terms of its functionality. The functionality is expressed by means of a set of descriptors regarding the game state.

Case	Parameters	Goals	Descriptors
C_1	**Hand To Hand Stealth Attack**		
	(hasTarget, entity: ALIVE)	Attack	?target.distance ≤ MEDIUM ?this.personality = STEALTHY ?this.defensive ≥ MEDIUM ?this.health ≤ MEDIUM ?this.underAttack = LOW ?world.time = NIGHT
	Tries to approach an enemy without being noticed and attacks him using a close range, stealthy weapon. The entity executing it must remain undetected for the behaviour to be effective.		
C_2	**Long Range Stealth Attack**		
	(hasTarget, entity: ALIVE)	Attack	?target.distance ≥ MEDIUM ?this.personality = STEALTHY ?this.defensive ≥ MEDIUM ?this.health ≤ MEDIUM ?this.underAttack = LOW ?world.time = NIGHT
	Looks for cover in the surroundings and attacks the enemy with a stealthy weapon. The entity executing it must remain undetected for the behaviour to be effective.		
C_3	**Berserker**		
	(hasTarget, entity: ALIVE)	Attack	?target.distance = MEDIUM ?this.personality = BRUTE ?this.aggressive = HIGH ?this.health = HIGH
	Attacks an entity with the most powerful weapon available and without caring about own safety. This behaviour is used for very aggressive entities. A defensive entity will not show this behaviour.		
C_4	**Grenade Attack**		
	(hasTarget, entity: ENTITY)	Attack	?target.distance = HIGH ?this.personality = BRUTE ?this.aggressive ≥ MEDIUM
	Throws a grenade to an enemy and takes cover to avoid being affected by the explosion.		
C_5	**Elusive Attack**		
	(hasTarget, entity: ALIVE)	Attack	?this.personality = TIMID ?this.aggressive ≤ MEDIUM ?this.defensive ≥ MEDIUM ?this.health ≤ MEDIUM ?this.underAttack ≥ MEDIUM
	Approaches the enemy and shoots him while trying to cover behind the objects in the game world and zigzags to avoid being hit. It'is a defensive behaviour useful when the entity is being attacked or when the health is low.		

Table 1: Behaviors that satisfy the goal *Attack*

The *eCo* editor provides a query form, shown in Figure 2, for the user to enter the parameters of the query. The attributes that form a query are:

1. **Goals**: goals of the behaviour ontology that must fulfill the retrieved behaviour. The class of the goal can be selected in the tree on the left side of the query form,

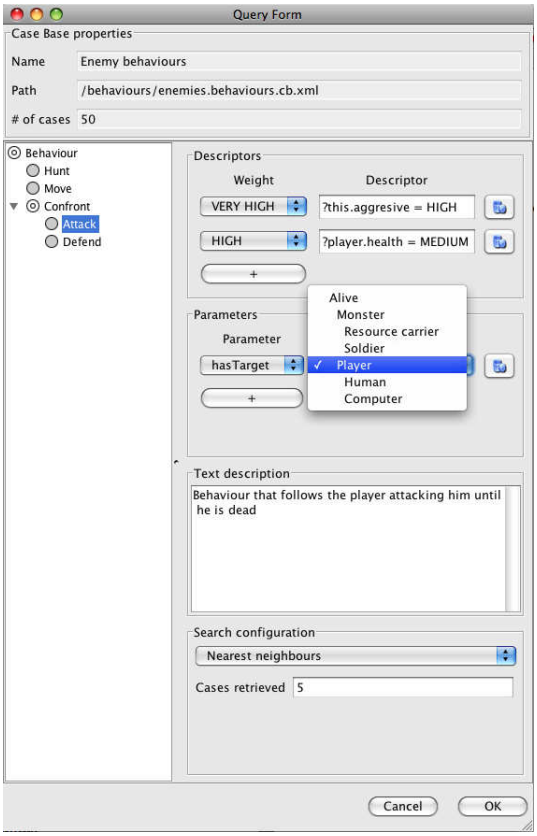


Fig. 2: Retrieval interface

that shows the behaviours taxonomy. The query may only retrieve behaviours for the selected class or any of its subclasses.
2. **Parameters**: restrictions on the type of the input parameters of the retrieved behaviours. For example `weapon` should be a `firearm`.
3. **Descriptors**: a set of restrictions declared over the game state that describe the behaviour to be retrieved.
4. **Weights**: the weight of each descriptor in the final similarity calculation.
5. **Textual description**: a natural language description of the behaviour that will be compared with the description in the header of the cases. The textual description allows the user to fine-tune the search.
6. **Cases retrieved**: the maximum number of behaviours the user wants to be retrieved.

The execution of the query goes as follows. First of all, the cases for the *Goal* specified in the query are retrieved. The similarity with the remaining cases is considered to be 0. If the user has specified any restrictions on the *Parameters*, they are

checked. Any candidate that does not satisfy the *Parameters* restrictions is excluded from the candidate set (again, its similarity is 0).

Then, the attributes of the query are compared to the attributes describing the BTs in the case base using a similarity function. Given a query, Q, and a case from the case base, C, the similarity value is obtained as follows:

$$sim(Q,C) = \begin{cases} \bullet \text{ The class of C doesn't belong to the goals of Q} \Rightarrow 0 \\ \bullet \text{ The restrictions on parameters in Q don't hold in C} \Rightarrow 0 \\ \bullet \text{ otherwise} \Rightarrow sim_{atr}(Q,C) \end{cases}$$

$$sim_{atr}(Q,C) = \sum_{d \in D(Q,C)} w_d \cdot sim_{loc}(Q_d, C_d)$$

$$D(Q,C) = Q.\text{descriptors} \cap C.\text{descriptors}$$

$$sim_{loc}(Q_d, C_d) = 1 - \frac{|Q_d.\text{value} - C_d.\text{value}|}{\text{size}_d}$$

$D(Q,C)$ is the intersection of the sets of descriptors of Q and C and size_d is the size of the interval of valid values for a descriptor d. Each w_d is the weight corresponding to the descriptor d, normalized so that the sum of all the w_d is 1.

To obtain the global similarity value between each of the cases and the query, the weighted similarity of the *Descriptors* is aggregated with the similarity due to the *Textual description* of each behaviour. Using a string similarity measure, the *Textual description* of the query is compared to the description in the *Header* of the case.

Finally, the candidates are sorted by their similarity value and the most similar ones to the query are retrieved.

3.2 Structure Based Retrieval

In some circumstances, the behaviour designer knows the general structure of the Behaviour Tree (i.e. the distribution of the nodes and their generic functionality). In these situations, it would be easier and faster for the designer if he could "sketch" the tree and let the editor find a similar one in the case base.

The user can draw a tree with empty nodes (a tree pattern) and let the system find a similar one with all nodes defined. But, by entering this data alone, the retrieved BT would be similar to the query only in terms of its shape. The behaviour it implements could be any. Hence, we need to allow the behaviour designer to point out the desired functionality of the retrieved tree and then, compare the desired functionality with the functionality implemented in the nodes of the trees in the case base.

The functionality of the drawn nodes is expressed by linking each node to a *Functionality Query* that the user must build to express the desired behaviour that should be contained in the node. The linked functionality queries are compared to

the descriptors in the nodes of the behaviours in the case base during the query process.

Keep in mind that a *functionality based query* (previous section) could be specified using one of these new structure based queries, drawing just a root node with no children. In that sense, we can see structure based retrieval as an additional refinement search step, where the designer wants to impose some structural restrictions to the children nodes. Retrieval compares the sketched tree with those BTs in the case base, using any of the existing techniques in the literature for comparing ordered trees (like [21, 25, 23]).

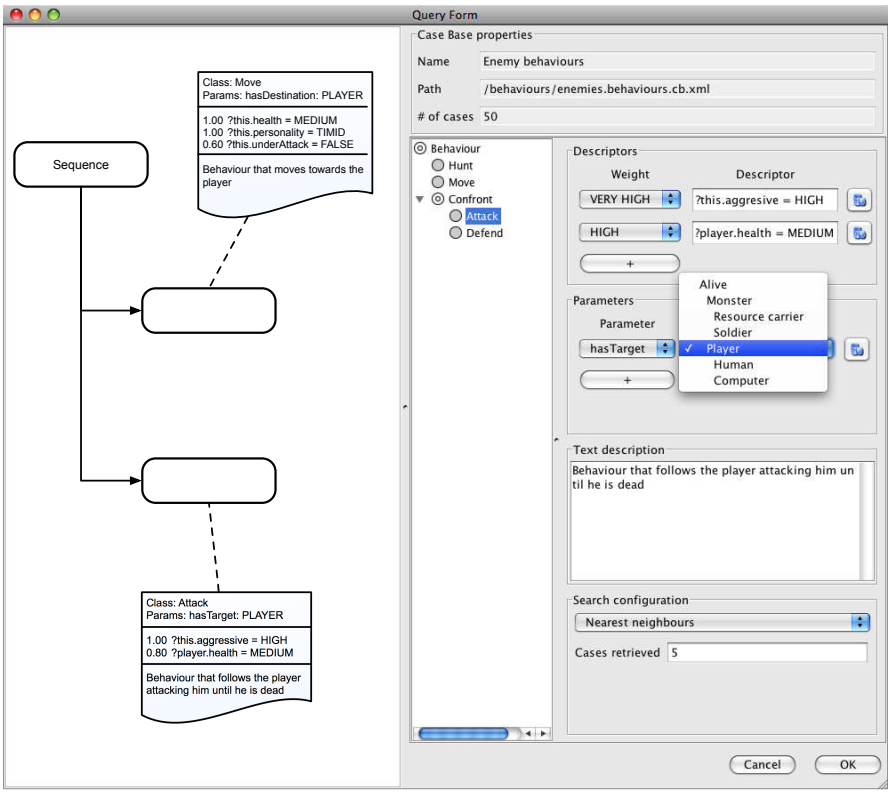


Fig. 3: Structural queries

Our approach to these structure based queries is to use the drawing facilities of the editor to “draw” the Behaviour Tree pattern, and then assign functionality based queries to the nodes, which will show the functionality of each node. Figure 3 shows the query editor for the structure based queries. In the left pane the user can draw a behaviour pattern and in the right pane he can specify the desired functionality of the retrieved behaviour by entering a functionality query. Additionally, each node

can be linked to another functionality query, as we have already mentioned, to tune up the search.

Further explanations regarding functionality and structure based retrieval can be found in [5].

3.3 Query Nodes at Run Time

Reusability and modularity are important advantages of using Behaviour Trees. Each BT represents an abstraction that can be reused as a composing piece of other BTs. Different BTs are created independently during the game design phase and they can be assembled as pieces of other existing BTs. The collection of game BTs includes different ways of solving a certain goal, e.g. different ways of getting food or stealth walking.

The search facilities included in our *eCo* editor described in the previous section provides *static reuse*: once the behaviour designer has chosen a suitable BT provided by the query, it is tied to the new BT been created. However, throughout the game development, both programmers and designers add more and more reusable BTs that could have been also suitable (even better) for those searches done previously. Then, to make the process consistent and useful it is important to review the pre-existing BTs that include a certain goal to check if it is convenient to assemble newer BTs (representing new ways to solve certain goals). This consistency checking process generates an extra effort that is sometimes skipped. That means that the behaviours added in the late design phases are not taken into account by the behaviours that were included in the early design phases.

To address this problem we propose a dynamic approach where the CBR system is queried at run time to find the most appropriate behaviour from a case base of implemented behaviours using BTs. The CBR processes work always with an up-to-date behaviour case base that allows retrieving the most convenient behaviour according to a certain query using the whole collection of designed behaviours and avoiding the extra cost of pre-checking its adequacy with newer behaviours.

Keep in mind that the reusing possibilities described in the previous section was an extra functionality provided by the *eCo* editor, that do not require runtime infrastructure in the BT framework. However, runtime queries require a new BT node, called *query node*, that stores the query attributes specified in design time, and makes the *BT retrieval* at runtime.

The attributes that describe these queries are the same ones used in the queries at design time (Section 3), adding a new *requery* field. Once the behaviour has been retrieved and is running, there may occur changes in the game state that would make another behaviour more suitable for the current situation. Using the *Requery* parameters we can specify the conditions or changes in the game state that should make the system repeat the query. Note that, although the query is done again, the results can be the same. In that case, the behaviour being executed is not restarted.

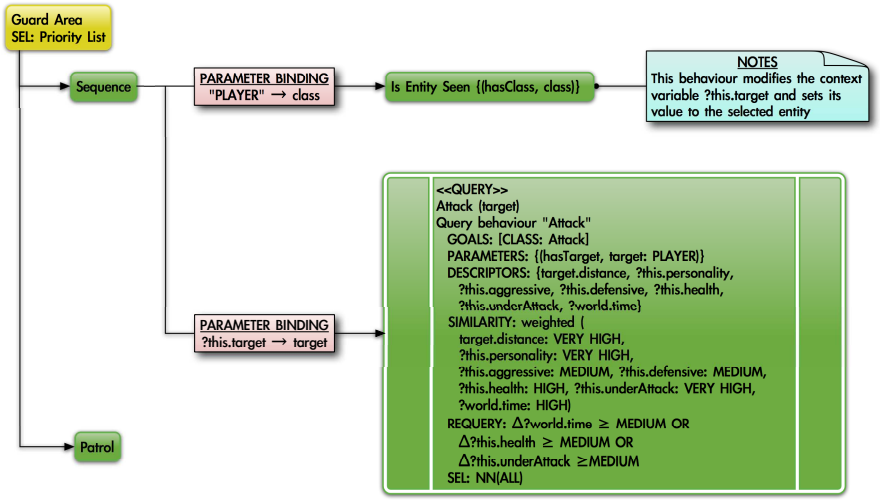


Fig. 4: Guard Area Behaviour

Although we have defended the advantage of runtime queries because they use all the available BTs, they provide an even more important benefit: they can use the *current world state* to select the more suitable behaviour. Parameters can now refer to the complete game state (`?this`, `?world` and `?target`), not only to static restrictions on the input parameters.

The retrieval process is very similar to the one explained for the functionality based retrieval. The main difference is that the values of the descriptors are not specified in the query. In this case, the query specifies the relevant descriptors and the values are taken from the game state at runtime, at the instant of time that the query is run.

Figure 4 shows an example of a *query node* that retrieves an Attack behavior.

4 Reflective Components

Although runtime search of BTs provides a lot of advantages against the static search at design time, they can become quite dangerous because the retrieved BT could not fit the NPC features. For example, an NPC could query for a behaviour to run away form the player, and receive a behaviour that uses a nearby car. At runtime, the system should check if the retrieved BT is suitable for the target NPC answering questions such as whether the NPC can drive.

Prior to explain our proposal to solve this problem, we need to introduce some implementation details about how game entities are usually coded. The runtime object-management system is in general an important part of a videogame, and cre-

ating this piece of code takes a great amount of time. To mention just two examples, a mature game such as Half-Life dated at 1999 has more than 65.000 non-empty no-comments lines of code on that module while Far Cry at 2004 exceed 95.000 lines of C/C++ code[1] even though the majority of the module was actually written in LUA [10].

That reveals that when creating this module we should try to design it to promote reusability in the sense that every single piece of code general enough to be used on a different title should be reused.

When we inspect how this module is usually coded, we find that it was traditionally based on an inheritance hierarchy, where all different kinds of entities derived from the same base class often called `CEntity`. Some of the consequences of this extensive use of class inheritance were an increase in the compilation time [14], a code base difficult to understand and big base classes. To mention just two examples, the base class of Half-Life 1 had 87 methods and 20 public attributes while Sims 1 ended up with more than 100 methods. The consequence is the well known *fragile base class problem* [20].

Due to all these problems, today developers tend to use a different approach, the so called component-based systems [24, 19, 2, 8]. Instead of having entities of a concrete class which define their exact behaviour, now each entity is just a *component container* where every functionality, skill or ability that the entity has, is implemented by a component. From the developer point of view, every component inherits from a specific class or interface (called, for example, `IComponent`), while an entity becomes just a list of `IComponent`s.

As the components are now generic objects with a common interface independent of their functionality, the usual method invocation is not enough. We cannot have a piece of code calling a method like `moveTo()`, because no such method even exists. What we have now is a *component* (a class called for example `MoveTo` that inherits from the previous `IComponent`) that is able to move the entity from one point to another; however externally this is just an `IComponent` indistinguishable from other.

The communication is therefore performed in a different way, using message passing. The `IComponent` is viewed as a *communication port* that is able to receive and process messages. A message is just a piece of data with an identification and some optional parameters (the implementation may vary from a plain `struct` with generic fields used in different ways depending on the type of message, to a base class such as `CMessage` and a hierarchy of messages). Components have a method like `handleMessage()` that is called externally to send the piece of information to it; depending on the concrete component, the message will be ignored or processed accordingly. In this scenario, entities play the role of the broadcaster of messages. Both, internal components and external modules, may send messages to the entity that are automatically distributed among all its components. Message types a component intercepts and processes usually corresponds with the basic entity actions this component can carry out so, an entity is able to executes so many actions as

[1] Lines Of Code (LOC) obtained using SLOCCount by David A. Wheeler.

the sum of messages its components can process. For instance, when the AI component (that which provides the entity with the ability to *think*) wants to move the entity from one point to another, it *sends* a `MoveTo` message to *all* the components of its entity. The component that implements the ability of movement (`MoveTo` component in our previous example) intercepts the message, calculates the path to be followed and emits periodically `UpdatePosition` messages to notify other components (graphical and physical among others) the change of the position.

As entities are now just a list of components, the concrete components (or abilities) that constitute them may be specified in an external file (usually known as *blueprint*) that is processed in execution time. This approach eases the creation of new kind of entities, because it does not require any development task but just the selection of the different skills we want our new entity to have from a set of components.

The approach also fosters the reuse of the components in other projects. As the responsibility of every component is neatly defined and it is in charge of just a small set of tasks, most of them are general enough to be useful in other applications.

In order to allow fine-grained adjustment of the behaviour (or skills) of different entities, their definition may also set the values of different attributes that components use as parameters of their behaviours. For instance, the component that provides the entity with the ability of picking up objects may use an attribute that specify the strength of the entity.

Keep in mind that entities construction in runtime is now *generic* due to the *blueprint* file described previously. Therefore, the concrete parameter values (such as the strength of each NPC race) must be also provided as data instead of being hard coded in source. This information is also provided in an external file, known as *archetypes*, containing the default values for each parameter of each entity in the *blueprint*. Map files for game levels will have the opportunity of override the default archetype values for some concrete entities, providing, for example, more strength than the default one to a specific NPC.

As an example, Figure 5 presents a `Patrol Soldier` entity built by components. This figure contains parts of the both mentioned entity descriptions files where the blueprints file reflects the abilities of the entity as a collection of components and the archetypes file displays the attribute-value pairs that makes the fine-grained data oriented entity description possible. The blueprints shows that a `Patrol Soldier` can be rendered and animated, can collide with other physic entities, execute BTs, walk from one place to another, etc. whilst the archetypes file sets the entity attributes to their default values.

Note that both entity description files just add information to the *entity ontology* described in Subsection 2.1, where the `Patrol Soldier` would be a specialization of the *Computer* category that represents an AI controlled avatar. So entities in the blueprint and archetypes files must fit with entities described there.

It is important to stress that our entity ontology just simplify the entity distribution, in a high level, through *is-a* relations but these relations *do not* involve that a child concept has all the abilities that its parent ontology concept has. Entity ontologies are excellent mechanisms to take high level decisions, but their *is-a* relation is

```
<!- blueprints file ->
<blueprints>
   <entity type="Patrol Soldier">
      <component type="AnimatedGraphic"/>
      <component type="Physic"/>
      <component type="BTExecuter"/>
      <component type="MoveTo"/>
      <component type="ShootTo"/>
      <component type="HandToHandAttack"/>
      <component type="Skills"/>
   </entity>
   ...
</blueprints>
...
<!- archetypes file ->
<archetypes>
   <entity type="Patrol Soldier">
      <attrib name = "life" value = "100"/>
      <attrib name = "speed" value = "1.5"/>
      <attrib name = "arm" value = "gun"/>
      <attrib name = "arm" value = "rifle"/>
      <attrib name = "model" value = "patrol_soldier.n2"/>
      ...
   </entity>
   ...
</archetypes>
```

Fig. 5: `Patrol Soldier` entity built by components.

not a good idea to implement low level details in big projects as it has been exposed in this section. That is the reason why the entity ontology is not translated in hierarchy classes when implementing our games in a programming language such as C++ but in entities built by components.

The reusability that components give us comes at a price, though. As the entity definition is made from text files, the consistence of the created entity class is not guaranteed. Prior the use of components, when new entity class was developed completely in a programming language such as C++, the compiler itself checked if the new class was complete before allowing programmers to create an object of it. Therefore, an entity with the ability of, say, walk to a location was *always* able to set the `walk` animation; otherwise the `setAnimation()` method invocation would not have compile.

When the declaration of entities becomes the addition of a set of lines in a text file, developer may forget to provide the entity class with some ability that is needed by other components. Following with the previous example and noticing the `Patrol Soldier` entity blueprint (Figure 5), if the entity has the ability to walk from one point to another (that implies the blueprint file states that the entity possesses a particular component such as the `MoveTo` component), it should also be

able to change the animation presented on the screen (possessing another component such as the `AnimatedGraphic` component) because the `MoveTo` component just sends a `SetAnimation` message and update regularly the entity position sending `SetPosition` messages.

Our solution to this problem is what we call *reflective components* [15]. The technique consists on enhance components with some methods that allow at design and even at runtime to check whether an entity is able to perform an action (and therefore has that particular ability).

During runtime, components that are related to the behaviour of the entity (*AI components*) such as those that manage BTs (`BTExecuter` component in Figure 5), send messages to the entity they belong to order which actions must be executed. This is due to AI components do not have the ability of executing these primitive entity actions because they only perform the decision-making process. In that sense, an entity with just the `BTExecuter` component is not complete, because it is not able to actually execute the tasks that the AI selects.

Due to now entities are specified in terms of their components, and that a component can be seen as an ability that an entity has, it makes sense therefore to try to identify the failures related to the inherent nature of the entities using such a description. The easy (and naive) approach is to make direct associations between basic actions (or messages that represent them) and components which are capable of executing these actions (process these messages).

Nevertheless, this approach would not be enough. Sometimes a component could not be able to carry out an action, although it has the ability to do it, either because it needs the collaboration of other components, which may not be in the entity, or because the component cannot correctly execute the action with the parameters associated with this action.

Let us imagine a situation where the `BTExecuter` sent a `MoveTo` message to makes the entity walks. The only existence of the `MoveTo` component would not assure the correct execution of the action since the `MoveTo` component would send `setAnimation` and `SetPosition` that other components should process. In the same way, the existence of a `AnimatedGraphic` component would not assure the correct execution of a `SetAnimation` message because the 3D model associated to the component could not have the given animation.

In order to manage both kinds of errors and with the purpose of giving a fine-grained approach, the methods we propose to enhance the component based systems will accept messages that encapsulate actions to ask whether they are able to handle a concrete message according to their configuration. So, we shall query them using the same messages that BT actions (or other kind of *AI systems*) generates during the game execution. Then, if any component needed the collaboration of other components, it would only have to query the entity it belongs with the same message that it would generate during runtime and finally the component would return whether the collaboration succeeded. Furthermore, as messages and components are parametrized, the new check methods can carry out fine-grained approach using them in the association process.

So, in order to check whether a full BT can be carried out by an entity, the `BTExecuter` component, for every action of its behaviour, just has to query to their entity components if some of them are able to execute it. Note that this is general enough. In games where a task may be perform using different methods, each entity capable of performing that task will be provided with the component that execute it using a particular method. As the behaviour component queries for the *ability* of executing the *task* instead of asking for the particular component that implements a method, the consistency check will work.

Again, we have a coarse-grain approach, though; not due to the reflective components but the BT action iteration. Just iterating over the list of actions of the BT is not precise enough since, in this way, the system would only validate or invalidate associations between BTs and entities, but if the system invalidated an association, it would not locate where and why this association was invalidated, so it can not be fixed easily.

Therefore, a fine-grained approach should locate which branches of the BT were not able to be carried out by the entity and which the node and the reason that made it crash. Bearing in mind that a BT may have different decision nodes (Section 2) and the chosen children to be evaluated depends on them, different kinds of nodes have to be evaluated by different methods.

As its name denotes, a sequence represents a chain of behaviours. Thus, to validate a sequence, all its children nodes must have been validated with the entity before. Therefore, if there was one node of the sequence that was not validated, the whole sequence would be invalidated knowing why and where the problem would be.

But both static and dynamic priority list represent a behaviour that chooses between different ways of resolving a problem. So only one of the child nodes would be executed during the game rather than in previous example, in which all the children would be executed (sometimes more than one child is executed, but only if there are more children available and child guards changes during the selector execution). As a result of this, a fault detected in a child of the selector was less critical than faults detected in a child of sequences. This is because there were probably another choice (other child) selectable by the selector. Therefore we could call these faults as warnings, instead of failures, if the child node that fails has at least one other brother node that has been validated.

So, although interested readers are referred to [15] for more details, to summarize, in order to validate a BT with an entity, the `BTExecutor` would try to validate the root node of the BT with the entity it belongs and this would be recursively spread to all the nodes of the tree. Finally, the leaves of the tree, which contains the final actions, would be checked with the entity, passing the messages that they generates during runtime to the check method of the entity. The entity would broadcast the passed message to its components and they would validated/invalidated the action. Return signals would go up from leaves to the root of the BT and, as a result of this, failures and warnings would be located and associated with one branch of the BT (depending on the decision nodes). Therefore, how these failures and warnings would be fixed or reported in design time, would depend on how the tool works.

Nevertheless, the easier way for solving a warning during runtime is to remove the whole branch whilst failures must directly invalidate the association between the entity and the BT.

Once all this infrastructure is working, is easy to use it as a sanity check for all the runtime retrieved BTs for our *query nodes*. That avoids blindly try to run a BT that will fail in the long run because the entity is unable to execute some of the primitive actions. The next section will described a detailed example of the whole process.

5 Example

Let us imagine a shooter game in which a soldier watches over the approach roads of a bunker, patrolling the area and killing the enemies (players) without being seen whenever it is possible. The behaviour executed by this NPC can be the one shown in Figure 4, that represents the BT corresponding to the goal `Guard Area`. This BT has two branches, one to patrol and another to kill the enemy. Due to there are several ways to kill an enemy, and the chosen way will depend on the virtual environment, the NPC type and its parameters, the `Guard Area Behaviour` BT has a *query node* to choose the attack behaviour.

On the other, Figure 5 shows the `Patrol Soldier` entity type made up by components with its default attributes. During the game, an entity of the `Patrol Soldier` type, among others, will carry out the `Guard Area` BT (Figure 4) so, as we will see, the attack behaviour will be chosen accordingly.

The execution context of the `Guard Area` BT is composed of three variables: `?this`, `?world` and `?target`. `?this` references the NPC executing the BT and its attributes describe its properties. `?world` references the virtual environment in which the game takes place. `?target` is an input variable for the Attack behaviour and references the entity targeted for this behaviour.

To execute this behaviour, the node `Guard Area` is executed. Being a *dynaymic priority list*, it will try to execute the first of its children and, if it is not possible, it will pass to the next one. The first child of the `Guard Area` node is a sequence, so it tries to execute all of its children, one after the other, beginning with `Is Entity Seen`. Now suppose that `Is Entity Seen` fails (there are no visible entities of type PLAYER). This makes the sequence to fail and the next behaviour in the priority list, `Patrol`, is then executed. As the `Guard Area` behaviour is a *dynamic* priority list, it will keep trying to execute the first child (the sequence) in the subsequent cycles.

While executing `Patrol`, let us suppose that a `PLAYER` is seen by the NPC. The `Patrol` behaviour is interrupted to launch again the Sequence. It tries to execute again the behaviour `Is Entity Seen`, succeeding this time. The entity detected by `Is Entity Seen` is stored in the attribute `?this.target` and the next behaviour in the sequence is executed. The next behaviour is a query, so it has to be solved to a BT before it can be executed.

The attributes for this query are:

1. **Header**: the name of the query behaviour (`Attack (?target)"`) and a description ("Query behaviour Attack").
2. **Domain**: the retrieved behaviours should belong to the class `Attack` or to any of its children.
3. **Parameters**: the retrieved behaviour has to have an input parameter, `target`, which should be applicable to an entity of class PLAYER.
4. **Descriptors**: this attribute lists the game state descriptors that are considered relevant for the query:
   - `?target.distance`: the distance to the target entity.
   - `?this.personality`: the personality attribute of the entity executing the behaviour.
   - `?this.aggressive` and `?this.defensive`: the aggressiveness and defensiveness levels of the entity.
   - `?this.health`: the health of the entity.
   - `?this.underAttack`: measures the attack received by the entity, being `Low` when it is not being attacked and HIGH when being attacked by several entities in the close range.
   - `?world.time`: the current time in the simulation environment (NIGHT or DAY).
5. **Weights**: the similarity section of the query refers to the importance of the descriptors for this query. The distance, personality of the entity and the fact of being attacked (the `underAttack` condition) are very important. The `health` and the current `time` are important. The `aggressiveness` and `defensiveness` are taken into account, but they are not as important as the rest.
6. **Requery**: the query has to be repeated when there is a significative change in the time, health or `underAttack` descriptors.
7. **Cases retrieved**: the query will retrieve all the cases in the case base.

To retrieve a Behaviour Tree we have to compare the query with all the cases in the case base. First we filter the case base using the *Goal* attribute, keeping only the cases that belong to the *Goal* class or any of its subclasses. In the next step we also take away all the cases with parameters that are not compatible with the ones in the query. Finally, the values of the descriptors of the cases are compared with the values of the relevant descriptors of the game state. Using the weights, a similarity value is obtained for each case.

For instance, let us imagine a night situation in which the NPC is close enough to a player to detect it but not so close, thus the player has not seen and attacked it yet. After filtering the case base the query process retrieves the set of cases shown in Table 1 (the ones for the *Attack* goal). Then, every case has to be compared with the query. The values of the relevant descriptors of the query are retrieved from the game state, and compared to the corresponding descriptors in the cases. Table 2 shows the values of the relevant descriptors for our example query and the results of calculations of the similarity values for each case and the query. As it is shown in the table, stealth behaviours are predominant over the rest because of the night situation, the personality of the NPC and due to the fact that the NPC is not under attack. `Long`

Descriptor	Game State
target.distance	HIGH
this.personality	STEALTHY
this.aggressive	HIGH
this.defensive	MEDIUM
this.health	MEDIUM
this.underAttack	LOW
world.time	NIGHT

(a) Game State

Case	Similarity
C_1	0,90
C_2	**1,00**
C_3	0,42
C_4	0,60
C_5	0,39

(b) Similarity

Table 2: Game State and Similarity values

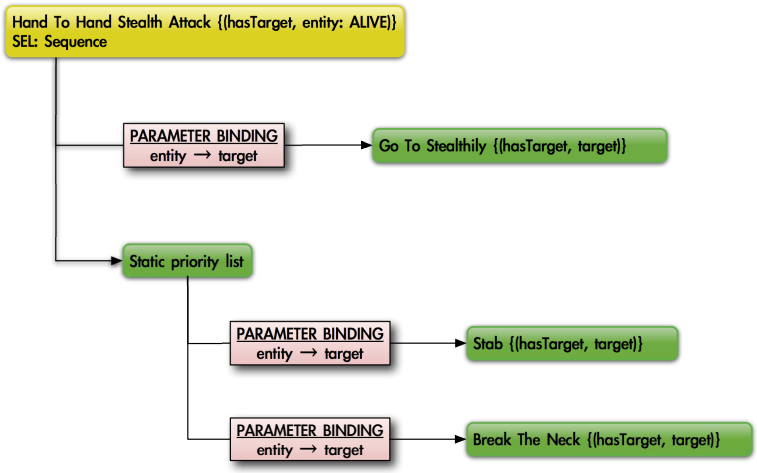
`Range Stealth Attack` has better score than `Hand To Hand Stealth Attack` just because of the distance between the NPC and the player.

Once the set of cases have been retrieved ordered by its similarity, the query process must return the most similar behaviour to the query but, at the same time, the NPC must be able to carry out this behaviour. There should be taken into account that the behaviours stored in the case base may not be suitable for every entity. Different entity types will have different abilities and even, entities of the same type could have different parameter values (for example, strength).

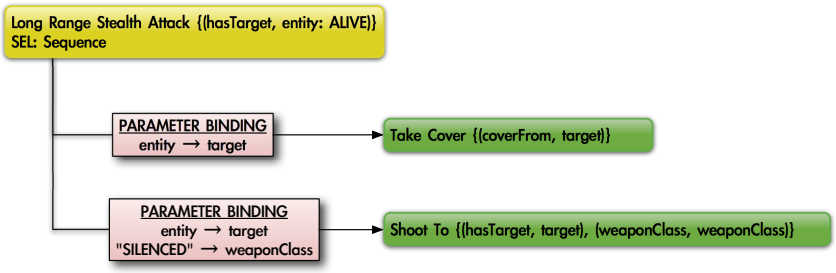
Here is where the *reflective components*, described in section 4, become useful. When the first part of the query process ends up with a list of BTs ordered by its similarity with the query, the query process iterates over them looking for the first one that may be executed by the actual entity. The query process will finally return the behaviour most similar to the query that can be carried out by the NPC.

In our example, the query process has to check which of the retrieved BTs can be executed by the `Patrol Soldier` entity, whose components are listed in Figure 5. We will reduce our explanation just to the two most similar BTs retrieved from the query, `Long Range Stealth Attack` and `Hand To Hand Stealth Attack`, which appear in Figure 6. These BTs need special skills to be carried out so the query process must assure, by means of our *reflecting components*, the NPC will be able to execute these retrieved BTs. When validating `Long Range Stealth Attack` prior to its execution, the system detects that the `Shoot To` action cannot be carried out by the NPC. Although a `Patrol Soldier` has a `ShootTo` component that allows long range attacks with firearms and a `Patrol Soldier` has a rifle and a gun, it does not have a silencer thus the action will not be successfuland consequently this BT is rejected.

Then is the turn of the `Hand To Hand Stealth Attack` BT. In the same way, when validating `Hand To Hand Stealth Attack` prior to its execution, the system detects that the `Stab` action cannot be carried out by the NPC. In this case, the failure returned by the `Stab` action is because the NPC does not have a sharp arm like a knife. The failure is propagated to the sequence node, however, in this case, the failure is not propagated further on because the static priority list node has another valid choice to execute: the `Break The Neck` action.



(a) Hand To Hand Stealth Attack



(b) Long Range Stealth Attack

Fig. 6: BTs of the example

Consequently, the `Hand To Hand Stealth Attack` BT is the behaviour returned due to the fact that it is the most similar behaviour to the query that *can* be executed by the NPC. Once the retrieval process has ended, the execution of the original BT continues in the query node of the original BT and it executes the recovered `Hand To Hand Stealth Attack` BT in a transparent way.

6 Conclusions

BTs are a great tool for design AI game behaviour, because they have an easy graphical representation and promote reuse of complete or partial BTs based on their hierarchical nature. Unfortunately, they intrinsically include some programming concepts that provide them with the expressive power of a general purpose program-

ming language, make them difficult to understand for non-technical designers. As a consequence, BTs are mainly used by *programmers*, who *draw* the behaviours instead of just writting down in some concrete programming language. Designers are, usually, in charge of very high-level BTs, with just a few nodes that are easy to create and debug. They are built putting togheter more complex BTs created by programmers, so designers must have an easy access to the *library of BTs* where all the BTs created for the game team are stored.

This can be, in fact, quite complex. At the end of the game developing cicle, the team can have produced a quite high ammount of BTs, where designers (and also programmers) must dive into in order to look for concrete behaviours while creating new ones. Some kind of *automatic search* is welcome in the BT library for alleviate the time spent while looking for BTs. In this chapter we have presented a tool for BT design that includes such a feature, using CBR techniques for retrieve the more adecuate BTs [5].

On the other hand, as was stated in this chapter, during game production AI designers create *Behaviour Trees* mixing the basic behaviours with aggregation in Behaviour Trees. *At the same time, developers* create new basic behaviours depending on the ongoing necessities (the `Stealth attack` of our example would be one of them). As a result, designers will have more basic behaviours to play with at the end of the production, and the last created BTs will be richer than the first ones.

The ad hoc solution for this consistent problem is to revise the older BTs for detecting if they could be improved using the more recent basic behaviours created by the development team. Unfortunately, this revision effort needs a lot of time and should be performed during all the game production timeline.

Using our *query nodes* [6], on the contrary, old BTs are automatically benefited from new behaviours if they are correctly stored and annotated in the case base. The example has shown that, when using our technique in the `Attack` node, no revision is needed if a new `Stealth Attack` behaviour is developed.

The main advantage of our proposal is that the number of *basic behaviours* can grow throughout the game development and, even so, be quite sure that they will be used in older complex behaviours. Having this confidence when using static BTs requires a manual revision of the previous developed BTs, something only affordable if the number of added behaviours is kept low. Consequently, our proposal provides a better scalability for the *growth of basic behaviours*.

As a welcome secondary effect, and due to the fact that the query nodes take into account all the basic behaviours in the case base, BTs using them could be provide richer behaviours with no design effort. The manual alternative would require the substitution of our *query node* with a priority list (as in the example) with all the available basic behaviours. Again, this becomes impractical, demonstrating that *query nodes* provides a better scalability also *in the number of basic behaviours considered at run-time*.

Unfortunately, all these advantages do not come for free. The cost for this saving is, obviously, categorizing each new basic behaviour in order for the *query node* to recover it in the correct moments. Behaviour and entities ontologies (the vocabu-

lary for describing our cases) must also be created, although they could be reused between projects (after all, reuse is one of the goals of ontologies).

At run-time, our *query node* will spend more time *the first time* for extracting the appropriate basic behaviour if comparing with a priority list. But, due to the *re-query* attribute in the query node, we avoid spending time every AI cycle to change the first election, something that priority lists do not do. On the other hand, debug behaviours using our *query nodes* will be a bit more complex due to the new uncertainty ingredient added to the behaviour selection. This problem can, in fact, be seen as an advantage, because some *emergent behaviour* usually is considered to provide game variability.

References

1. Atkin, M.S., King, G.W., Westbrook, D.L., Heeringa, B., Cohen, P.R.: Hierarchical agent control: a framework for defining agent behavior. In: Agents, pp. 425–432 (2001)
2. Buchanan, W.: Game Programming Gems 5, chap. A Generic Component Library. Charles River Media (2005)
3. Crytek: CryENGINE 3 SDK, Sandbox Editor (2010). Http://mycryengine.com
4. Esmurdoc, C.: Head games: Double fine's psychonautic break. Game Developer Magazine **12**(7), 30–38 (2005)
5. Flórez-Puga, G., Díaz-Agudo, B., González-Calero, P.A.: Experience-Based Design of Behaviors in Videogames. In: K.D. Althoff, R. Bergmann, M. Minor, A. Hanft (eds.) ECCBR, *Lecture Notes in Computer Science*, vol. 5239, pp. 180–194. Springer (2008)
6. Flórez-Puga, G., Gómez-Martín, M.A., Díaz-Agudo, B., González-Calero, P.A.: Dynamic Expansion of Behaviour Trees. In: C. Darken, M. Mateas (eds.) AIIDE. The AAAI Press (2008)
7. Games, E.: Unreal Development Kit (2009). Http://www.udk.com/
8. Garcés, S.: AI Game Programming Wisdom III, chap. Flexible Object-Composition Architecture. Charles River Media (2006)
9. Hocking, C.: Ubisoft Montreal's Far Cry 2 Postmortem. Game Developer Magazine pp. 30–38 (2009)
10. Ierusalimschy, R.: Programming in Lua, second edn. Lua.org (2006)
11. Isla, D.: Handling Complexity in the Halo 2 AI. In: Game Developers Conference (2005)
12. Isla, D.: Halo 3 - building a better battle. In: Game Developers Conference (2008)
13. Krajewski, J.: Creating all humans: A data-driven AI framework for open game worlds. Gamasutra (2009)
14. Lakos, J.: Large Scale C++ Software Design. Addison Wesley (1996)
15. Llansó, D., Gómez-Martín, M.A., González-Calero, P.A.: Self-validated behaviour trees through reflective components. In: Proceedings, The Fifth AAAI Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 76–81. AAAI Press (2009). URL `http://www.aaai.org/Conferences/AIIDE/aiide09.php`
16. Millington, I., Funge, J.: Artificial Intelligence for Games, second edn. Morgan Kaufmann (2009)
17. Rabin, S. (ed.): AI Game Programming Wisdom 3. Charles River Media (2006)
18. Rabin, S. (ed.): AI Game Programming Wisdom 4. Charles River Media (2008)
19. Rene, B.: Game Programming Gems 5, chap. Component Based Object Management. Charles River Media (2005)
20. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Professional (1997)
21. Tai, K.C.: The tree-to-tree correction problem. J. ACM **26**(3), 422–433 (1979). DOI http://doi.acm.org/10.1145/322139.322143

22. Ubisoft Montreal Studios: Far Cry 2 (2008)
23. Wang, J.T.L., Shapiro, B.A., Shasha, D., Zhang, K., Currey, K.M.: An algorithm for finding the largest approximately common substructures of two trees. IEEE Transactions on Pattern Analysis and Machine Intelligence **20**, 889–895 (1998)
24. West, M.: Evolve your hierarchy. Game Developer **13**(3), 51–54 (2006)
25. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. **18**(6), 1245–1262 (1989). DOI http://dx.doi.org/10.1137/0218082

6.2. Semiautomatic edition of behaviours in videogames

Cita completa

Gonzalo Flórez Puga y Belén Díaz Agudo. *Semiautomatic Edition of Behaviours in Videogames.* Proceedings of AI2007, 12th UK Workshop on Case-Based Reasoning. Cambridge. Diciembre, 2007.

Resumen original de la contribución

The edition of intelligent behaviours in games is not an easy task. Amongst other activities, it implies identifying the entities which must behave intelligently, and what kind of behaviours they must show without being too predictable; designing and integrating these new behaviours with the virtual environment, in terms of perception and actuation over the environment, and implementing them. In this paper we present an ongoing work using Case Based Reasoning (CBR) to design intelligent behaviours in videogames. We have developed a graphical editor based on hierarchical state machines that includes a CBR module to retrieve and reuse stored behaviours. The editor and the CBR module are generic and reusable for different games. We have tested our module a soccer simulation environment (SoccerBots) to control the behaviour of the soccer players.

Referencia de citas bibliográficas

Bowling et al. (2006), Girault et al. (1999), Brownlee (2002), Fu y Houlette (2002), Manning et al. (2008), Gómez-Martín et al. (2003), Flórez Puga et al. (2008), Flórez-Puga et al. (2008)

Semiautomatic edition of behaviours in videogames *

Gonzalo Flórez-Puga and Belén Díaz-Agudo
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
email: gflorez@fdi.ucm.es, belend@sip.ucm.es

Abstract

The edition of intelligent behaviours in games is not an easy task. Amongst other activities, it implies identifying the entities which must behave intelligently, and what kind of behaviours they must show without being too predictable; designing and integrating these new behaviours with the virtual environment, in terms of perception and actuation over the environment, and implementing them. In this paper we present an ongoing work using Case Based Reasoning (CBR) to design intelligent behaviours in videogames. We have developed a graphical editor based on hierarchical state machines that includes a CBR module to retrieve and reuse stored behaviours. The editor and the CBR module are generic and reusable for different games. We have tested our module on a soccer simulation environment (SoccerBots) to control the behaviour of the soccer players.

Keyword: Intelligent Agents, Behaviours in Games, State Machines, CBR

1. Introduction

The aim of almost any game is to provide some amusement to the player. This task can be performed in several ways. In the particular case of computer games, besides a good story or spectacular graphics, the game must be a real challenge for the player. An appropriate method for achieving this is by providing the opponents (and the allies) of the player with intelligence [1].

The edition of intelligent behaviours in games or simulation environments is a difficult task. Amongst other activities, it implies identifying the entities which must behave intelligently, and what kind of behaviours they must show (e.g. helping, aggressive, elusive), designing and implementing them, integrating them in the game and testing.

Designing new behaviours could be greatly benefited from two features that are common in most of nowadays videogames. First of all, modularity in behaviours. That means that complex behaviours can be decomposed into simpler ones, that are somehow combined. Second, and related with the former, simpler behaviours tend to recur within complex behaviours of the same game, or even in different games of the same genre. Both features are useful to build new complex behaviours based on simple behaviours as the building blocks that can be reused.

In the ongoing work described in this paper we are developing a graphical behaviour editor that is able to store, index and reuse behaviours previously designed. Our editor (eCo) is generic and applicable to different games, as long as it is configured by a game model file. The underlying technologies of eCo are Hierarchical Finite State Machines (HFSMs) [2] and Case Based Reasoning (CBR).

* Supported by the Spanish Committee of Science & Technology (TIN2006-15140-C03-02)

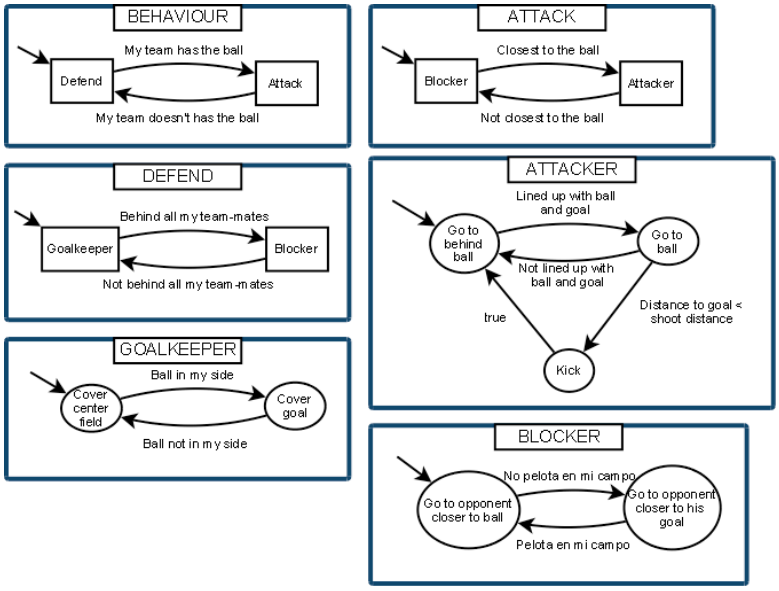


Figure 1. Example of HFSM

HFSMs are appropriate and useful tools to graphically represent behaviours in games, facilitating the modular decomposition of complex behaviours into simpler ones and the reuse of simple behaviours. The eCo behaviour editor provides a graphical interface which allows the user to create or modify behaviours just by "drawing" them. On the other hand, by means of a CBR-based module, the user can make approximate searches against a case base of previously edited behaviours. Both technologies work tightly integrated. Initially, the case base is empty, so all the editing has to be done via the manual editing (graphic) tools. Once there are enough cases in the case base, new behaviours can be constructed by retrieving and adapting the stored ones.

There exist several tools oriented towards the edition of finite state machines. Most of them are general purpose state machine editors that don't allow the use of HFSMs, nor facilitates the reusing. Regarding game editors, most of them are only applicable to one game or game engina (e.g. Valve Hammer Editor). Besides, the vast majority only allow map edition. The few that allow editing the behaviours, are usually script based.

Finally, there exist some tools like BrainFrame and its later version, Simbionic, which are game oriented finite state machine editors. These editors allow the specification of the set of sensors and actuators for any game. There are two crucial differences wtih our approach. First of all, the Simbionic editor doesn't offer any assistance for reusing the behaviours, like the CBR approximate search engine integrated into the eCo editor. And second, to integrate a behaviour edited with the Simbionic editor with a game, it is mandatory to integrate the Simbionic runtime engine with the game.

In section 2 we introduce some general ideas on behaviour edition. In section 3 we present the eCo behaviour editor, and in section 4 we show a small example of application of the editor to a simulation environment: SoccerBots. Section 5 describes the CBR module integrated in the editor. As the editor and the CBR module are reusable through different environments, in section 6 we outline the integration of the editor with different games and simulation environments. Finally, in section 7 and 8, we present related work, future goals and conclusions.

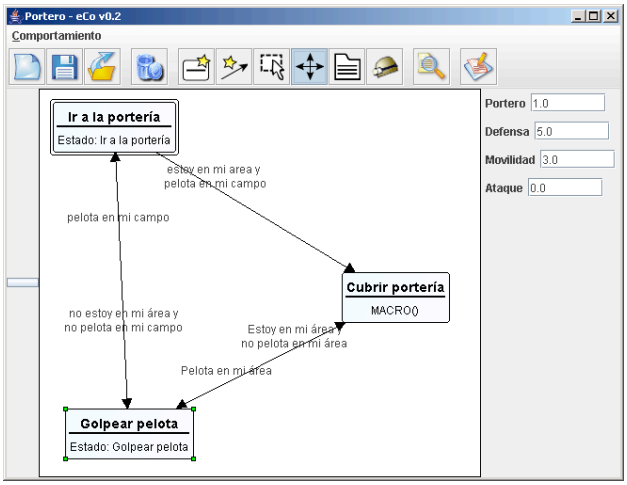


Figure 2. The eCo behaviour editor

2. Behaviour editing in simulation environments

Each behaviour is typically defined by means of a set of actions or reactions performed by an entity, usually in relation with its environment. In a computer game or simulation, each entity gathers information about its environment using a set of *sensors*, which could be compared to the senses of the living beings. Depending on this information, the entity performs certain actions, using a set of *actuators*. In general, is different for each game or simulation environment, although there will be similarities between games of the same genre. For instance, commonly used sensors in a first-person-shooter (FPS) game could be the position, the health or the visibility of other entities. Regarding the actuators, the entity can shoot, look at or go to a place, talk to other entities, among others.

There are two properties, shown by game behaviours, which have been of critical importance for the development of the editor prototype: m*odularity* (complex behaviours are usually composed of simpler behaviours) and *reuse* (simpler behaviours tend to recur in complex behaviours).

Several suitable techniques exist for the representation of behaviours. Due to its expressive power and simplicity, the Finite State Machines (FSMs) is one of the most widespread of them. One of the drawbacks of the FSMs is that they can be very complex when the number of states begins to grow. To prevent this we used Hierarchical Finite State Machines (HFSMs), which are an extension to the classic FSMs. In a HFSM (like the one shown in Figure 1), besides a set of actions, the states can contain a complete HFSM, reducing the overall complexity and favouring its legibility [2]. Each HFSM can be considered as an abstract, modular component, which can be used anywhere in the hierarchy. FSMs have been used successfully in commercial games (e.g. Quake [3]), and in game editing tools (e.g. Simbionic [4]). Representation of behaviours using HFSM is very suitable to be used within a CBR system. Next we describe the basic working aspects of the eCo editor, and an example of its use in the SoccerBots simulation environment. Section 5 describes the CBR system.

3. The eCo Behaviour Editor

The eCo Behaviour Editor (Figure 2) is a graphical editing tool which uses HFSMs to represent behaviours, allowing the user to “draw” the behaviour he wants to get. It also is able to automatically

generate the code to execute the behaviour. The editor is strongly dependant on a CBR module which allows reusing behaviours previously edited. The design of the editor worked towards the achievement of three objectives, namely:

- Easiness of use: the user shouldn't need any technical or architectural knowledge about the game. This is achieved by the use of HFSMs as an intermediate graphic format.
- Applicability: the editor must be able to generate behaviours for different games or simulations, regardless of its genre. To accomplish this goal, the editor can use different configuration files (called game models) and code generators, suitable for each specific game.
- Assistance to users: this goal is met reusing previously edited behaviours, via a CBR module. This module should be able to make approximate retrieving and adaptation of the behaviours.

In section 3.1 we describe the configuration files (game models). Section 3.2 deals with the manual edition of behaviours.

3.1 Defining the game models

A game model is a configuration file that describes some details of a game or a simulation environment. The game models allow the user to use the eCo editor in different games.

Each game model includes the information about sensors and actuators, and a set of descriptors. The sensors and actuators are obtained from the game API. Regarding the descriptors, they are numeric or symbolic attributes that will be used by the CBR module to describe the behaviours and retrieve them from the case base. The descriptors are obtained through the observation of the characteristics of the different behaviours that exist in the domain of the game and must be enough extensive and representative to describe most of the behaviours we can come across for that particular game.

3.2 Editing behaviours, generating code and storing cases

The eCo editor provides a set of editing tools that allows the user to create behaviours from scratch or from previously edited behaviours stored in disk.

Once the behaviour is complete, it is possible to use the code generation tool to generate the source code corresponding to the behaviour. This tool uses the structure of the state machine together with the information in the game model to obtain the source file. As the game model and the source file required are usually different for each game, the code generator will also be unique for each game. The saving tool also allows the user to store the behaviour being edited in the case base for later reusing. We have used XML files to store the case bases. Each case is described by an attribute-value set of descriptors:

- Attributes: numeric and symbolic parameters that describe different properties of the behaviour. The attributes are different for each game, although similar games (e.g. games of the same genre) will share similar attribute sets.
- Description: textual description of the behaviour. It serves a double purpose: the user can use it to fine tune the description given by the numeric and symbolic attributes, and it is shown to the user during the retrieval phase, so he can select the most appropriate case.
- Enclosed behaviours: specifies which behaviours are hierarchically subordinated. This allows the user to retrieve behaviours which include a specific set of sub-behaviours or actuators.

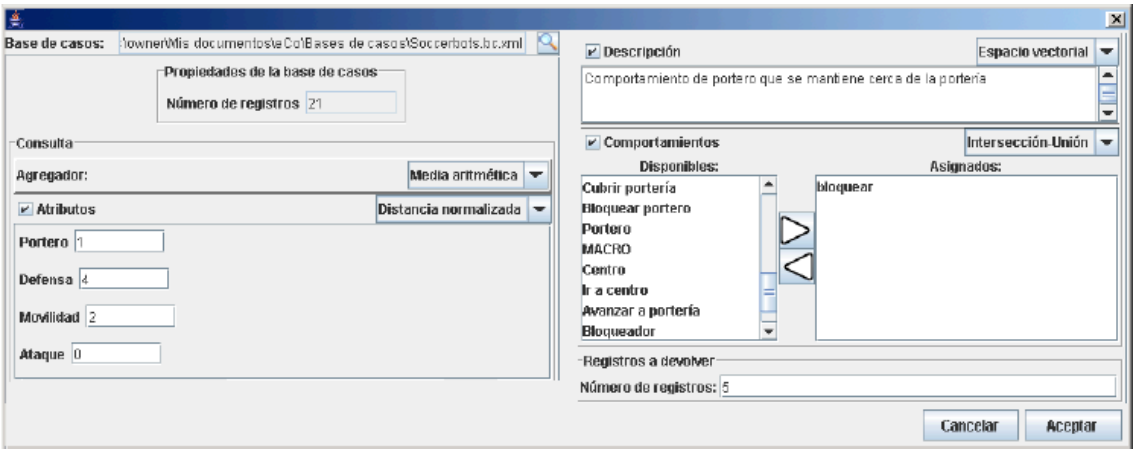


Figure 3. Functionality based queries editor

4. SoccerBots Example

The behaviour editor described in Section 3, and the CBR system that we are describing in section 5, are independent of any specific game. However, for the sake of an easier exposition we are explaining the basic ideas using a simple game. SoccerBots is a simulation environment developed by Tucker Balch, where two teams play in a soccer match. Simulation time, behaviour of robots, colours, size of field, and many other features are configured from a text file. Basically, rules are similar to those from Robocup.

The first step in using eCo to generate behaviours for the SoccerBots environment is to define the game model with the information about sensors, actuators and CBR descriptors of the SoccerBots simulation environment. In the SoccerBots API we can find sensors like `getBallX`, which checks the X, position of the ball, and actuators (i.e. actions that robots can take) like `setSteerHeading(int)`, which changes the direction the robot is facing.

As we stated before, the descriptors are obtained through the observation of the characteristics of the different possible behaviours. We used four numeric parameters to describe SoccerBots behaviours:

- Mobility: ability to move all over the playfield.
- Attack: ability of the robot to play as an attacker.
- Defence: ability of the robot to play as a defender.
- Goalkeeper: ability of the robot to cover the goal.

5. The CBR system

The CBR system takes advantage of the modularity and reuse properties of the behaviours; it assists the user in the reuse of behaviours by allowing her to query a case base. Each case of the case base represents a behaviour. By means of these queries, the user can make an approximate retrieval of behaviours previously edited, which will have similar characteristics. The retrieved behaviours can be reused, modified and combined to get the required behaviours.

Initially, the case base is empty, so all the editing has to be done via the manual editing (graphic) tools. Once there are enough cases in the case base, new behaviours can be constructed by retrieving and adapting the stored ones.

There are two kinds of queries: functionality based queries and structure based queries. In the former, the user provides a set of parameters to specify the desired functionality for the retrieved behaviour. In the latter, a behaviour is retrieved, whose composition of nodes and edges is similar to the one specified by the query. In the current version of the editor, only functionally based retrieval is possible.

5.1 Functionality based retrieval

The most common usage of the CBR system is that the user wants to obtain a behaviour similar to query in terms of its functionality. The functionality is expressed by a set of parameters, which can be any (or all) of the descriptors of the cases presented in section 3.2 (i.e. the attributes, the textual description and the enclosed behaviours). The parameters that form the query are used to describe the behaviour, and are closely related to the game model. The more differences exist between two games, the more different the associated behaviours are and, hence, the parameters used to describe them. The eCo editor provides a query form, showed in Figure 3, for the users to enter the parameters of the query.

To obtain the global similarity value between the cases and the query, the similarity of the numeric and symbolic attributes is aggregated with the similarity due to the textual description of each behaviour. The user can select the most appropriate operator to combine them in the query form. Some examples of operators could be the arithmetic (used in this example) and the geometric mean or the maximum. Figure 3 shows an example query for the SoccerBots environment with the following parameters:

Goalkeeper	1	Attack	2	Description	Goalkeeper behaviour that stays near the goal
Mobility	4	Defence	0	Enclosed behaviours	Block

5.2 Descriptor based similarity

Using the aforementioned form the user can enter the query descriptor values and select the similarity measure used to compare them to the ones in the cases of the case base. To obtain the similarity value between two descriptors, we use the normalized difference value.

In the following table we show an example of the calculus of the similarity measure for the query in Figure 3 and a hypothetical case:

Descriptor	Range	Query	Case	Similarity
Goalkeeper	[0, 1]	1	1	1
Mobility	[0, 5]	4	2	0.6
Attack	[0, 5]	2	3	0.8
Defence	[0, 5]	0	5	0

5.3 Textual based similarity

Description of behaviours by means of a detailed vector of descriptors can be cumbersome and difficult. It would result in excessively long descriptions. Furthermore, it is difficult to identify all them. However is useful to have this descriptors as indexes to filter and select cases.

To make the querying process easier, the user can use a textual description to fine tune the query by including in it characteristics not considered by the attributes. For instance, in the example, the user is requesting a behaviour that stays near the goal. There is no specific descriptor in the game model, as it is

not relevant for most of the behaviours. Instead, the textual description is used. In the current version, we use the vector space model [5] to compute the similarity measure between the text descriptions.

6. Integration with other games

JV^2M [6] is a third-person action game conceived to teach the operation of the Java Virtual Machine (JVM). The game takes place in a space station, which acts as a metaphor of the JVM. The development of JV^2M is currently in progress and the set of sensors and actuators is not defined, so we had to sketch a sensory model, based in the model of the game FarCry.

Neverwinter Nights is a role playing computer game that takes place in the Dungeons & Dragons universe. It includes the Aurora Toolset, which allows scripting the NPC's behaviours. To carry out the integration, we have used RCEI (Remote Controlled Environments Interface), a protocol conceived to communicate a virtual environment with a remote controller application, via ASCII sockets.

Finally, we tested the editor with an AIBO pet, a multipurpose robotic pet. The code controlling the AIBO was built over the library URBI (Universal Real-time Behaviour Interface), which allows controlling the robot remotely, via a wireless connection.

In summary, we have tested the integration in environments with very different nature (a sport simulator, a role playing game, an action game and a real life multipurpose robot) and with different integrating characteristics. For instance, while in JV^2M we define the set of sensors and actuators, it is fixed for the other environments; while Neverwinter Nights is highly event-oriented, the rest of the environments are basically reactive systems.

7. Conclusions and Future Work

In this paper we have described an ongoing work using Case Based Reasoning (CBR) to design intelligent behaviours in videogames. We have developed a graphical editor based on hierarchical state machines that includes a CBR module to retrieve and reuse stored behaviours. One of the main advantages of our approach is that the editor and the CBR module are generic and reusable for different games. We have shown the applicability in a soccer simulator environment (SoccerBots) and we are working in applying our editor to JV2M, a third-person action game conceived to teach the operation of the Java Virtual Machine, that is currently being developed by our research group.

The eCo behaviour editor is easy to use and offers a friendly interface based on a well known technique typically used to represent behaviours: HFSMs. The editor assists the user in the definition of new behaviours through a CBR module that retrieves previously stored behaviours.

We have described the current state of the work but there are many open lines of work. We have finished the graphical editor, defined the structure of the cases and the game models, and we have been working on case representation, storage and similarity based retrieval. Current lines of work are structure based retrieval, more sophisticated similarity measures, automatic reuse of behaviours and learning.

The use of hiecharchical state machines offers many possibilities to reuse and combine pieces of behaviours within other, more complex, ones. We are working on the definition of an ontology on game genres to be able to reuse behaviours, vocabulary, sets of sensors and actuators and even game models between different games of the same genre.

There exist numerous techniques, besides HFSMs, to represent behaviours, like rule based systems, or HTNs, for instance. One of the opened investigation lines is the study of the pros and cons of each one of them and the possibility of combining some of them to create the behaviours

In the current version, the learning of the CBR system is totally user guided: the user indicates which cases must be stored in the case base and also introduce the values for the descriptors. The set of values for each descriptor is a very subjective matter, so it would be a good idea to automatize this process, or, if it is not possible, make the system suggest values using machine learning approaches.

Besides the functionality based queries, presented in this paper, we are working on queries based on the structure of nodes and edges of the state machine that define the behaviour. We are studying about graph similarity measures, like the ones presented in [7], the restrictions they involve (for instance, in the case representation), and the applicable adaptation techniques.

The CBR techniques presented in this paper can also be used in runtime, to retrieve behaviours based in the defined attributes and the state of the game or simulation environment. This is another open research line that is currently being developed [8].

8. References

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

[2] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-Aided Design, 18(6):742–760, June 1999.

[3] Jason Brownlee. Finite State Machines. AI Depot. Available from http://ai-depot.com/FiniteStateMachines/FSM.html (accessed September 18, 2007)

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

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

[6] Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín, and Pedro Antonio González-Calero. Javy: Virtual Environment for Case-Based Teaching of Java Virtual Machine. LNAI Volume 2773, subseries of LNCS, pages 906–913. Springer Berlin / Heidelberg, 2003.

[7] Gonzalo Flórez Puga, María Belén Díaz Agudo, Pedro Antonio González Calero. "Experience Based Design Of Behaviours In Videogames". Advances in Case Based Reasoning 5239. 180–194. Springer. Dresde, 2008.

[8] Gonzalo Flórez Puga, Marco Gómez Martín, Belén Díaz Agudo, Pedro A. González Calero. "Dynamic Expansion of Behaviour Trees". Proceedings of the 4th AIIDE Conference. 36–41. AAAI Press. Stanford, 2008.

6.3. Supporting Sketch-based Retrieval from a Library of Reusable Behaviours

Cita completa

Gonzalo Flórez Puga, Pedro Antonio González Calero, Guillermo Jiménez Díaz y Belén Díaz Agudo. *Supporting sketch-based retrieval from a library of reusable behaviours.* Expert Systems with Applications, Volume 40, Issue 2. Agosto 2012, P. 531–542.

Resumen original de la contribución

Building the behaviour for non-player characters in a game is a complex collaborative task among AI designers and programmers. In this paper we present a visual authoring tool for game designers that supports behaviour reuse. We describe a visual editor, capable of storing, indexing, retrieving and reusing behaviours previously designed by AI programmers.

One of the most notable features of our editor is its capability for *sketch-based retrieval*: searching in a repository for behaviours that are similar to the one the user is drawing, and making suggestions about how to complete it. As this process relies on graph behaviour comparison, in this paper, we describe different algorithms for graph comparison, and demonstrate, through empirical evaluation in a particular test domain, that we can provide structure-based similarity for graphs that preserves behaviour similarity and can be computed at reasonable cost.

Referencia de citas bibliográficas

Hocking (2009), Esmurdoc (2005), Isla (2005), Flórez-Puga et al. (2011), Flórez-Puga et al. (2009), Burkard et al. (2009), Ko et al. (2011), Alexander et al. (2010), Heckel et al. (2009), Heckel et al. (2010), Gerbaud et al. (2008), Smith et al. (2011), Bourg y Seemann (2004), Millington (2006), Jiménez-Díaz y Díaz-Agudo (2007), Glickman (1995), Bunke y Messmer (1994), Riesen y Bunke (2009), Burkard et al. (2009), Jiménez-Díaz et al. (2011), Witten y Frank (2005), Yu y Liu (2003), Manning et al. (2008), Bollmann y Raghavan (1988)

Expert Systems with Applications 40 (2013) 531–542

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Supporting sketch-based retrieval from a library of reusable behaviours ☆

Gonzalo Flórez-Puga *, Pedro A. González-Calero, Guillermo Jiménez-Díaz, Belén Díaz-Agudo

Complutense University of Madrid, 28040 Madrid, Spain

ARTICLE INFO

Keywords:
Graph matching
Behaviour authoring
Game design
FSMs
Non-playing characters
AI authoring tools

ABSTRACT

Building the behaviour for non-player characters in a game is a complex collaborative task among AI designers and programmers. In this paper we present a visual authoring tool for game designers that supports behaviour reuse. We describe a visual editor, capable of storing, indexing, retrieving and reusing behaviours previously designed by AI programmers. One of the most notable features of our editor is its capability for *sketch-based retrieval*: searching in a repository for behaviours that are similar to the one the user is drawing, and making suggestions about how to complete it. As this process relies on graph behaviour comparison, in this paper, we describe different algorithms for graph comparison, and demonstrate, through empirical evaluation in a particular test domain, that we can provide structure-based similarity for graphs that preserves behaviour similarity and can be computed at reasonable cost.

1. Introduction

Building behaviours for non-player characters (NPC) in a game is a collaborative effort among AI designers and programmers. Programmers provide the designers with the *building blocks* for specifying behaviour in the game, as a collection of parameterized systems, entity types and actions those entities may execute. Designers use some combination of state machines, scripting, visual languages and map editors to build complex behaviours by composing the basic pieces the programmers provide. Just to give a hint about the magnitude of the task, developing a AAA game such as Far Cry 2, according to Hocking (2009), required an average of 150 people (including testers) over 43 months, which means, with a conservative assumption of 20% designers, in 30 designers working for three years to create game play contents for a shooter.

Ideally, a detailed design document should serve as the specification contract among designers and programmers: before entering into the production stage, it should be perfectly clear which building blocks the programmers should build and which building blocks the designers should count on to design the game levels. However, in actual development processes, the design of the game usually becomes a moving target, with designers coming up with new requirements for programmers as new mechanics are explored. Furthermore, programmers overwhelmed by their current tasks can feel tempted to let designers use their dubious scripting skills to implement such additions, which later will probably result in the programmer debugging a designer's script during crunch time.

A key problem in this process is that even if a good game designer may not have programming skills, most of the time (s)he is building portions of a software system. A possible solution to this problem is to hire designers who know how to program, which actually some companies do (Double Fine fired the whole level design department in the middle of the development of Psychonauts, and hired fresh college graduates from Computer Science departments to script the levels (Esmurdoc, 2005)). Another approach, also used in industry, is to let designers use visual languages that are supposed to facilitate the process by hiding the formal syntax of the programming language. UnrealKismet, integrated in the Unreal Development Kit game editor, and Flow-Graph Editor, integrated in the Sandbox Editor of CryENGINE 3 SDK, are two such visual scripting tools that let designers model the gameplay of a level without touching a single line of code through some variation of data flow diagrams.

The motivation for the work presented here is a new authoring tool for game designers. The novelty of our approach is to leverage a collection of reusable behaviours. Typically in a large game we can find simple behaviours that are replicated within different complex behaviours. For instance, in a soccer game, *defend* could be a complex behaviour that is composed of two simpler behaviours like *go to the ball* and *clear*; meanwhile *attack* could be made up of *go to the ball*, *dribbling* and *shoot*. However, the actual process of AAA game development, where, as described above, a group of game designers and programmers collaborate over a long period of time to iteratively design a large number of complex behaviours (for instance, Halo 2 had an average of 60 different behaviours arranged in four layers (Isla, 2005)), does not currently rely on

☆ Supported by the Spanish Ministry of Science and Education (TIN2009-13692-C03-03). Funded by Complutense University of Madrid.

* Corresponding author. Tel.: +34 91 394 75 76; fax: +34 91 394 75 47.
E-mail addresses: gflorez@fdi.ucm.es (G. Flórez-Puga), pedro@sip.ucm.es (P.A. González-Calero), gjimenez@fdi.ucm.es (G. Jiménez-Díaz), belend@sip.ucm.es (B. Díaz-Agudo).

0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2012.07.067

behaviour reuse. Without supporting tools and technology, reuse is not an option, and game designers tend to develop new behaviours from scratch, resulting in variations of similar behaviours coexisting in the same game, ignoring the benefits of reuse in terms of quality and scalability.

Through behaviour reuse we make possible scenarios such as these:

1. A designer needs to create a new behaviour that is somehow similar to a behaviour already created. The designer locates the similar behaviour, analyzes it and transforms it to fulfil the new requirements.
2. A designer needs to create a new behaviour that is the combination of two already created behaviours. Again, has to locate, analyze and find the way of combining them to achieve the goal of the new behaviour.

To help designers in the task of building and reusing behaviours we are developing a visual editor capable of storing, indexing, retrieving and reusing previously designed behaviours. Although in this paper we exemplify the approach with behaviours represented as hierarchical finite state machines (HFSMs), the editor can deal with other formalisms typically employed for designing behaviour in videogames, such as finite state machines (FSMs), and behaviour trees (BTs) (Flórez-Puga et al., 2011). We have presented some results on behaviour retrieval with BTs in (Flórez-Puga, Gómez-Martín, Gómez-Martín, Díaz-Agudo, & González-Calero, 2009).

One of the most notable features of our editor is its capability for *sketch-based retrieval*. In the image retrieval domain, sketch-based retrieval consists in finding a complex image using an approximate representation of it (an sketch) as a query. We can translate that idea to the behaviour domain, where a sketch is a partial representation of a behaviour (for instance, a FSM that is missing some edges or where the behaviour of a node has not been specified). In sketch-based retrieval of behaviours we search in a repository for behaviours that are similar to the one the user is drawing, making suggestions about how to complete it. Despite the usefulness of this feature, it poses a difficult problem and an open question that we describe now.

Regardless of the formalism employed – FSMs, HFSMs or BTs–, sketch-based retrieval essentially translates into comparing a graph, the one being sketched, against a collection of graphs representing reusable behaviours. Unfortunately, the problem of assessing similarity between two graphs easily becomes intractable when using methods that take the graph structure into account. Some methods for assessing similarity between two graphs, such as graph edit distance, are based on finding a subgraph isomorphism, which is an NP-complete problem (Bunke & Messmer, 1994). Essentially, it requires enumerating every possible mapping from the nodes and edges of one graph into the other, in order to determine which mapping maximizes similarity. Given the difficulty of finding the best isomorphism between two graphs, we make use of an alternative heuristic approach that finds solutions which are very close to the optimal solution in a particular domain, as will be explained in Section 7.1.

The proposed heuristic is straightforward: given two graphs to be compared, we generate just one mapping, the one maximizing node similarity. Given two sets of nodes and a similarity measure between nodes, the problem of obtaining the node mapping that maximizes similarity is equivalent to the assignment problem, which can be solved by the Hungarian algorithm with a time complexity of $O(n^3)$, n being the number of nodes (Burkard, Dell'Amico, & Martello, 2009).

The open question in sketch-based retrieval of similar behaviours through graph comparison is whether structurally similar graphs actually represent similar behaviours. In order to answer this question, we characterize behaviours in a quantifiable and parametric perspective by using gameplay metrics. Gameplay metrics are data extracted from computer game engines during play. The analysis of these metrics has been used to derive *play-personas*, archetypes that describe the behaviour pattern of a human player (Canossa & Drachen, 2009), which we are using here to characterize a synthetic one.

One of the advantages of our approach is that it can be applied to different behaviour representations, provided that these representations are based in a graph. To use sketch-based retrieval we just have to extract the graph and apply the structural similarity functions to find other similar graphs.

In this paper, we present different algorithms for graph comparison, and demonstrate, through empirical evaluation in a particular domain, that we can provide structure-based similarity for graphs that preserves behaviour similarity and can be computed at reasonable cost.

The rest of the paper runs as follows. The next section presents related work. Section 3 describes our experimental domain, and Section 4 describes our visual authoring tool. Section 5 describes the structure-based similarity functions that we consider and presents the heuristic mapping. Section 6 introduces the experimental set-up while Section 7 discusses the experimental results. Finally, Section 8 presents conclusions and future work.

2. Related work

From a general perspective, this work is related to the field of end-user programming. Game designers build portions of a software system without actually being software developers. Nevertheless, end-user programming is such a broad field that in this section we will concentrate on related work in authoring tools for behaviour in games. See Ko et al. (2011) for a review of the state of the art in end-user software engineering.

Unreal Kismet[1] belongs to the Unreal Development Kit (UDK). Kismet allows non-programmers to script complex gameplay flow in a game level (and, in particular, to create the AI of NPCs). Behaviours created with Kismet can only be used by games developed for the UDK.

The behaviours are specified as *Sequences*: collections of simple *Sequence Objects*. The Sequence Objects have outputs that can be connected to the inputs of other Sequence Objects to form bigger and more complex sequences.

One of the major flaws of Kismet is its scalability. The editor does not allow designers to assign the sequences to an specific entity. Although it is possible to organize hierarchically the sequences using subsequences, that act as black boxes, in the end all the sequences for a level have to be added to the same sheet. Kismet does not offer any assistance to manage collections of sequences. It only allows to search by name inside a sequence.

The *Flow Graph Editor*[2] is a tool integrated in the Sandbox Editor of the CryENGINE 3 Free SDK. It is also a visual-scripting tool that allows to create the behaviours for all entities in the levels created in the Sandbox Editor. The Flow Graph Editor is very similar to Kismet. In this case, behaviours are represented as *Flow Graphs*. The Flow Graphs are composed of nodes and links. The nodes represent the components (i.e. the actions) and entities. Nodes have input and output ports that are connected by links. There is also a special type of component (the *Modules*) that allows to invoke the execution of another graph, permitting the creation of a hierarchy of graphs. The

[1] Kismet: http://www.unrealengine.com/features/kismet/.

[2] Flow Graph Editor: http://freesdk.crydev.net/display/SDKDOC2/Flow+Graph+Editor.

main difference with Kismet is that, in the Flow Graph Editor, each entity can have its own Flow Graph. This favors scalability when the number of game entities increases. The Flow Graph Editor also offers tools to add and remove the behaviours to a collection. There is also a tool to search in the collections but only allows to search for node names and values.

Unity[3] is an authoring tool for creating multi-platform 3D games. The tool consists of a development environment for creating and designing games and an engine for executing them. The current version of the Unity editor (3.5) does not provide any visual tool to aid designers in the creation of intelligent behaviours. They have to program them in any of the scripting languages supported. Nevertheless, there are 3rd party plugins that can be used by designers.

Behave[4] is a plugin for Unity that allows users to graphically design Behaviour Trees for the entities in the game. The use of this editor is very intuitive given a general knowledge about BTs. Although the edition is visual, some scripting is needed to create the code that executes the BTs and to create the basic behaviours contained in the leaf nodes.

In Behave, all created BTs belong to a Collection, which is used to organize the trees and to access them through code. Behave allows the reuse of the BTs stored in the Collections by adding references to them as nodes inside another BT. To find a behaviour, the user has to manually browse all his Collections, because the editor does not include any assistance to search for behaviours.

RAIN{one}[5] and *Playmaker*[6] are also Unity plugins. Both of them are commercial and both of them allow users to graphically design behaviours (RAIN{one} uses BTs while Playmaker uses FSMs). To allow non-programming users to build behaviours they offer a set of predefined actions, but users also have the option to use scripts to create more specific actions tailored to their needs. They do not provide the possibility to create hierarchies of behaviours or allow to search for behaviours.

BehaviorShop (Alexander, Youngblood, Heckel, Hale, & Ketkar, 2010, Heckel, Youngblood, & Hale, 2009, Heckel, Youngblood, & Ketkar, 2010) is a behaviour editor based on the subsumption architecture proposed in Brooks (1986). Behaviours are defined as a set of layers, that are composed of a triggering condition and a corresponding action to run in the form of an `if ... then` rule. Each layer also has a priority. Layers with higher priority can override the lower layers. Behaviours in BehaviorShop can be organized hierarchically: a single layer can be composed of two or more subordinated layers. This editor is not designed to support the management of collections of behaviours.

Apart from these examples, there is a whole range of genre-specific tools to assist in authoring content for a particular type of games, such as serious games for procedural training (Gerbaud, Mollet, Ganier, Arnaldi, & Tisseau, 2008), or even to autonomously generate content as in (Smith et al., 2011) for 2D platformers, but our approach is unique in dealing with libraries of complex behaviours represented in genre-independent formalisms such as FSMs and BTs.

We can see that there are different approaches to the edition of behaviours. Most of them try to facilitate the design task by using different techniques for representation of behaviours (FSMs, BTs, rules, etc.). On the other hand, the use of techniques for reusing previously edited behaviours is not very extended. Most of the tools reviewed do not cover the management of collections of behaviours, and those that do just allow adding and removing behaviours to the collections. Searching in the collections, when it is allowed, is limited to a textual search for the name or the components of a behaviour.

3. Domain description

FSMs are a popular method to model the behaviour of NPCs in videogames (Bourg & Seemann, 2004). FSMs are graphs in which the nodes represent the different states an NPC can be in, while the edges represent the transitions between the states. Nodes are labeled with the actions that are executed when the NPC reaches that state, and edges are labeled with the conditions that control the state changes.

One of the drawbacks of FSMs is that their complexity grows along with the number of states. A possible solution is to use HFSMs (Millington, 2006) to represent the behaviours. HFSMs are an extension of the classic FSMs. Besides the basic actions, a node in a HFSM can be labeled with another state machine (then is called a *composite node*). This way, the overall complexity of the behaviours is reduced, favouring their legibility. Any HFSM can be *flattened*, that is, transformed into an equivalent, non-hierarchical FSM, by following a simple algorithm.

Several reasons drove us to use HFSMs as the manner to represent the behaviours. In the first place, it is a technique widely used in videogame development. Furthermore, for applying sketch-based retrieval we need behaviours that can be represented as graphs, so their structures can be compared. This makes techniques like HFSMs or BTs appropriate to represent them. And finally, we already have a repository of state machines provided by the students of the Knowledge Based Systems course at the Complutense University of Madrid (Jiménez-Díaz & Díaz-Agudo, 2007). Every year, a soccer bots competition is held between teams created by groups of students. We used the teams to create a repository and obtain the experimental results.

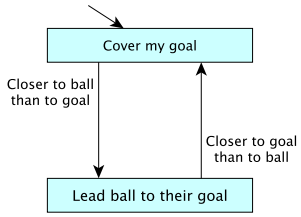
eCo,[7] the visual editor described in this paper (see Section 4) helps the development of NPC behaviours by using HFSMs for SBTournament[8] (Jiménez-Díaz & Díaz-Agudo, 2007), a framework built on top of SoccerBots,[9] a well-known simulation environment developed by Tucker Balch which simulates the dynamics and dimensions of a regulation RoboCup[10] small size robot league game: two teams of five robots compete on a soccer field by pushing and kicking a ball into the opponent's goal.

In order to implement robot behaviours, SBTournament provides users with a set of sensors and actuators, which are actually a superset of those provided by SoccerBots. Actuators are the most simple actions that a robot can execute, while sensors are the pieces of information that a robot can gather from the game world. For example, actuators in SBTournament allow users to kick the ball or set the desired heading and speed for a robot. Likewise, sensors provide information about the ball position or the position of the opponent's goal.

We use sensors and actuators to build the HFSMs that our robots will execute. On one hand, we use sensors to build the conditions for the edges of the HFSMs. On the other hand, we use actuators to build the *basic behaviours*, i.e. the basic building blocks for the robot's behaviour. Basic behaviours are the simplest actions that can be executed in a node of a robot's HFSM. These basic behaviours generally consist of a sequence of calls to different actuators.

Fig. 1 shows the `Forward` behaviour, an example of a simple state machine for a striker. The state machine consists of two nodes. The initial one is labeled with the basic behaviour `Cover`

[3] Unity: http://unity3d.com/.
[4] Behave: http://eej.dk/angryant/behave/.
[5] RAIN{one}: http://rivaltheory.com/product.
[6] Playmaker: http://hutonggames.com/index.html.
[7] eCo: http://gaia.fdi.ucm.es/research/eco-behaviour-editor.
[8] SBTournament: http://gaia.fdi.ucm.es/research/sbtournament.
[9] SoccerBots: http://www.cs.cmu.edu/trb/TeamBots/Domains/SoccerBots.
[10] Robocup: http://www.robocup.org/.



Basic Behaviours

Cover goal

```
Vec2 dest =
   new Vec2(myRobotAPI.getOurGoal());
dest.add(myRobotAPI.getBall());
myRobotAPI.setSteerHeading(dest.t);
myRobotAPI.setSpeed(1.0);
```

Lead ball to goal

```
myRobotAPI.alignedToBallandGoal();
myRobotAPI.kick();
```

Fig. 1. "Forward" behaviour.

`my goal`, which uses the actuators `setSteerHeading` and `setSpeed` to go to a position in the path between the ball and the goal. The edge labeled `Closer to ball than to goal` checks the sensors `Distance to ball` and `Distance to goal` and compares their magnitude until the former is less than the latter. At that moment the state changes to `Lead ball to their goal`, which tries to direct the ball towards the opponent's goal. This basic behaviour consists of two actuators: `Align with ball and goal` and `kick`. There is another edge, `Closer to goal than to ball`, that will be activated when the sensor `Distance to ball` has a value greater that `Distance to goal`.

4. eCo Behaviour Editor

eCo, the behaviour editor we are presenting in this paper allows users to "draw" HFSMs to specify the behaviour of SBTournament's robots and teams. Although we exemplify its use with SBTournament, the editor is easily configurable to be used with any other videogame or simulation environment.

As the user draws the behaviours, the partially completed HFSM is used as a sketch to retrieve previously created behaviours. The users can reuse the retrieved behaviours to complete the one being edited. Once a HFSM is finished it can be exported and executed in SBTournament. Additionally, the HFSM is also stored in a repository in order to be reused in future HFSMs.

We distinguish between two perspectives for our editor: the Player Edition Perspective and the Team Edition Perspective.

Fig. 2 shows the Player Edition Perspective. This perspective allows users to design individual players. The area in the middle is a canvas where users can draw the HFSMs that represent robot behaviours. As stated in the previous section, the nodes can contain basic behaviours (for instance, the node `Cover Goal`) or another state machine (for instance, the nodes `Go To Goal` and `Kick Ball` in Fig. 2).

Under the drawing canvas there is a code editor where users can create and modify basic behaviours. The code editor is divided into three sections. On the leftmost panel there is a list that shows all the basic behaviours created. The user can create a new behaviour by adding it to this list, or edit any behaviour that it contains. The lists on the right show all the available sensors and actuators that will be used by the basic behaviours. The user can drag them to the text editor to incorporate them to the current behaviour. Finally, in the middle there is a text editor where the user can modify the actions associated with the basic behaviour. In this editor the user adds the calls to the actuators that will be executed by the basic behaviour. The user can also write any arbitrary snippet of Java code in the editor that allows the behaviour to access the sensor data or execute any other operation.

On the left of the canvas there is a suggestions panel. While building a behaviour, this panel shows the HFSMs representing other behaviours stored in the repository that are similar to the one being edited. The idea here is to use the behaviour being edited as a sketch of a complete behaviour. The sketch, an unfinished version of the desired behaviour, is used to retrieve similar behaviours that are presented to the user.

When the user selects a suggested behaviour, the editor shows some statistics about this behaviour in the table below, like the number of matches played, the number of goals scored, the average amount of time playing as goalkeeper and as forward in each match, the average amount of distance covered, etc. The statistics are gathered by making the teams of the repository play several matches versus a predefined set of *trainer teams*.

When presented with a suggestion, users can choose to ignore it and continue editing the behaviour manually, accept it and discard the current behaviour, or combine it with the current. The inner workings of the sketch-based retrieval procedure will be explained in the following sections.

The adaptation process is not automatized, but the system offers some assistance so users can do the adaptation manually. The information regarding the gameplay of the behaviours suggested can be employed by the users to adjust the behaviour being built. For instance, if the users want to develop a behaviour that has an offensive gameplay they could compare their behaviour with the behaviours suggested and find a more offensive one (with more goals scored or more time playing as forward) but still similar, and use it as a model to modify its configuration.

The Team Edition Perspective is shown in Fig. 3. It is designed to configure teams using the HFSMs created in the Player Edition Perspective. In the central area, users can assign a previously created HFSM to each robot within the team. The available HFSMs are displayed in the rightmost area. The lower panel is populated with previously designed teams that are similar to the user's team, and with some statistics regarding the gameplay of the teams suggested (like average of goals for and against, rate of wins and losses or average time the players spend on each field). Using the information provided in the lower panel, users can complete their teams by adding the suggested behaviours to their formation, or finding out which teams will play in a similar fashion.

In addition to the statistical information, teams in the repository are ranked using an algorithm based on the Elo rating system (Glickman, 1995). Elo was originally a chess rating system, which nowadays has became a popular rating system for multiplayer competitions in several computer games. In particular, Elo has been adapted to team sports. Team ranking – on the lower right corner of the Team Edition perspective (see Fig. 3) – shows the position of the teams that are similar to the one they are editing. This way the users can get an approximate idea of how well their team will perform.

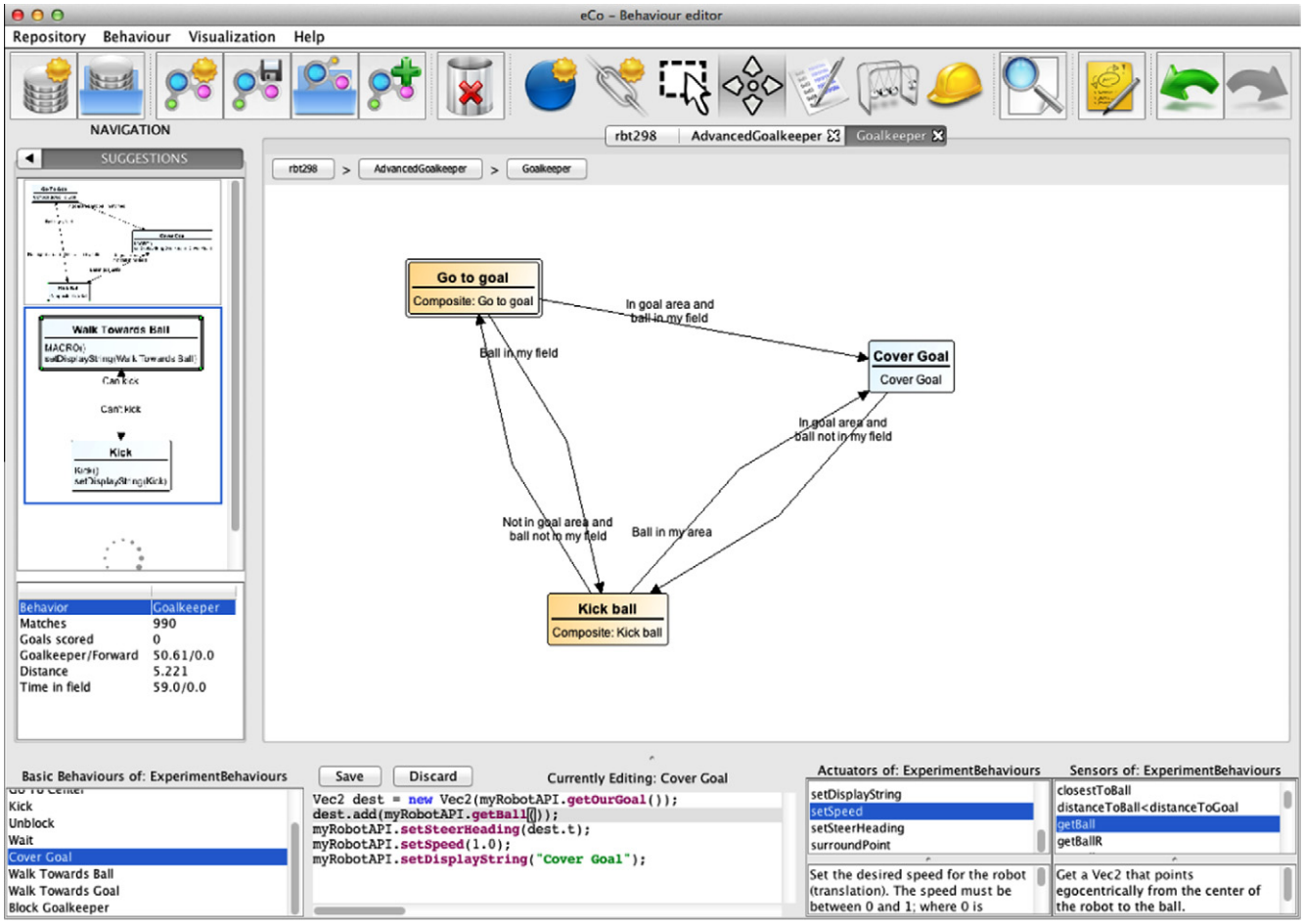


Fig. 2. Player edition perspective.

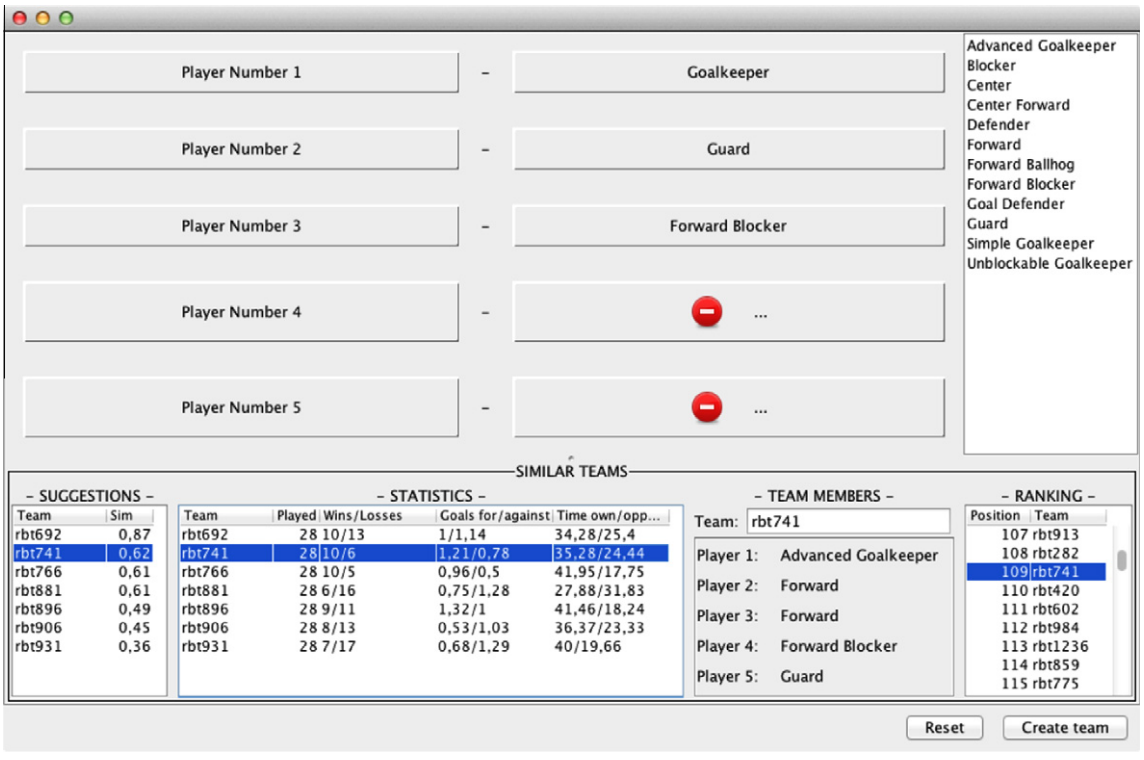
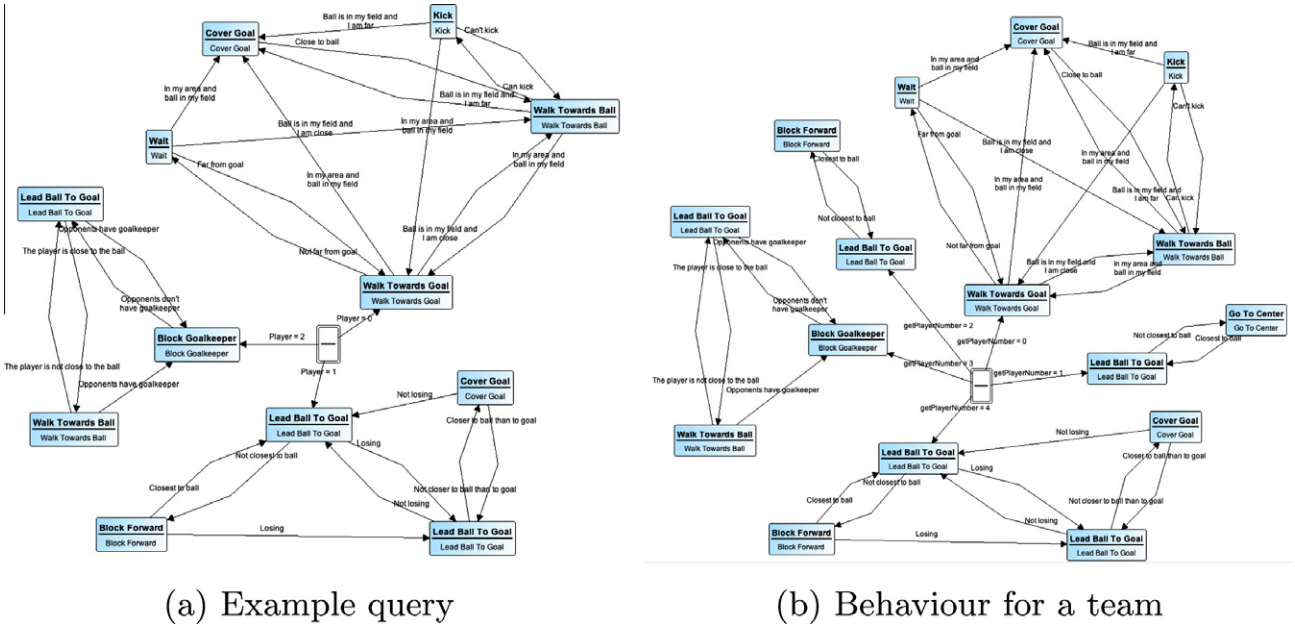


Fig. 3. Team edition perspective.



(a) Example query (b) Behaviour for a team

Fig. 4. Examples of a query and a candidate team for that query.

The created team is another HFSM that has one composite node for each of the behaviours of the players. Fig. 4 shows the state machines of the query in the upper side of Figs. 3 (Fig. 4a) and the behaviour selected in the lower side (Fig. 4). In the figures we have flattened the HFSMs to show their contents.

From the point of view of *eCo*, a team is managed like a single NPC that is composed of five parts, each part being one of the team's players. The task of the team's state machine is to control and to coordinate each of those parts.

The most important feature of the editor is that it uses the sketches of the behaviours being drawn by the user to retrieve similar behaviours from the repository. The retrieved behaviours are presented to the user, who can use them to transform the sketch into a complete behaviour.

For complete behaviours like the ones in the repository, we can make them play and gather statistics about their gameplay to see if they are similar. But in the case of a sketch that is not possible, because the behaviour is not finished yet. Instead, we have to rely on another similarity metric that allows us to compare behaviours and predict which of them behave similarly.

An HFSM can be seen as a graph with directed edges and labels on edges and nodes. Hence, we can compare two HFSMs by comparing their underlying graphs, using the structural similarity functions that exist for graphs. It is important to note that, in fact, this technique can be applied to any other formalism of representation of behaviours, as long as they can be represented as graphs. For instance, if we store the behaviours as BTs, we can use the same similarity functions to compare them.

The next section describes the graph similarity functions we used in the sketch-based retrieval process.

5. Similarity functions for behaviours

To perform the retrieval of the HFSMs we have to compare the query, a sketch of a behaviour in the form of a HFSM, with all the HFSMs in the repository, and return the most similar ones. To perform this comparison, we first flatten the HFSMs into their equivalent FSMs. Then we calculate their similarity and return the HFSMs corresponding to the most similar FSMs. This transformation simplifies the calculation of the similarity value, as we will see.

An FSM is a directed labeled graph defined as a tuple of four elements:

$$G = \langle N, E, \mu, \nu \rangle,$$

where

N is the set of nodes,
$E \subseteq \{N \times N\}$is the set of edges,
μ: $N \to L_N$ is the node labeling function, which assigns a label to each node in N. The node labels are actions built using the actuators from the game.
L_N is the set of labels for the nodes,
ν: $E \to L_E$ is the edge labeling function, which assigns a label to each edge in E. The edge labels are built using the sensors.
L_E is the set of labels for the edges.

To compare the structure of two graphs we are using the edit distance for graphs (Bunke & Messmer, 1994), which is a generalization of the string edit distance. To calculate the edit distance we must define a set of elementary *edit operations*. An edit operation, *op*, is an operation that modifies the graph, like removing an edge or changing the label of a node. We will consider the following set of edit operations: adding a node (A), deleting a node (D) and changing the label of a node (C), and adding an edge (A′), deleting an edge (D′) and modifying its label (C′).

Each edit operation has a cost $c(op) \geqslant 0$. Generally, a constant cost is assigned to each edit operation. To improve the similarity model, in our approach we consider the cost of modifying the label belonging to a node or to an edge as a function concerning the similarity between source and target labels. This expresses more accurately the intuitive idea that changing one label for another is cheaper in cost if the labels are more similar.

Hence, to completely define the modifying operations we need a node similarity function ($sim_N(\mu(n),\mu(n'))$) and an edge similarity function ($sim_E(\nu(e),\nu(e'))$) that allows us to compute the similarity value between two labels. The cost of a modifying operation will be inversely proportional to the similarity of the labels. If the nodes or edges are very similar, the cost of replacing one with the other

should be small (in particular, if the similarity is 1 the cost of the edit operation should be 0).

If we used HFSMs, the cost functions would not be so straightforward. The problem is that, in a HFSM a node can contain another state machine. Hence, the operation of adding or removing one of such nodes (A and D operations) should have a cost that is proportional to the size of the subordinated HFSM. The *C* operation is even more complex, because we can find situations where we want to transform a composite node into a simple one or into a different composite one. If we want to operate on nodes that are at different levels in the hierarchy of a HFSM we would need to define a new operation to change their hierarchical level first. On the other hand, if we flatten the HFSMs, all nodes are at the same level.

Furthermore, using HFSMs the same behaviour can be represented in different ways only changing how nodes are grouped in the different composite nodes. By flattening the state machines, those equivalent HFSMs are transformed into the same FSM, simplifying the similarity process.

A sequence of edit operations is called an *edit sequence* (*es*). The total cost of an edit sequence is the sum of the costs of its operations. Given any two graphs, $G = \langle N, E, \mu, \nu \rangle$ and $G' = \langle N', E', \mu', \nu' \rangle$, there is at least one edit sequence that transforms G into G'. The trivial case would be deleting all nodes and edges from G and adding all nodes and edges from G'. Frequently there is more than one possible edit sequence. The edit distance is the minimum cost among all edit sequences that transform G into G':

$$dist(G, G') = min\{c(es) | es \text{ is an edit sequence that transforms } G \text{ into } G'\}$$

The edit distance is a measure of dissimilarity, but it can be easily converted into a similarity measure:

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

The edit distance between two graphs, G and G', can be obtained in a pretty straightforward manner, by generating all the possible edit sequences that transform G into G', calculating their cost and keeping the minimum one.

The issue with this approach is its complexity: the number of edit sequences grows factorially with the number of nodes. For queries with graphs in the range of 15–25 nodes, like the ones we are using, this method is unpracticable. To reduce the complexity of the search we opted to use the heuristic method proposed in (Riesen & Bunke, 2009). Instead of searching the whole solution space, the idea is to generate only one edit sequence that is close enough to the best one. The sequence generated is the one that minimizes the cost of the operations related with the nodes. We can find this sequence using the Hungarian algorithm (Burkard et al., 2009), that has a complexity in time that is in $O(N^3)$.

The Hungarian algorithm uses as input the cost of the operations on nodes. Depending on the information contributed by the cost function, the results of the heuristic similarity will be more or less accurate. We propose two heuristic functions that differ in the cost function employed:

- Identity (f_{id}): in this case, the cost of modifying a node is the *identity* function, that is, 0 if the target node has the same label and 1 if the labels are different.
- Identity with edges (f_{edge}): this function compares two nodes using the identity function f_{id}, and adds to the result the similarity of the edges entering the nodes and leaving them.

To consider the initial node in the cost function, we add an extra cost value if one of the nodes is the initial and the other is not.

To compute the cost of the operation of modifying the label of an edge and, hence, to compute the cost of modifying a node using the function f_{edge}. we also need a similarity function for the labels of the edges. We used the Jaccard coefficient:

$$sim_E(l, l') = \frac{|sensors(l) \cap sensors(l')|}{|sensors(l) \cup sensors(l')|}$$

where *sensors* returns the set of sensors used in the label of an edge.

6. Experiment setup

One of the features of the editor tool described in this paper is the sketch-based retrieval of similar HFSMs from a repository. The challenge in this retrieval feature lies in finding HFSMs that *behave* similarly to a given one without the necessity of executing them, but by comparing their structure. Previous section described a similarity function that copes with the problem of comparing HFSM structures and current section will detail the experimental study run in order to validate that the proposed function can be employed to retrieve HFSMs with similar behaviour.

Sketch-based retrieval generates a list of HFSMs that are structurally similar to a given one. We want to determine if two HFSMs that are structurally similar also have a similar behaviour. To do that, we compare two lists: one generated using the proposed structural similarity functions, and the other generated with a similarity method that measures whether two HFSMs behave similarly.

The first step will be to create a test set of HFSMs that represent different behaviours for SoccerBots teams. Later, we need the reference measure, a golden standard to validate which elements in our test set share a similar behaviour. Finally, we will evaluate our heuristics by comparing the results obtained by both similarity measures. The details of these procedures are described in the following sections.

6.1. Test set generation

To test the similarity functions we built a set of 700 teams. Each element in this set is a HFSM that controls the behaviour of a standard SoccerBots team of five robots, like the example shown in Fig. 4b. The total number of nodes of the HFSMs in the test set ranges from 14 to 35. The HFSMs are composed of five smaller HFSMs, that control the behaviour of each robot. We will refer to each of the robots' HFSM as a *role*. In our pool of available roles there are three *Goalkeepers* and nine roles for other positions (e.g. we have one center, three kinds of forwards or two defenders). The teams are composed of five roles randomly selected from the aforementioned while applying two simple restrictions: each team must have exactly one goalkeeper and two different teams cannot have the same lineup (the same roles for all robots).

6.2. Functional similarity and game metrics dataset

We need a reference measure that, given two HFSMs, verifies whether they actually behave in a similar fashion. To obtain this function we simply "let the teams play" and gather some statistics about their gameplay. We use the values of those statistics to compare the behaviour of both teams.

We extracted the data using the tools included in SBTournament, which allow users to generate SoccerBots tournaments and traces of robot behaviour. SBTournament periodically extracts the position, direction and velocity of every robot and the ball during a match. The kick actions and goals are asynchronously extracted. SBTournament uses these traces to generate CSV files about every robot, team and match played (Jiménez-Dłaz, Menéndez, Camacho, & González-Calero, 2011).

For each match, m, and for each team playing in the match, t, we have a vector of 21 attributes related to the gathered statistics

$$\vec{a_{mt}} = (a_{0mt}, \ldots, a_{20mt})$$

where a_{imt} is the value of attribute a_i for match m and team t. The kind of attributes extracted are, for instance, the goals for and against, the time in different regions of the field or the average distance to each of the goals.

We can group the vectors by team, computing the average values of each attribute for the individuals on the same team:

$$\vec{a_t} = (a_{0t}, \ldots, a_{20t})$$

where each a_{jt} is the average value of attribute a_j for all the matches played by team t.

The reference similarity measure employed to evaluate the proposed heuristic is built using those statistics. We can compute a local similarity measure for each attribute of each pair of teams. Thus, if we have two teams, t_i and t_j, we obtain a vector of local similarities:

$$sim_L(t_i, t_j) = (ls_0, \ldots, ls_{20}), \text{ with } ls_k = sim(a_{kt_i}, a_{kt_j}),$$

where

$$sim(a_{kt_i}, a_{kt_j}) = 1 - \frac{|a_{kt_i} - a_{kt_j}|}{range(a_k)}$$

We can now use the local similarity vector of each pair of teams to compute a global similarity measure:

$$sim_{ref}(t_i, t_j) = \sum_{k=0}^{20} w_k \cdot ls_k$$

where $w = (w_0, \ldots, w_{20})$ is a vector of weights in which $\sum_{i=0}^{20} w_i = 1$. We will refer to this global similarity measure as the *functional similarity*.

The list of attributes shown above is too comprehensive. Irrelevant or redundant attributes included in the list can have a negative effect on the results of the reference similarity function (Witten et al., 2005). We can use the weights w_i to adjust the relevance of an attribute in our similarity function. A weight of 1 would mean that an attribute is very relevant while a weight of 0 is of no relevance at all.

To obtain the weights we use an attribute selection algorithm that keeps the "good" attributes and rejects the "bad" ones. In terms of classification, we can state that an attribute is good if it is relevant to the class concept but not redundant to any other relevant attribute. That is, if it is highly correlated with the class but has little correlation with other attributes. In our case, the class concept is the teams that produced the statistics, so we will try to find which attributes correlate better with the teams for all the matches. To measure this correlation we used *symmetric uncertainty* (Yu & Liu, 2003).

The attributes with higher weights were the times in each field, the time near the opponent's goal and in the center region and the distances to the center of the field and each goal.

The *functional similarity* function is used to compute, for any pair of teams in the test set, a value of similarity that tells us how similarly they play. At this point, the test set is enhanced by including 14 reference teams (the *trainers*) obtained from the standard distribution of SoccerBots and from the winner teams of the annual SoccerBots competition held by the students of the Knowledge Based Systems course at the Complutense University of Madrid. We selected the trainers by paying attention to different statistics about each team, such as, the difference between wins and losses, the time of the players near each goal or the goals for and the goals against. Instead of selecting the best teams, we tried to select teams with different values in these statistics.

To create the game metrics dataset employed by the *functional similarity* function, each team in the test set played two matches of 1 min versus each of the 14 trainers. With this dataset we are ready to compare both functional and structural similarity functions.

6.3. Evaluating the structural similarity function

What we want to evaluate is to what degree, the results of a query using the structural similarity functions are similar to the results of the same query using the functional similarity. In this case, the similarity values are unimportant: rank is the only feature that matters.

To achieve this, we compute the similarity of the FSMs associated to all the possible pairs of teams in our test set: we use the FSM from each team t_q as a query, and compare it with the FSMs of the remaining teams from the test set. For each query we obtain a list, l_{t_q}, of the similarities of the FSM from t_q with all the remaining FSMs in the test set. The jth element of the list is $sim(t_q,t_j)$.

The lists are sorted according to the similarity obtained. The first element is the most similar using the corresponding similarity function, the second is the next in similarity, and so on. Given a similarity function *sim* and a query t_q, we will refer to the ranking list generated as R_{sim,t_q}.

The ranked lists will be created using two structural similarity functions, according to the heuristics proposed in Section 5: f_{id} and f_{edge}. This way, for each team t_q we will have an R_{id,t_q} list and an R_{edge,t_q}. Additionally, we will create for each team t_q a reference list R_{ref,t_q} using the functional similarity function.

To compare the rankings we use the Normalized Discounted Cumulative Gain (NDCG) (Manning, Raghavan, & Schtze, 2008), a measure commonly used in information retrieval to quantify the quality of a ranking. The NDCG is the normalized version of the DCG (Bollmann & Raghavan, 1988). DCG is the sum of the relevancies of the ranked recommended elements, multiplied by a discounting factor that penalizes relevant elements when they appear at bottom places of the ranking. DCG will be higher when the most relevant results appear classified higher in the ranking.

To calculate the DCG we used the following formula:

$$\mathrm{DCG}(R_{sim,t_q}, k) = rel(R(1)) + \sum_{i=2}^{k} \frac{rel(R(i))}{\log_2(i+1)}$$

where k is the number of elements we are considering for the query and *rel* is a relevance function that assesses how relevant is an element of the ranking for the query t_q, that is, how similar is its behaviour with the query. Hence, we used as relevance function the functional similarity measure explained in the last section. For the sake of simplicity we denote $R(i)$ to the i-th element of the list R_{sim,t_q}.

To normalize DCG and obtain the NDCG we have to divide it by the ideal DCG (IDCG), the maximum value of DCG for a given k. IDCG is computed as the DCG value for the k most similar elements to the query t_q, sorted by the functional similarity value.

Additionally, we computed the *Precision at k documents retrieved* (P@k) (Manning et al., 2008) on the ranked lists, using different values of k. Precision is the proportion of relevant documents retrieved by a query:

$$\text{Precision} = \frac{\text{relevant documents retrieved}}{\text{documents retrieved}}$$

P@k is a variant of precision that only takes into account the first k documents retrieved by the query. For instance, when $k = 10$:

$$\text{P@10} = \frac{\text{relevant documents in the first 10}}{\text{documents retrieved}(= 10)}$$

In the case of precision, we only distinguish if an element is totally relevant or not relevant at all, but not intermediate situations like we did with NDCG. Given a query, to decide if a FSM is relevant we use the ranking from the functional similarity R_{ref,t_q}. We consider that a team t_i is relevant given a query t_q if its similarity value $sim_{ref}(t_q, t_i)$ is over the percentile 90. It means that at least the 90% of the teams are less similar to t_q than t_i. In our case, having a sample size of 700 teams, this percentile represents that t_i should be ranked in the 70 first elements of R_{ref,t_q}.

P@k is useful to compare the goodness of the results when we vary the number of elements retrieved (i.e. the value of k). When the user makes a query, it is important that the system will return good results, but also how many relevant results exist on the small list that the editor shows to the user.

7. Results

7.1. Preliminary tests on the heuristic functions

Prior to performing the experiments we conducted a preliminary study that sought to ascertain whether the results of the behaviour retrieval using the heuristic similarity functions are similar to those obtained using what we will call the *classic* method, that is, the one that evaluates every possible edit sequence and returns the best one.

To carry out this study we used each of the twelve roles mentioned in Section 6.1 as a query, and compared it with the remaining roles using the three similarity measures: the classic edit distance and the heuristics, f_{id} and f_{edge}. We then calculated the absolute difference between the values of each heuristic and the classic edit distance.

Fig. 5a and b shows the average value (in bars) and the standard deviation (in lines) of these absolute differences for each of the queries. In both cases the averages move around 0.06, being the absolute average of f_{id} 0.0598 and of f_{edge} 0.0595. In light of this data, we can assert that, in our domain, both f_{id} and f_{edge} return values of similarity that are close to those of the classic edit distance algorithm.

We have assessed the statistical significance of the differences between the results of the similarity functions using the Wilcoxon rank sum test. From the results of the tests we can conclude that there are no statistically significant differences between the classic algorithm and f_{id} and between the classic algorithm and f_{edge} with a confidence level of 95%.

Table 1 shows the average execution times in milliseconds for the different algorithms. We have grouped the times by the number of nodes of each graph, which range from 2 to 9. From the figures in the table we can see that the times for the classic algorithm grow in an exponential manner, as was expected.

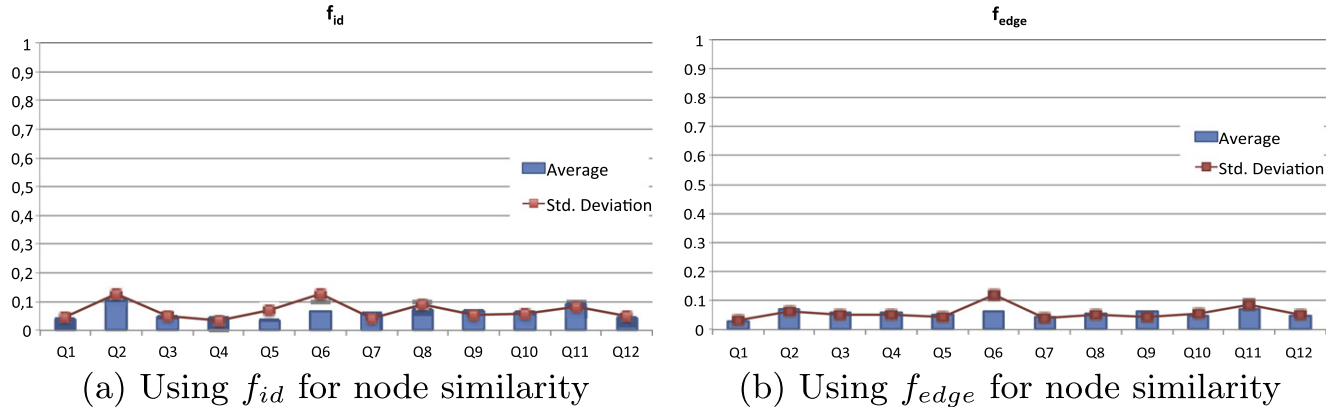
When we confront with larger graphs (e.g. an FSM for a team) the classic algorithm is not applicable because it would take too long to complete. Nevertheless, we can make an estimation of the time it would take to evaluate all the edit sequences for a pair of teams, based on the average times registered. For instance, when comparing two graphs with 9 nodes, the number of sequences to evaluate is $9! = 362880$. The average time elapsed for each sequence is:

$$\frac{55174}{9!} = 0.15\text{ms/sequence}$$

If we compare two graphs of 14 nodes (the smallest in our test set), the number of sequences grows to $14! = 87 \cdot 10^9$. If we suppose that each one takes 0.14 ms to be evaluated, the total amount of time to evaluate their similarity would be 153 days.

7.2. Experiment results

In this section we present the results obtained from running the experiment described in Section 6. The experiments have been executed on a Dual Core Intel Xeon processor with 2.33 GHz and 4 Gb of RAM, running Windows 2008 Server 32 bits and the Java JDK 6.0. Additionally to the two aforementioned structural similarity functions, f_{id} and f_{edge}, we added a random similarity function that returns a random ranking for each query. Using this ranking we can find out what the results would be for randomly returned values and use this value as a baseline to compare it with the results using the remaining functions.



(a) Using f_{id} for node similarity (b) Using f_{edge} for node similarity

Fig. 5. Differences between the results of similarity functions.

Table 1
Execution times for the "classic" algorithm.

Nodes	2	3	4	5	6	7	9
Classic times (ms)	0.04	0.37	1.43	11.1	91.5	704	55174
f_{id} Times (ms)	0.015	0.023	0.037	0.052	0.076	0.081	0.115
f_{edge} Times (ms)	0.024	0.116	0.281	0.734	1.71	2.304	2.996

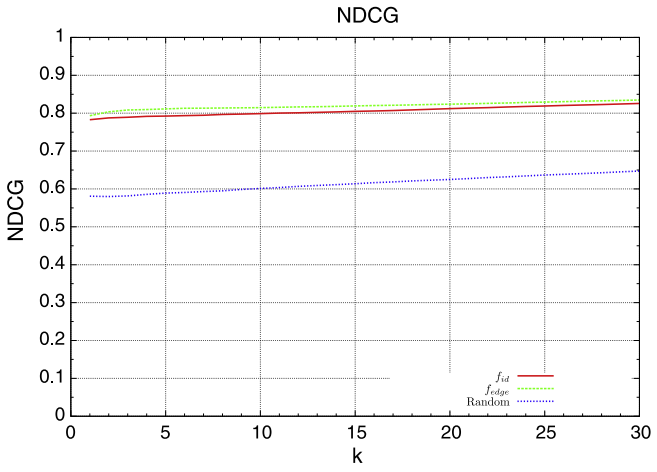


Fig. 6. NDCG.

In Fig. 6 we have represented the values of the NDCG for the first 30 values of k. We see that the values for both heuristic functions, f_{id} and f_{edge}, are quite high, hovering around 0.8. This means that both functions obtain a similar ranking to the one returned using the reference function. The results of this functions are superior to those of the random function, which are around 0.6. We have assessed the statistical significance of the differences between the average NDCG for each k using the Wilcoxon rank sum test. The differences between the Random function and both our proposals (f_{edge} and f_{id}) are statistically significant with a 95% confidence level. We can observe a slightly enhancement of the NDCG when employing the f_{edge} function but this difference with the results using the f_{id} function is only significant for k values contained in [3, 20] interval.

Fig. 7 shows the values of the P@k for different values of k in the interval between 1 and 30. When compared with other methods, the best results are returned by the one that uses the identity function together with the edges entering and leaving the nodes (f_{edge}). We have also assessed the statistical significance of the differences between the average P@k for each k using the Wilcoxon rank sum test. Using a 95% confidence level, we state that the differences between the f_{edge} and f_{id} and the random function are statistically significant. We can see that the information regarding the edges (f_{edge}) makes it behave better than the identity (f_{id}) to some extent. This enhancement in the precision is statistically significant with a 95% confidence level when the retrieved list contains between 2 and 19 elements (k value). However, as the number of elements increases, this difference can not be considered as significant, according to the statistical test employed.

We can see that the highest precision corresponds to lower values of k. This means that the relevant elements are more likely to be found at the first places of the ranking. For instance, for f_{edge}, the value of P@5 = 0.497 means that if we retrieve five documents there is an average of between 2 and 3 relevant documents between the first 5:

$$\text{P@5} = \frac{\text{relevant documents retrieved in the first5}}{5} = 0.497$$
$$\text{relevant documents retrieved} = 5 \cdot 0.497 = 2.49$$

whereas P@15 = 0.405 indicates that in the first 15 teams retrieved there are around 6 relevant documents. That is, if we show 10 more elements to the user, only 3 more relevant elements are found among them.

Using P@k alone we have information regarding how many relevant elements are retrieved, that is, how many of the retrieved elements have a similarity value above the percentile 90. But it does not contribute any information regarding the goodness of the remaining retrieved elements. Even though they are not relevant, the may have a similar behaviour to the one of the query. As a complement to the P@k, Fig. 8 shows the average value of the functional similarity of the first elements in the rankings for different query sizes.

Given a query t_q and a similarity function f we can obtain the ranking R_{f,t_q} with the results of the query, sorted by similarity. If we calculate the functional similarity for each of the k first elements of R we can learn what is the real similarity between the behaviours of the teams in the ranking and the behaviour of the query, regardless of whether they are relevant for the precision or not. Each point in the graphs of the figure represents the average of this similarity for each query and element of R in a position less that or equal to k.

For instance, the value of average similarity for k = 15 and for the function f_{edge} is 0.783. This means that, if we get the 15 first elements of R_{edge,t_q} for every query t_q, the average of their functional similarities is 0.783, indicating that, although the number of relevant elements retrieved is not very high (around 6), the similarity of the elements retrieved is high.

In the figure, we have plotted the two structural similarity functions, f_{id} and f_{edge}, the random similarity function and also the functional similarity, f_{ref}. The objective of plotting f_{ref} is to have a reference of the best average similarity we can obtain. In the graph

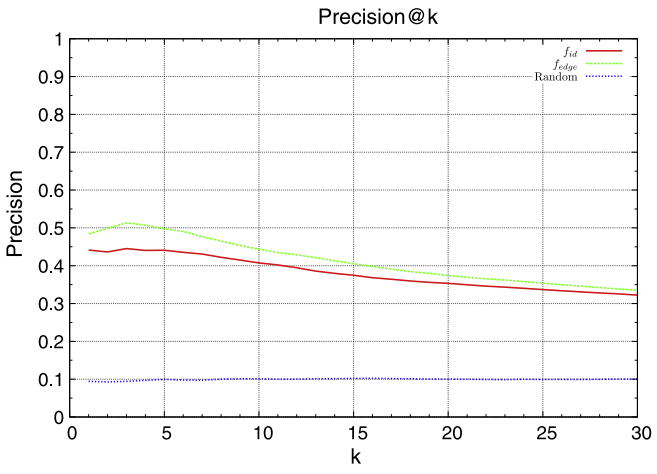


Fig. 7. Precision at k.

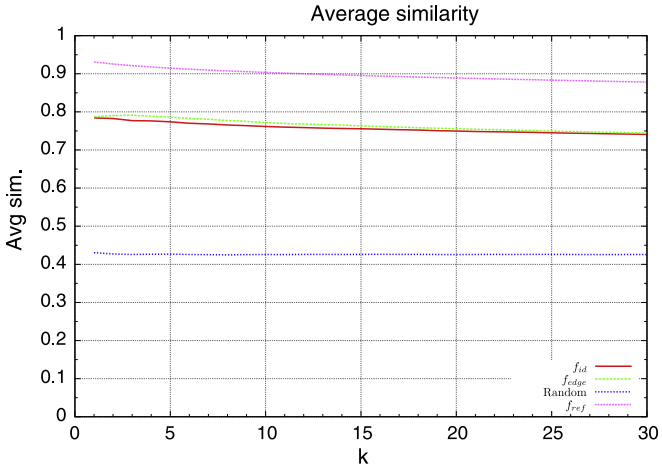


Fig. 8. Average similarity.

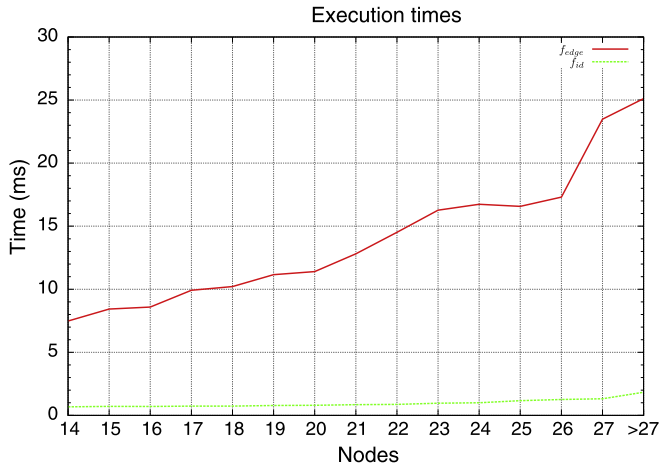


Fig. 9. Experiment execution times.

we can see that the results of both structural similarity functions are very close together and also close to the reference values.

We used the Wilcoxon rank sum test to assess the difference between the structural similarity functions, f_{id} and f_{edge}, and the Random function. The conclusion is that, again, the differences between Random and each of the structural functions are statistically significant with a 95% confidence level. However, although the differences in the average similarity values between the reference functions and both f_{id} and f_{edge} functions are small, these differences are statistically significant.

Lastly, we have measured the time the algorithm spends to retrieve similar FSMs. Fig. 9 shows the average execution times for each function. *X*-axis represents the number of nodes of the queries and *Y*-axis corresponds to the time in milliseconds. Each point in the graph represents the average time elapsed to compare a query with all the remaining behaviours in the test set. For instance, a query with 14 nodes takes an average time of 7.48 ms when using the f_{edge} heuristic. As expected, the proposed algorithms enhance the execution times of the classic algorithm that evaluates all posible edit sequences.

When comparing the times for the heuristics, we can see that f_{edge} takes more time to execute. This is due to the extra calculations needed to obtain the similarity of the edges of each pair of nodes. The difference grows as long as the number of nodes increases. The reason for this behaviour is attributable to the different costs of the heuristic functions. f_{edge}, requires more time to be evaluated than f_{id}, and when the number of nodes increases, the number of times the function has to be evaluated does the same.

7.3. Usability evaluation

To evaluate the usability of *eCo*, the editor presented in Section 4, we run a series of experiments with 43 students from the Knowledge Based Systems course of the Complutense University. In the first place, we divided the total amount of students in groups of 3. The experiment was run in two sessions of 2 h. In the first session we asked them that they followed a tutorial to learn how to use the editor. For the second session we asked them to design behaviours for several roles of a SoccerBots team (they had to design at least 3) and to implement a team with them using the editor. The average time they spent in implementing their teams was of 3 h.

After that, we asked them to complete an individual survey with assertions like "*Creating a behaviour using the editor is easier than programming it using Java*", and measured their agreement using a *Likert-like* scale. Regarding usability and interface questions, we asked things like if it is easier and faster to create behaviours in the editor than using other techniques or if its use is intuitive. In general, the answers were positive: around the 65% considered that the editor was an easier and faster alternative to programming the behaviours. An striking fact is that only 33% of the students agreed in that the behaviours created using the editor were as powerful as those programmed in Java, while 37% disagree in this assertion (the remainder neither agree nor disagree). These figures are explained because we are using Computer Science students, that have a profile that is closer to a programmer than to a designer, i.e., more oriented towards using programming languages than towards using design tools.

We also asked students about particular features of the behaviour editor, like the use of HFSMs or the code editor. Around the 80% of the students agreed in that the main features of the editor were useful. In addition, 70% of them thought that the search and recommendation features were also very useful, while only 5% thought they were not. We also wanted to evaluate if the editor was helping the students to achieve a better understanding of the matter they were studying. 65% agreed in that the editor helped them to better understand the concepts behind agent control using HFSMs and 55% also declared that the use of the editor made the assignment more interesting.

The answers to these questions, together with the comments of the students on particular improvements to the features of the editor, have constituted a valuable feedback that helped us to empirically verify that searches and recommendations of behaviours are two useful features from the point of view of potential users.

8. Conclusions and future work

In this paper we have presented *eCo Behaviour Editor*, an authoring tool for game designers that leverage a collection of reusable behaviours by providing tools and techniques for storing, indexing, retrieving and reusing previously designed behaviours. The main contribution of the proposed tool is to provide a smooth mechanism for retrieving similar behaviours to the one being built, understanding that retrieval is crucial for this approach to scale-up when dealing with large collections of reusable behaviours.

To achieve this so-called sketch-based retrieval we need a measure to assess if two behaviours are similar in terms of how they behave in the game, but we cannot make them play to verify it. On the contrary, sketch-based retrieval relies on comparing the behaviour graph that is partially drawn (the sketch), with respect to a repository of existing behaviours. We have described different algorithms to allow for this graph comparison, and provided a heuristic structure-based similarity for graphs that preserves behaviour similarity and can be computed at reasonable cost.

The question that raises now is whether structural similarity can be used to retrieve behaviours, not only with a similar structure, but also with a similar behaviour. To answer this question we have presented a case study domain, behaviours of SoccerBots robots represented as HFSMs, and designed an experiment to compare the results of the heuristic structural similarity functions with those of a reference similarity function that measures the similarity in the way they play. In our experiment we have demonstrated that if we consider only the best node mapping, we still find behaviours that "behave" in a similar way.

We have shown in Section 7 that both the structural similarity functions and the reference function rank the results in a similar order, obtaining a NDCG that is around 0.8. We also have shown that the most relevant results are within the first few results retrieved. This means that, when showing the results to the users, they will be more likely to find the relevant behaviours in the first page of the results, which is a desirable quality for them. As a

conclusion to the experiment we can state that using structural similarity we can retrieve behaviours that are similar in the way they behave in the game.

Regarding the applicability of the proposed approach, it should be noticed that sketch-based retrieval from a library of reusable behaviours only makes sense when reusing behaviours actually provides benefits to the authoring process. While in game genres such as sport games, shooters, or action games behaviours tend to repeat and therefore reuse should bring benefits, in more narrative-driven games it may be more difficult to find significant portions of reusable behaviours.

We also carried out an experiment with real users. In such experiment, students created several teams from scratch using the editor, and completed a survey about their experience. The results of this survey indicate that the editor improves the process of creation of behaviours in terms of its complexity and time (i.e. it is easier and faster). Additionally, students consider the feature of making searches in a repository to be positive and useful. In addition to the surveys, we also have the behaviours created by the students and the trace of their actions while they were using the editor. As a future work we plan to use this information to assess the gains in productivity, development time and quality that can be obtained through the proposed approach.

Furthermore, in order to improve precision in retrieval we plan to integrate more domain knowledge into the node similarity measure. To this end we are planning to embed the behaviours in a domain ontology and use a similarity measure that takes into account the relations between them. We expect that this added knowledge translates into a more accurate retrieval of the behaviours.

References

Alexander, G., Youngblood, G. M., Heckel, F. W. P., Hale, D. H., & Ketkar, N. S. (2010). Rapid development of intelligent agents in first/third-person training simulations via behavior-based control. In *Proceedings of the 19th behavior representation in modeling and simulation conference (BRIMS)*.

Bollmann, P., & Raghavan, V. V. (1988). A utility-theoretic analysis of expected search length. In *Proceedings of the 11th annual international ACM SIGIR conference on Research and development in information retrieval, SIGIR '88* (pp. 245–256). New York, NY, USA: ACM.

Bourg, D. M., & Seemann, G. (2004). *AI for game developers*. Sebastopol: O'Reilly Media, Inc.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation, 2*(1), 14–23.

Bunke, H., & Messmer, B. T. (1994). Similarity measures for structured representations. In *EWCBR '93: Selected papers from the First European workshop on topics in case-based reasoning* (pp. 106–118). London, UK: Springer-Verlag.

Burkard, R. E., Dell'Amico, M., & Martello, S. (2009). *Assignment problems*. SIAM.

Canossa, A., & Drachen, A. (2009). Patterns of play: Play-personas in user-centred game development. In A. Barry, K. Helen, & K. Tanya (Eds.), *Breaking new ground: Innovation in games, play, practice and theory: Proceedings of the 2009 digital games research association conference*. London: Brunel University.

Esmurdoc, C. (2005). Head games: Double fine's psychonautic break. *Game Developer Magazine, 12*(7), 30–38.

Flórez-Puga, G., Gómez-Martín, M. A., Gómez-Martín, P. P., Díaz-Agudo, B., & González-Calero, P. A. (2009). Query enabled behaviour trees. *IEEE Transactions On Computational Intelligence and AI in Games, 1*(4), 298–308.

Flórez-Puga, G., Llansó, D., Gómez-Martín, M. A., Gómez-Martín, P. P., Díaz-Agudo, B., & González-Calero, P. A. (2011). Empowering designers with libraries of self-validated query-enabled behaviourtrees. In P. A. González-Calero & M. A. Gómez-Martín (Eds.), *Artificial intelligence for computer games* (pp. 55–82). New York: Springer.

Gerbaud, S., Mollet, N., Ganier, F., Arnaldi, B., & Tisseau, J. (2008). Gvt: a platform to create virtual environments for procedural training. In *IEEE virtual reality conference 2008 (VR 2008)* (pp. 225–232).

Glickman, M. E. (1995). Chess rating systems. *American Chess Journal, 3*, 59–102.

Heckel, F. W. P., Youngblood, G. M., & Hale, D. H. (2009). Behaviorshop: An intuitive interface for interactive character design. In C. Darken & G. M. Youngblood (Eds.), *AIIDE*. Stanford, California: The AAAI Press.

Heckel, F. W. P., Youngblood, G. M., & Ketkar, N. S. (2010). Representational complexity of reactive agents. In G.N. Yannakakis, J. Togelius, (Eds.), *CIG* (pp. 257–264). IEEE.

Hocking, C. (2009). Ubisoft Montreal's Far Cry 2 postmortem. *Game Developer Magazine*, 30–38.

Isla, D. (2005). Handling complexity in the Halo 2 AI. In *Game developers conference*.

Jiménez-Díaz, G. & Díaz-Agudo, B. (2007). SB tournament: Competiciones de robots en asignaturas de inteligencia artificial. In *Procs. of the 9th edition of the international symposium on computers in education*. SIIE.

Jiménez-Díaz, G., Menéndez, H. D., Camacho, D., & González-Calero, P. A. (2011). Predicting performance in team games. The automatic coach. In *3rd international conference on agents and artificial intelligence (ICAART 2011)* (pp. 401–406). Rome, Italy: SciTePress.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A. F., Burnett, M. M., Erwig, M., et al. (2011). The state of the art in end-user software engineering. *ACM Computer Survey, 43*(3), 21.

Manning, C. D., Raghavan, P., & Schtze, H. (2008). *Introduction to information retrieval*. New York, NY, USA: Cambridge University Press.

Millington, I. (2006). *Artificial intelligence for games. The Morgan Kaufmann series in interactive 3D technology*. San Francisco: Morgan Kaufmann Publishers Inc.

Riesen, K., & Bunke, H. (2009). Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing, 27*(7), 950–959.

Smith, G., Whitehead, J., Mateas, M., Treanor, M., March, J., & Cha, M. (2011). Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Transactions on Computer Intelligence and AI in Games, 3*(1), 1–16.

Witten, I. H. & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques (Morgan Kaufmann series in data management systems)* (2nd ed.). Morgan Kaufmann.

Yu, L. & Liu, H. (2003). Feature selection for high-dimensional data: A fast correlation-based filter solution. In *ICML* (pp. 856–863).

6.4. Similarity Measures in Hierarchical Behaviours from a Structural Point of View

Cita completa

Gonzalo Flórez Puga, Belén Díaz Agudo y Pedro Antonio González Calero. *Similarity Measures in Hierarchical Behaviours from a Structural Point of View*. Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference. Daytona Beach, Florida, 19-21 Mayo, 2010. P. 330–335.

Resumen original de la contribución

Case-Based Reasoning (CBR) systems dealing with complex object-based case representation structures need to employ complex structured-based similarity measures. However, obtaining such similarity requires to solve problems on graphs are known to be NP-complete. In this paper, we show that, in spite of its theoretical complexity, structured-based similarity is of practical use and can be incorporated into the CBR toolbox. We analyze, in terms of quality and efficiency, three different methods for assessing similarity between graphs, which we apply in the domain of behaviour generation for a soccer simulation environment (SoccerBots). Our implementation of such methods has been incorporated into jCOLIBRI, a general framework for CBR development, and ready to be tested on other applications.

Referencia de citas bibliográficas

Cunningham (2009), Garey y Johnson (1979), Bunke y Messmer (1994), Flórez Puga et al. (2008), Bunke y Messmer (1994), Champin y Solnon (2003), Wang y Ishii (1997), Levenshtein (1966), Cunningham (2009), Garey y Johnson (1979)

Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference (FLAIRS 2010)

Similarity Measures in Hierarchical Behaviours from a Structural Point of View *

Gonzalo Florez-Puga and **Belen Diaz-Agudo** and **Pedro Gonzalez-Calero**
Dep. de Ingenieria del Software e Inteligencia Artificial
Universidad Complutense de Madrid
28040, Madrid, Spain

Abstract

Case-Based Reasoning (CBR) systems dealing with complex object-based case representation structures need to employ complex structured-based similarity measures. However, obtaining such similarity requires to solve problems on graphs are known to be NP-complete. In this paper, we show that, in spite of its theoretical complexity, structured-based similarity is of practical use and can be incorporated into the CBR toolbox. We analyze, in terms of quality and efficiency, three different methods for assessing similarity between graphs, which we apply in the domain of behaviour generation for a soccer simulation environment (SoccerBots). Our implementation of such methods has been incorporated into jCOLIBRI, a general framework for CBR development, and ready to be tested on other applications.

Introduction

Although simple similarity measures through feature vectors is the most common approach to assess similarity in Case-Based Reasoning Systems, more complex case representations requiring more sophisticated similarity mechanisms have been investigated (see (Cunningham 2009) for a taxonomical review).

In particular we are interested in similarity measures for cases represented as graphs. A case representation language based on graphs is more expressive than one based on feature vectors, but, unfortunately, the problem of assessing similarity between two graphs is essentially intractable when using methods taking the graph structure into account. The methods for assessing similarity between two graphs are based on finding a subgraph isomorphism, which is an NP-complete problem (Garey and Johnson 1979), or computing some measure of the graph edit distance between the two graphs which is also NP-complete (Bunke and Messmer 1994).

The question that we try to answer in this work is whether we can find, for a particular application, any meaningful difference in terms of quality or execution time of the results between three methods used for graph-based similarity measures. To estimate the quality of the similarity, we compare with a high level description produced by an expert, in the domain of behaviour generation for a soccer simulation environment (SoccerBots).

The rest of the paper runs as follows. The next section briefly describes the methods for structured-based similarity between graphs. Section 3 describes the experiment and its results, while last Section presents the conclusions of the experiments.

*Supported by the Spanish Ministry of Science and Education(TIN2009-13692-C03-03)

Similarity Measures

We have proposed an approach to the similarity problem in graphs, and more specifically in finite state machines (Flórez-Puga, Díaz-Agudo, and González-Calero 2008) that is based in both the structure of the graph and the labeling in the nodes and edges. The labels associated to the nodes are used to express the functionality of the behaviours contained in them.

The next subsections deal with three different approaches to assess this similarity measure.

Edit Distance Based Similarity

This approach is based on the calculation of the edit distance between two graphs (Bunke and Messmer 1994). The distance is obtained as the sum of the operations needed to transform one graph into the other. The set of edit operations we are using is: adding a node (A), deleting a node (D) and editing the label of a node (E), and adding an edge (A'), deleting an edge (D') and editing an edge(E')).

Each operation has an associated cost (C_A, C_D, C_E, etc.). Using different sets of cost values will lead us to different results. In our approach, we are considering the costs of edit operations, not as constants, but as functions defined over the source and target nodes or edges. This way, we can express the intuitive idea that changing one label for another is cheaper in cost if the labels are more similar.

The edit distance ($dist$) is the minimum cost among all sequences of edit operations that transform the source graph into the target graph. The distance can be converted into a similarity measure by defining a function that uses the distance, like:

$$\mathrm{sim}(G_1, G_2) = [1 + dist(G_1, G_2)]^{-1}$$

We also impose the following restrictions on the possible values of the cost functions, so the results of the distance

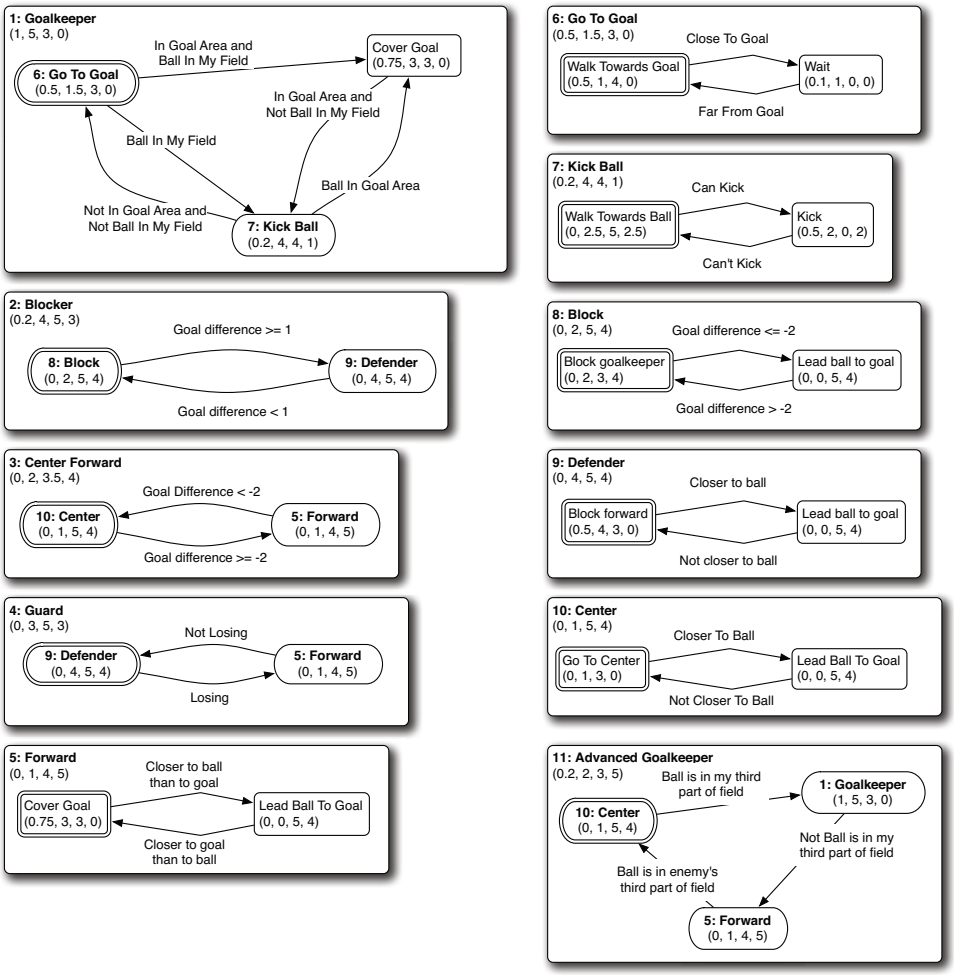


Figure 1: Graphs used in the experiment

function are reasonable:

1. $C_E \leq C_A + C_D$ and $C_{E'} \leq C_{A'} + C_{D'}$
   This means that editing the label of a node is cheaper than an addition and a deletion of the same node with different labels.
2. $C_A = C_D$ and $\mathrm{sim}(X, Y) = \mathrm{sim}(Y, X)$
   These two restrictions give symmetry to our distance measure.

Correspondence Based Similarity

This approach is based in the definition of a correspondence between the nodes of the query and the case graphs. It is based in the similarity measure proposed by (Champin and Solnon 2003).

Each graph G is defined by a triplet $\langle V, r_V, r_E \rangle$ where V is the finite set of nodes, r_V is a relation that associates vertices with labels, and r_E is a relation that associates pairs of vertices (i.e. edges) with labels. r_V and r_E is called the set of features of the graph. A correspondence C between G_1 and G_2 is a subset of $V_1 \times V_2$, that associates, to each vertex of one graph, 0, 1 or more vertices of the other.

Given a correspondence C between G_1 and G_2, the similarity is defined in terms of the intersection of the sets of features (r_V and r_E) of both graphs with respect to C.

We add a value β to each tuple in the intersection. This value represents the similarity between the labels of the nodes or edges:

$$
\begin{aligned}
&descr\,(G_1) \cap_C descr\,(G_2) = \\
&\quad \{(v, v', \beta) \mid (v, v') \in C \wedge (v, l) \in r_{V1} \wedge (v', l') \in r_{V2} \wedge \\
&\qquad \beta = \mathrm{sim}\,(l, l')\} \cup \\
&\quad \{\big((v_i, v_j), (v'_i, v'_j), \beta\big) \mid (v_i, v'_i) \in C \wedge \big(v_j, v'_j\big) \in C \wedge \\
&\qquad (v_i, v_j, l) \in r_{E1} \wedge \ \big(v'_i, v'_j, l'\big) \in r_{E2} \wedge \\
&\qquad \beta = \mathrm{sim}\,(l, l')\,\} \\
&\mathrm{sim}_C\,(G_1, G_2) = \\
&\qquad \frac{f\,(descr\,(G_1) \cap_C descr\,(G_2)) - g(splits(C))}{F}
\end{aligned}
$$

Where $splits(C)$ is the set of vertices from $V_1 \cup V_2$ which are associated to two or more vertices by the correspondence C.

The similarity degree of two graphs G_1 and G_2 is the maximum similarity of G_1 and G_2 over all the possible correspondences:

$$\mathrm{sim}\left(G_1, G_2\right) = \max_C \left\{\mathrm{sim}_C\left(G_1, G_2\right)\right\}$$

The similarity value β is used by the function f to obtain the final similarity value, and the constant F is an upper bound of f that maintains the result in the interval $[0, 1]$.

To simplify this approach, we can consider only the nodes and edges whose β is greater than a certain threshold.

Weighted Similarity

The third approach is also based in defining the possible correspondences between the graphs being compared, and is based on the one proposed by (Wang and Ishii 1997).

This method doesn't use the intersection, but an algebraic formula to obtain the final similarity measure. As in the previous approach, the similarity degree of two graphs G_1 and G_2 is the maximum similarity of G_1 and G_2 over all the possible correspondences.

The similarity of G_1 and G_2 over the correspondence C is

$$\mathrm{sim}_C\left(G_1, G_2\right) = \frac{F_n + F_e}{M_n + M_e}$$

$$F_n = \sum_{n \in V_1} \frac{W\left(n\right) + W\left(C\left(n\right)\right)}{2} \cdot \mathrm{sim}\left(n, C\left(n\right)\right)$$

$$F_e = \sum_{e \in E_1} \frac{W\left(e\right) + W\left(C\left(e\right)\right)}{2} \cdot \mathrm{sim}\left(e, C\left(e\right)\right)$$

$$M_n + M_e = \max\left\{\sum_{n \in V_1} W\left(n\right), \sum_{n \in V_1} W\left(C\left(n\right)\right)\right\} + \max\left\{\sum_{e \in E_1} W\left(e\right), \sum_{e \in E_1} W\left(C\left(e\right)\right)\right\}$$

where W is the weight of a node or an edge. The weight is a value between 0 and 1 that indicates the importance of a node or an edge in the final similarity result.

Experimental Results

For this experiment we are using Hierarchical Finite State Machines (HFSMs) that represent behaviours for the Soccerbots simulation environment. HFSMs are an extension to traditional Finite State Machines. In a HFSM there are two kind of nodes: atomic nodes, that contain actions that can be executed, and composite nodes, that contain a subordinate HFSM. This hierarchical organization can help to reduce the overall complexity of the state machine, favoring its legibility.

Figure 1 shows the testing set TS of HFSMs used for the experiment. The bold-typed ones are composite behaviours that reference another HFSM. The ones in plain type are atomic behaviours that cannot be further decomposed.

Each behaviour, wether it is atomic or composite, has a set of attributes used to describe them. The attributes we are using for Soccerbots are:

- Goalkeeper: proficiency as goalkeeper. Can take real values in the interval $[0, 1]$.
- Defender: proficiency as defender. Its valid interval is $[0, 5]$.
- Mobility: ability to move around the playing field. Can take real values between $[0, 5]$.
- Attacker: proficiency as attacker. Its valid interval is $[0, 5]$.
- Description: a natural language description of the behaviour. It's value can be any string.

All the behaviours in TS have been classified by an expert using this set of attributes. Their values are shown in Figure 1, under the name of each behaviour. For the sake of clarity we have omitted the attribute names and the textual description.

Experimental Similarity Measures

As we have seen, to completely specify a graph similarity function we need two more similarity functions, one for nodes and another one for edges.

To measure the edge similarity we use a function based on the Levenshtein distance on the edge labels (Levenshtein 1966).

To obtain node similarity we used two different functions:

- Attribute based similarity: the similarity is given by the average of the similarity of each attribute of the behaviour in the node, including the textual description:

$$sim_{ATTR}(n_1, n_2) = sim_{AS}(n_1.attrSet, n_2.attrSet)$$

$$sim_{AS}(attrSet_1, attrSet_2) = = \sum_{\substack{attr \in \\ attrSet_1 \cap attrSet_2}} \left(\frac{sim_{attr}(n_1.attr, n_2.attr)}{max\{|attrSet_1|, |attrSet_2|\}}\right)$$

$$sim_{attr}(a_1, a_2) = \frac{|a_1 - a_2|}{L}$$

  where $n.attrSet$ is the set of attributes associated to the behaviour in node n and L is the size of the interval of valid values for each attribute.

- Structural similarity: since the graphs we are comparing correspond to HFSMs, if the node is composite it can contain a subordinate HFSM. If this is the case for both of the nodes, we can use a graph similarity measure to compare the subordinate HFSMs:

$$sim(n_1, n_2) = = \begin{cases} \bullet\ n_1 \text{ is composite } \wedge\ n_2 \text{ is composite:} \\ \quad sim_{STR}(n_1.subgraph, n_2.subgraph) \\ \bullet \text{ otherwise:} \\ \quad sim_{ATTR}(n_1, n_2) \end{cases}$$

  Where $n.subgraph$ is the graph corresponding to the subordinate behaviour of a composite node and sim_{STR} is

any of the structure based similarity measures in section : Edit Distance Similarity, Correspondence Based Similarity or Weighted Similarity.

Another variation axis is the use of flattened HFSMs. The process of flattening a HFSM consists in transforming it in another state machine that has the same functionality, and in which all the nodes are atomic. We can use the following algorithm to flatten a HFSM:

```
for each composite node n in HFSM
  set G' to the sub-behaviour
  contained in n
  for each edge e that enters n
    change the destination of e to
    the first node of G'cx
  end for
  for each edge e that leaves n
    for each node n' in G'
      create a copy of e and change
      its origin to n'
    end for
    remove e
  end for
end for
remove n
add G' to HFSM
```

The similarity value is obtained flattening the state machines before applying any of the similarity functions. In particular, for our experiment, we flattened the state machines down to the last but one hierarchy level.

In summary, we are using 10 different similarity measures, that we will name as follows:

1. ATTR: Attribute based similarity.
2. EDA: Edit distance with attribute based similarity for sub-behaviours.
3. CSA: Correspondence based similarity with attribute based similarity for sub-behaviours.
4. WSA: Weighted similarity with attributes based similarity for sub-behaviours.
5. EDS: Edit distance using structural similarity for sub-behaviours.
6. CSS: Correspondence based similarity using structural similarity for sub-behaviours.
7. WSS: Weighted similarity using structural similarity for sub-behaviours.
8. EDF: Edit distance using structural similarity for sub-behaviours. The HFSMs are flattened prior to similarity assessing.
9. CSF: Correspondence based similarity using structural similarity for sub-behaviours. The HFSMs are flattened prior to similarity assessing.
10. WSF: Weighted similarity using structural similarity for sub-behaviours. The HFSMs are flattened prior to similarity assessing.

Figure 2: Similarity results

For the correspondence based similarity measures (CSA, CSF and CSS) the functions f and g we used were:

$$f(I) = \sum_{\text{for each node n in I}} (f_N(n)) + \sum_{\text{for each edge e in I}} (f_E(e))$$

$$f_N((v, v', \beta)) = \beta$$

$$f_E(((v_i, v_j), (v'_i, v'_j), \beta)) = \beta$$

$$g(S) = |S|$$

$$F = \max\{|r_{V1}|, |r_{V2}|\} + \max\{|r_{E1}|, |r_{E2}|\}$$

For the weighted measures (WSA, WSF and WSS) we are supposing that the weights for nodes and edges are 1.

Procedure and results

For each similarity measure, sim, and each HFSM, Q, from the test set (TS), our experiment consisted in measuring its similarity to the remaining individuals in the set. Therefore, for each similarity measure and HFSM, we got a list, $L_{(sim,Q)}$, with the rest of HFSMs, sorted by its similarity to Q.

To compare the results of the different measures, we compared each $L_{(sim,Q)}$ with a list obtained applying experts

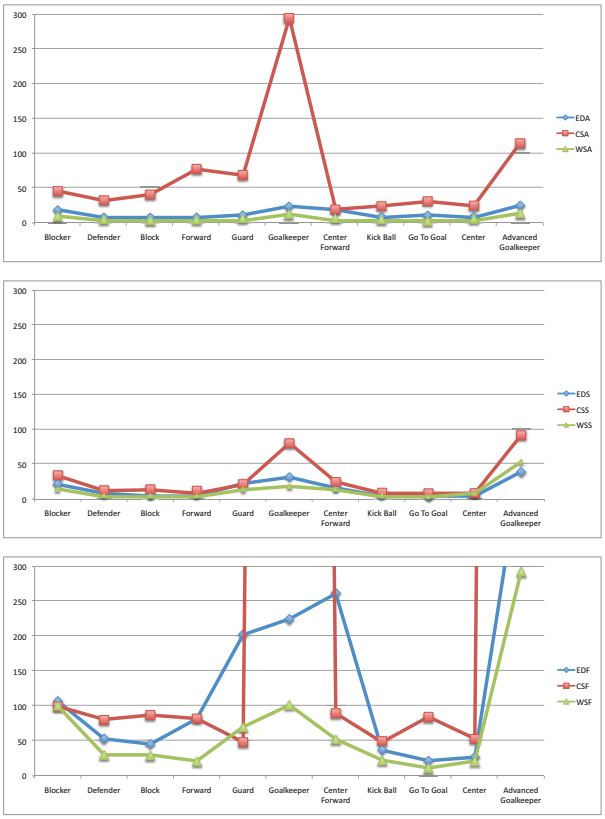


Figure 3: Times measured for the execution of each query

knowledge. In this case, we used the $ATTR$ similarity function with the expert parameters, $L_{(ATTR,Q)}$.

Both lists being compared are permutations of the set $SS = TS \setminus Q$ and, thus, they are equal on size. To compare them, we used the following similarity measure:

$$sim_L(l_1, l_2) = \frac{\sum_{e \in SS} (1 - dist(e, l_1, l_2))}{|SS|}$$

$$dist(e, l_1, l_2) = \frac{|pos_{l_1}(e) - pos_{l_2}(e)|}{|SS|}$$

Where $pos_l(e)$ is the position of element e in the list l.

Figure 2 shows the similarity results obtained for each measure using each graph of TS as query.

Conclusions and Future Work

Regarding quality of the results, we have found no meaningful difference between the three methods. As Figure 2 shows, the three methods provide highly similar results along all the examples in the corpus.

Regarding efficiency, on the other hand, we can conclude that for this particular domain, the WSS method consistently outperform the other two, as can be seen in Figure 3. Nevertheless, further experimentation should be done in order to characterize the properties of the soccerbots domain that are responsible for these results.

All three methods follow the general algorithm shown in Listing 1, differing in the assignment of the initial list of correspondences (in line 2) and the similarity function used to obtain the similarity of two graphs given a correspondence (line 7). Edit distance similarity and weighted similarity use one-to-one correspondences, that is, each node in the first graph is mapped to one node in the second (in the event that one graph has more nodes than the other, the extra nodes will be mapped to $\emptyset$). The correspondence based method uses a many-to-many correspondence in which each node is mapped to zero, one or more nodes of the other graph. In the case of weighted similarity, we also made test using many-to-many correspondences, which gave similar times to the ones obtained for the correspondence based methods.

The reason for this is that the similarity function is evaluated once per correspondence in the correspondence list LC. Given two graphs, G_1 and G_2, in the case of one-to-one correspondences it means that it is evaluated $\max\{|N_{G_1}|, |N_{G_2}|\}!$, being N_{G_i} the set of nodes of graph G_i. On the other hand, for many-to-many correspondences, the similarity function is evaluated $2^{|N_{G_1}| \cdot |N_{G_2}|}$ times.

This also explains the peaks found for the goalkeeper and the advanced goalkeeper behaviours. Both behaviours have 3 nodes each. In the worst case, when they are being compared one against the other, the number of correspondences is 3! for one-to-one correspondences and 2^9 for many-to-many. If we flatten the graphs, this difference is even greater. The flattened Goalkeeper behaviour has 5 nodes and the Advanced Goalkeeper, when flattened, has 9. For the one-to-one correspondences, we have $9! = 362880$ different cases, and for many-to-many correspondences the number grows to $2^{5 \cdot 9} = 35.18 \cdot 10^{12}$

For the typical cases we are dealing with, with graphs sizes between two and six nodes, the one-to-one correspondence leads to better results in the number of evaluations of similarity function.

In any case, heuristic methods can be found to prune the search space, so we can deal with bigger graphs. These methods depend on the graph, node and edge similarity functions and demand further research.

Although in this paper we have focused in similarity, our work is also related to reuse as more similar cases are more easily adaptable and more applicable in the query situation. As future work we will consider if there are dependencies between how useful and reusable were the structurally similar cases. An advantage that the approach discussed in the paper is that it already takes into account some amount of the effort needed to transform one case to another.

References

Bunke, H., and Messmer, B. T. 1994. Similarity measures for structured representations. In *EWCBR '93: Selected papers from the First European Workshop on Topics in Case-Based Reasoning*, 106–118. London, UK: Springer-Verlag.

Champin, P. A., and Solnon, C. 2003. Measuring the similarity of labeled graphs. In Ashley, K. D., and Bridge,

```
1  similarity(Q: Graph, G: Graph)
2    generate a list LC of correspondences
3        between nodes of Q and G
4    set best_sim to −∞
5    for each correspondence C in LC
6      add_Required_Edges(C, Q, G)
7      set current_sim to sim_C(Q, G)
8      if best_sim < current_sim then
9        set best_sim to current_sim
10   end for
11 end


14 add_Required_Edges (C: Correspondence,
15     Q: Graph, G: Graph)
16   for each edge e in Q
17     set n_o to origin of e
18     set n_t to target of e
19     set n'_o to C(n_o)
20     set n'_t to C(n_t)
21     if there is an edge e' from n'_o to n'_t
22       add the pair (e, e') to C
23       mark e' as visited
24     else
25       add the pair (e, ∅) to C
26     end if
27   end for
28   for each edge e' in G
29     if e' is not marked
30       add the pair (∅, e') to C
31 end
```

Listing 1: Pseudo-code for the general similarity assessment

D. G., eds., *5th Int. Conf. On Case-Based Reasoning (ICCBR 2003)*, LNAI, 80–95. Springer.

Cunningham, P. 2009. A taxonomy of similarity mechanisms for case-based reasoning. *IEEE Transactions on Knowledge and Data Engineering* 21(11):1532–1543.

Flórez-Puga, G.; Díaz-Agudo, B.; and González-Calero, P. 2008. Experience-based design of behaviours in videogames. In Springer-Verlag., ed., *Proceedings Of The 9th European Conference on Case Based Reasoning*, 180–194.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10(8):707–710.

Wang, Y., and Ishii, N. 1997. A method of similarity metrics for structured representations. *Expert Systems with Applications* 12(1):89–100.

6.5. Experience-Based Design of Behaviors in Videogames

Cita completa

Gonzalo Flórez Puga, Belén Díaz Agudo y Pedro Antonio González Calero. *Experience-Based Design of Behaviors in Videogames*. Proceedings of the 9th European Conference on Advances in Case-Based Reasoning. Trier. 1-4 Septiembre, 2008. P. 180–194.

Resumen original de la contribución

Artificial intelligence in games is usually used for creating player's opponents. Manual edition of intelligent behaviors for Non-Player Characters (NPC) of games is a cumbersome task that needs experienced designers. Amongst other activities, they design and integrate the new behaviors with the virtual environment, in terms of perception and actuation over the environment. Behaviors typically use recurring patterns, so that experience and reuse are crucial aspects for behavior design. In this paper we present a behavior editor (eCo) using Case Based Reasoning to retrieve and reuse stored behaviors represented as hierarchical state machines. In this paper we focus on the application of different types of similarity assessment to retrieve the best behavior to reuse. eCo is configurable for different domains. We present our experience within a soccer simulation environment (SoccerBots) to design the behaviors of the automatic soccer players.

Referencia de citas bibliográficas

Bowling et al. (2006), Flórez-Puga y Díaz-Agudo (2007), Champandard (2003), Brownlee (2002), Girault et al. (1999), Fu y Houlette (2002), Manning et al. (2008), Bunke y Messmer (1994), Wagner y Fischer (1974), Champin y Solnon (2003), Wang y Ishii (1997), Gómez-Martín et al. (2003)

Experience-Based Design of Behaviors in Videogames

Gonzalo Flórez Puga, Belén Díaz-Agudo, and Pedro Gonzalez-Calero

Department of Software Engineering and Artificial Intelligence,
Universidad Complutense de Madrid, Spain
gflorez@fdi.ucm.es, {belend,pedro}@sip.ucm.es

Abstract. Artificial intelligence in games is usually used for creating player's opponents. Manual edition of intelligent behaviors for Non-Player Characters (NPC) of games is a cumbersome task that needs experienced designers. Amongst other activities, they design and integrate the new behaviors with the virtual environment, in terms of perception and actuation over the environment. Behaviors typically use recurring patterns, so that experience and reuse are crucial aspects for behavior design. In this paper we present a behavior editor (eCo) using Case Based Reasoning to retrieve and reuse stored behaviors represented as hierarchical state machines. In this paper we focus on the application of different types of similarity assessment to retrieve the best behavior to reuse. eCo is configurable for different domains. We present our experience within a soccer simulation environment (SoccerBots) to design the behaviors of the automatic soccer players.

1 Introduction

Artificial Intelligence for interactive computer games is an emerging application area where there are increasingly complex and realistic worlds and increasingly complex and intelligent computer-controlled characters. Interactive computer games provide a rich environment for incremental research on human-level AI behaviors. These artificial behaviors should provide more interesting and novel gameplay experiences for the player creating enemies, partners, and support characters that act just like human players [1].

The edition of intelligent behaviors in videogames (or simulation environments) is a cumbersome and difficult task where experience has shown to be a crucial asset. Amongst other activities, it implies identifying the entities which must behave intelligently, the kind of behaviors they must show (e.g. helping, aggressive, elusive), designing, implementing, integrating and testing these behaviors in the virtual environment.

Designing new behaviors could be greatly benefited from two features that are common in most of everyday videogames. First of all, modularity in behaviors. That means complex behaviors can be decomposed into simpler behaviors that are somehow combined. Second, simpler behaviors tend to recur within complex behaviors of the same game, or even in different games of the same genre.

For instance, in a soccer game "defend" could be a complex behavior that is composed of two simpler behaviors like "go to the ball" and "clear"; meanwhile "attack" could be composed of "go to the ball", "dribbling" and "shoot". Both features are useful to build new complex behaviors based on simple behaviors as the building blocks that can be reused.

In this paper we describe our ongoing work developing a graphical behavior editor that is able to store, index and reuse behaviors previously designed. Our editor (eCo) [6] is generic and applicable to different games, as long as it is configured by a game model file. The underlying technologies of eCo are Hierarchical Finite State Machines (HFSMs) [8] and Case Based Reasoning (CBR). In this paper we focus on the similarity assessment and retrieval processes and give some ideas about our future work on reuse.

HFSMs are appropriate and useful tools to graphically represent behaviors in games, which provide a suitable starting point to automatically generate the code that implements the behavior and that will be integrated in the game [4]. HFSMs facilitate the modular decomposition of complex behaviors into simpler ones, and the reuse of simple behaviors. The eCo behavior editor provides with a graphical interface which allows the user to manually create or modify behaviors just by "drawing" them. Using a CBR-based module, the user can make approximate searches against a case base of previously edited behaviors. Both technologies work tightly integrated. Initially, the case base is empty, so all the editing has to be done via the manual editing (graphic) tools. Once there are enough cases in the case base, new behaviors can be constructed by retrieving and reusing the stored ones.

First, in Section 2, we introduce some general ideas on behavior representation and present the approach followed by the eCo behavior editor. In Section 3 we show a small example of application of the editor to a simulation environment: SoccerBots. Section 4 describes the CBR module integrated in the editor focusing in the different ways of computing similarity. Finally, in Section 5 and 6, we present related work, future goals and conclusions.

2 Modelling Reusable Behaviors

In general terms, the execution of a computer video game can be viewed as the continuous execution of a loop of perceiving, deciding the behavior, acting and rendering tasks. The behavior for each NPC basically decides the set of actions or reactions performed by the controlled entity, usually in relation with its environment. In a computer game or simulation, each entity gathers information about its environment using a set of sensors, which could be compared to the senses of the living beings. Depending on this information, the entity performs certain actions, using a set of actuators. In general, the set of sensors and actuators is unique for all the entities of a game and is different for each game or simulation environment, although there will be similarities between games of the same genre. For example, sensors in a first-person-shooter (FPS) game will give access to the position, the steering, the health, the visibility of other entities or

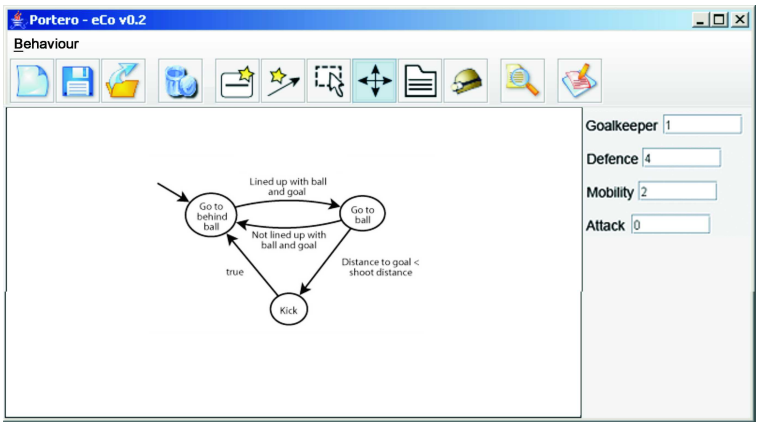


Fig. 1. Example of a HFSM

the remaining fuel of a vehicle. Regarding the actuators, the entity can shoot, look at or go to a place, talk to other entities, among others.

Several suitable techniques exist for the representation of behaviors. Due to its expressive power and simplicity, Finite State Machines (FSMs) is one of the most popular techniques. FSMs have been used successfully in several commercial games, like Quake [2], and in game editing tools, like Simbionic [7]. A FSM is a computation model composed of a finite set of states, actions and transitions between states. Simple states are described by the actions or activities which will take place in a state and the transitions point out changes in the state, and are described by conditions formulated over the sensors. One of the drawbacks of the FSMs is that they can be very complex when the number of states grows. To prevent this situation, we used Hierarchical Finite State Machines (HFSMs), which are an extension to the classic FSMs. In a HFSM , besides a set of actions, the states can contain a complete HFSM, reducing the overall complexity and favoring its legibility [8].

We have developed eCo, a game designer oriented tool that represents behaviors using HFSMs. The main module offers a graphical editor to manually "draw" the state machine representing a certain behavior. It includes tools for loading, saving and importing the behaviors from disk, drawing and erasing the nodes and edges, and specifying their content (actions or subordinate state machines, and conditions respectively). Once the behavior is complete, it is possible to use the code generation tool to generate the source code corresponding to the behavior. This tool uses the structure of the state machine together with the information in the *game model* to generate the source file. As the game model and the source file required are usually different for each game, the code generator will also be unique for each game.

The *game model* is a configuration file that describes some details of a game or a simulation environment. Each game model is an XML file, which includes the information about sensors and actuators, and a set of descriptors. The sensors

and actuators are obtained from the game API. Descriptors are the attributes used by the CBR module to describe the behaviors and retrieve them from the case base. The descriptors are obtained through the observation of the characteristics of the different behaviors that exist in the domain of the game.

Every manually designed behavior is stored and indexed and, as behaviors tend to recur, there is a CBR module that allows retrieving and reusing behaviors previously stored. We use XML files to store the cases. Each case in the case base represents a behaviour using the following components:

- Attributes: descriptors that characterize different properties of the behavior. The attributes are different for each game, although similar games (e.g. games of the same genre) will share similar attribute sets. The designer specifies as many attributes as necessary in the game model.
- Description: textual description of the behavior used to fine tune the description given by attributes.
- Enclosed behaviors: specifies which behaviors are hierarchically subordinated. This allows the user to retrieve behaviors which include a specific set of sub-behaviors or actuators.

Next we describe an example using a Soccer simulation environment.

3 SoccerBots Example

As we have already mentioned, the behavior editor described in Section 2, and the CBR system that we are describing in Section 4, are independent of any specific game. However, for the sake of an easier exposition we are explaining the basic ideas using a simple game. SoccerBots is a simulation environment developed by Tucker Balch, where two teams play in a soccer match. Simulation time, behavior of robots, colours, size of field, and many other features are configured from a text file. Basically, rules are similar to those from Robocup.

The first step in using eCo to generate behaviors for the SoccerBots environment is to define the game model with the information about sensors, actuators and CBR descriptors of the SoccerBots simulation environment. In the SoccerBots API we can find sensors like getBallX, getBallY, which checks the X, Y position of the ball, getBallR, which checks its distance, and getBallT, which checks its angle. Some examples of actuators (i.e. actions that robots can take) are kick, which kicks the ball, setSpeed(int), which changes the speed of the robot, or setSteerHeading(int), which changes the direction the robot is facing.

As we stated before, the descriptors are obtained through the observation of the characteristics of the different possible behaviors. We used four numeric descriptors to characterize SoccerBots behaviors, namely *mobility* is the ability to move all over the playfield; *attack* is the ability of the robot to play as an attacker; *defence* is the ability of the robot to play as a defender; and *goalkeeper* is the ability of the robot to cover the goal. Next section describes how to deal with these and others ways of describing behaviors in the CBR system.

4 CBR for experience based behaviour design

Case Based Reasoning is specially well suited to deal with the modularity and reuse properties of the behaviors; it assists the user in the reuse of behaviors by allowing her to query a case base. Each case of the case base represents a behavior. By means of these queries, the user can make an approximate retrieval of behaviors previously edited, which will have similar characteristics. The retrieved behaviors can be reused, modified and combined to get the required behaviors.

Initially, the case base is empty, so all the editing has to be done via the manual editing (graphic) tools. Once there are enough cases in the case base, new behaviors can be constructed by retrieving and adapting the stored ones. The number of cases necessary in the case base to obtain relevant results will vary from game to game, depending on the complexity of the descriptors and the heterogeneity of the behaviors that can be constructed for that particular game. In the example of the Soccerbots environment, we began with a small case base composed of five cases, and made it grow until we obtain reasonable results for the queries. This happened with a case base of 25 cases. To analyse the goodness of the results of the queries we adopted a subjective criteria but we should work about other more quantifiable criteria.

There are two kinds of queries: functionality based queries and structure based queries. In the former, the user provides a set of attribute-value parameters to specify the desired functionality for the retrieved behavior. In the latter, a behavior is retrieved, whose composition of nodes and edges is similar to the one specified by the query.

4.1 Functionality based retrieval

The most common usage of the CBR system is when the user wants to obtain a behavior similar to a query in terms of its functionality. The functionality is expressed by means of a set of parameters, which can be any (or all) of the descriptors of the cases presented in Section 2 (i.e. the attributes, the textual description and the enclosed behaviors).

The parameters that form the query are used to describe the behavior, and are closely related to the game model. The more differences exist between two games, the more different the associated behaviors are and, hence, the parameters used to describe them. The eCo editor provides a query form, showed in figure 2, for the user to enter the parameters of the query.

To obtain the global similarity value between the cases and the query, the similarity of the numeric and symbolic attributes is aggregated with the similarity due to the textual description of each behavior. The user can select the most appropriate operator to combine them in the query form. Some examples of operators could be the arithmetic and the geometric mean or the maximum.

The user enters the query using a form and (s)he can also select the similarity measure used to compare them to the ones in the case base. Descriptor based similarity is based on standard similarity measures here, like the normalized difference value for numbers.

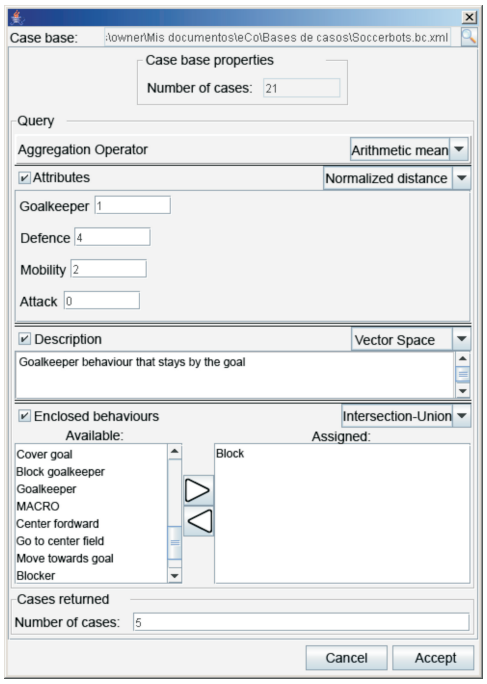


Fig. 2. Functionality based queries

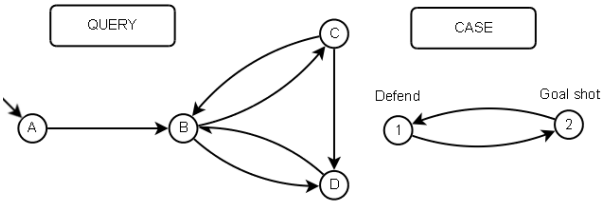
To easy the querying process, the user can use a textual description that is used to fine tune the query by including in it characteristics not considered by the attributes. Each case is described by a short textual description of the represented behavior. For instance, in the previous example, the user is requesting a behavior that stays near the goal. This descriptor was not included in the game model, as it is not relevant for most of the behaviors. Instead, the textual description is used. In the current version, we use the vector space model [10] to compute the similarity measure between the text descriptions.

4.2 Structure based retrieval

There are cases in which the behaviour designer knows the general structure of the state machine (i.e. the distribution of the nodes and edges and the generic functionality of them). In these cases, it would be easier and faster for the designer if he could "draw" the state machine and let the editor find a similar state machine in the case base.

Finite state machines are directed graphs, so we can compare them using any of the existing techniques in the literature. In the left part of figure 3 there is an example of a query for a Soccerbots behaviour.

Entering this data, the retrieved state machine would be similar to the query in terms of its shape, but the behaviour it implements could be any. Hence, we



	Defend	Goal shot
Go to my goal	0.5	0
Defend	1	0
Dribble	0.5	0.5
Goal shot	0	1

Fig. 3. Query and case for structure based retrieval and similarity between nodes

need to allow the behaviour designer to point out the desired functionality of the retrieved state machine and then, compare the desired functionality with the functionality implemented in the nodes of the state machines in the case base.

The functionality of the drawn nodes is expressed linking each node to a functionality query (see Section 4.1) that the user must build to expresses the desired behaviour that should be contained in the node. The linked functionality queries are compared to the descriptors in the nodes of the behaviours in the case base during the query process. In the aforementioned example, and for the sake of simplicity, instead of expliciting the whole functionality query, we will use a descriptive name to express it. Thus, for instance, the user could link node *A* to a behaviour whose desired functionality is "Go to my goal". To do this (s)he must build a functionality query that expresses this and link it to the node. For the examples we will consider the following linking of the nodes:

- A: "Go to my goal".
- B: "Defend".
- C: "Dribble".
- D: "Goal shot".

Our approach to these *structure based queries* is to use the drawing facilities of the editor to "draw" the state machine (the behaviour pattern) and then assign functionality based queries to the nodes, which will show the functionality of each node. Figure 4 shows the query editor for the structure based queries. In the left pane the user can draw a behaviour pattern and in the right pane he can specify the desired functionality of the retrieved behaviour by entering a functionality query. Additionally, each node can be linked to another functionality query, as we have already mentioned, to tune up the search.

In the next section we review different techniques to calculate labelled graph similarity and how they can be applied to our specific problem.

Graph similarity

The graph similarity problem is an issue that has been approached in several

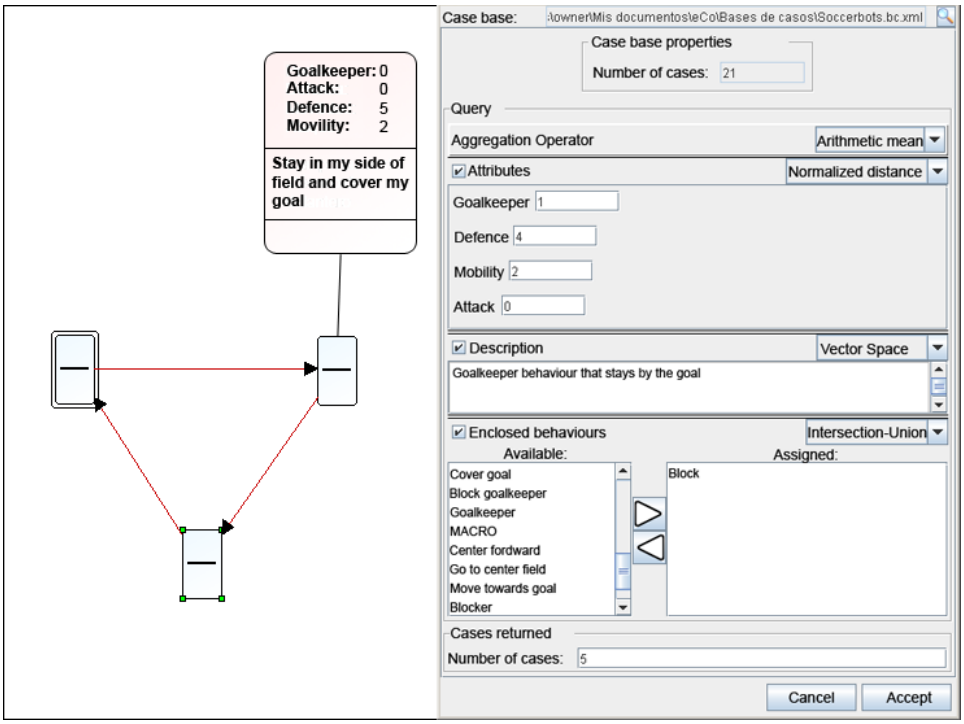


Fig. 4. Structure based query editor

different ways in the literature. Each approach has its own advantages and disadvantages. In the following paragraphs we review some of them and explain how we adapted them to solve our current problem, the labelled graph similarity.

First approach

Bunke and Messmer's approach [3] is based in the calculation of the weighted graph edit distance, a generalization of the string edit distance [11]. They define a set of edit operations (namely, adding a node (A), deleting a node (D) and editing the label of a node (E), and adding an edge (A'), deleting an edge (D') and editing an edge(E')). Each operation has an associated cost (C_A, C_D, C_E, etc.). Using different sets of cost values will lead us to different results. The edit distance ($dist$) is the minimum cost among all sequences of edit operations that transform the source graph into the target graph. The distance can be converted into a similarity measure by defining a function that uses the distance, like:

$$\mathrm{sim}(G_1, G_2) = [1 + dist(G_1, G_2)]^{-1}$$

For instance, for the example in figure 3, valid sequences of edit operations are:

$$S_1 = \{D(A), D(C)\}$$
$$S_2 = \{D(A), D(B), E(C)[\text{Dribble} \rightarrow \text{Defend}], A'(D, C)\}$$
$$S_3 = \{E(A)[\text{Go to my goal} \rightarrow \text{Goal shot}], D(C), D(D), A'(B, A)\}$$

$C_1 = 2 \cdot C_D$	$C_2 = 2 \cdot C_D + C_E + C_{A'}$	$C_3 = 2 \cdot C_D + C_E + C_{A'}$

Intuitively, if C_E and $C_{A'}$ are greater than 0, the sequence S_1 has the lowest cost, and therefore, is the edit distance.

The sequence associated to the edit distance contains the operations needed to transform one graph into the other, and hence, it can be used to perform the adaptation of the retrieved behaviour later.

In the worst case, the complexity of the computation of the graph edit distance is exponential in the size of the underlying graphs, although it can be speeded up using heuristics and bound techniques.

This approach considers the labels in the nodes and edges of the graphs. Continuing with the former example, in the second sequence we deleted nodes A and B, and added an edge from D to C. After doing this edit operations, the resulting graph is equal in shape to the case graph, but still differs from it in the labels, so we have to use one edit operation to change the label on node C.

One of the limitations of this approach is, as we can see in the example, that all the node editing operations have the same cost (C_E) regardless of the labels contained in the nodes. For instance, sequence 2 and sequence 3 have the same cost, but the behaviours in nodes C (Dribble) and 1 (Defend) are more similar than the ones in nodes A (Go to my goal) and 2 (Goal shot). In our approach, as we will see later, we use a cost function. This function takes into account the similarity of nodes in edit operations.

Second approach

The approach followed by Champin and Solnon in [5] is based on the definition of correspondences between nodes of the source and target graph.

Each graph G is defined by a triplet $\langle V, r_V, r_E \rangle$ where V is the finite set of nodes, r_V is a relation that associates vertices with labels, and r_E is a relation that associates pairs of vertices (i.e. edges) with labels. r_V and r_E is called the set of features of the graph. A correspondence C between G_1 and G_2 is a subset of $V_1 \times V_2$, that associates, to each vertex of one graph, 0, 1 or more vertices of the other.

Given a correspondence C between G_1 and G_2, the similarity is defined in terms of the intersection of the sets of features (r_V and r_E) of both graphs with respect to C:

$$\begin{aligned} &descr\,(G_1) \cap_C descr\,(G_2) = \\ &\qquad \{(v,l) \in r_{V1} |\, (v,v') \in C \wedge (v',l) \in r_{V2}\} \cup \\ &\qquad \{(v',l) \in r_{V2} |\, (v,v') \in C \wedge (v,l) \in r_{V1}\} \cup \\ &\qquad \left\{(v_i,v_j,l) \in r_{E1} |\, (v_i,v_i') \in C \wedge \left(v_j,v_j'\right) \in C \wedge \left(v_i',v_j',l\right) \in r_{E2}\right\} \cup \\ &\qquad \left\{\left(v_i',v_j',l\right) \in r_{E2} |\, (v_i,v_i') \in C \wedge \left(v_j,v_j'\right) \in C \wedge (v_i,v_j,l) \in r_{E1}\right\} \end{aligned} \tag{1}$$

$$\mathrm{sim}_C\,(G_1,G_2) = \frac{f\,(descr\,(G_1) \cap_C descr\,(G_2)) - g(splits(C))}{f\,(descr\,(G_1) \cup descr\,(G_2))}$$

Where *splits* is the set of vertices from $V_1 \cup V_2$ which are associated with 2 or more vertices by C. The total similarity value is the maximum similarity value of all the possible correspondences:

$$\mathrm{sim}\left(G_1, G_2\right) = \max_{C \subseteq V_1 \times V_2} \left\{\mathrm{sim}_C\left(G_1, G_2\right)\right\}$$

The complexity of this problem is, again, exponential in the number of vertices of the graphs being compared, but the use of heuristics and bounding functions can accelerate the search.

This approach is more sensible to the similarity of the labels in the edges. On the other hand, the possible values when comparing one label with another (whether it is a node or an edge label) can only express if they are identical or not. We need a way to compare, not only the shape of the behaviours but also their functionalities and, in the scenario we are dealing with, its uncommon to find two nodes or two edges which have exactly the same labels, so we will need some way to relax this comparison.

Third approach

The similarity measure proposed by Wang and Ishii in [12] is also based in the definition of correspondence relations between the nodes of the two graphs.

This method doesn't use the intersection, but an algebraic formula to obtain the final similarity measure. As in the previous approach, the similarity degree of two graphs G_1 and G_2 is the maximum similarity of G_1 and G_2 over all the possible correspondences:

$$\mathrm{sim}\left(G_1, G_2\right) = \max_{C} \left\{\mathrm{sim}_C\left(G_1, G_2\right)\right\}$$

and the similarity of G_1 and G_2 over the correspondence C

$$\begin{aligned}
\mathrm{sim}_C\left(G_1, G_2\right) &= \frac{F_n + F_e}{M_n + M_e} \\
F_n &= \sum_{n \in V_1} \frac{W\left(n\right) + W\left(C\left(n\right)\right)}{2} \cdot \mathrm{sim}\left(n, C\left(n\right)\right) \\
F_e &= \sum_{e \in E_1} \frac{W\left(e\right) + W\left(C\left(e\right)\right)}{2} \cdot \mathrm{sim}\left(e, C\left(e\right)\right) \\
M_n + M_e &= \max\left(\sum_{n \in V_1} W\left(n\right), \sum_{n \in V_1} W\left(C\left(n\right)\right)\right) + \max\left(\sum_{e \in E_1} W\left(e\right), \sum_{e \in E_1} W\left(C\left(e\right)\right)\right)
\end{aligned}$$

where W is the weight of a node or an edge.

For this approach, the labels in the nodes and edges are single variables or constants, and their similarity is defined by the following functions:

- For nodes, if the value represented for the constant or variable in both nodes is the same, then the similarity is 1, and 0 in any other case.

– For edges, if the source and target nodes of the edges are related by C and the labels are equal, then the similarity is 1; if the labels are different, the similarity is 0.5 and is 0 in any other case.

In this case we can change this similarity function so we can obtain a more descriptive value. We use a functionality based similarity function (Section 4.1) to compare the descriptors of the nodes. As with the previous techniques, the complexity of this one is also exponential and its also possible to reduce the search space by the use of heuristics and bounding techniques.

Our approach

Our approach to the similarity problem in finite state machines is based in both the structure of the state machine and the labeling in the nodes. The labels associated to the nodes are used to express the functionality of the behaviours contained in them.

In our implementation we allow the user to select any of the three techniques explained before to obtain the similarity measure in the structure based retrieval.

First approach

This approach is based in the calculation of the edit distance between two graphs. The distance is obtained as the sum of the operations needed to transform one graph into the other.

The cost assigned to each edit operation determines the final distance. In our approach, we are considering the costs of edit operations, not as constants, but as functions defined over the source and target nodes or edges. This way, we can express the intuitive idea that changing one label for another is cheaper in cost if the labels are more similar. For instance, the cost of the edit operation $E(C)[\text{Dribble} \rightarrow \text{Defend}]$ is:

$$cost(E(C)[\text{Dribble} \rightarrow \text{Defend}]) = C_E \cdot (1 - \text{sim}(\text{Dribble}, \text{Defend}))$$

where Dribble and Defend are the labels of the nodes (actually, the labels are the functional descriptors of the behaviours, but we used these descriptive names to simplify the example) and the sim function is the similarity function used in functionality based retrieval in Section 4.1.

We also impose the following restrictions on the possible values of the cost functions, so the results of the distance function are reasonable:

1. $C_E \leq C_A + C_D$ and $C_{E'} \leq C_{A'} + C_{D'}$
   This means that editing the label of a node is cheaper than an addition and a deletion of the same node with different labels.
2. $C_A = C_D$ and $\text{sim}(X, Y) = \text{sim}(Y, X)$
   These two restrictions give symmetry to our distance measure.

For instance, to obtain the similarity between the query and the case in Figure 3, if we use the costs $C_A, C_D, C_E, C_{A'}, C_{D'}, C_{E'} = 1$, and the sequences:

$$S_1 = \{D(A), D(C)\}$$
$$S_2 = \{D(A), D(B), E(C)[\text{Dribble} \rightarrow \text{Defend}], A'(D, C)\}$$
$$S_3 = \{E(A)[\text{Go to my goal} \rightarrow \text{Goal shot}], D(C), D(D), A'(B, A)\}$$

The distances are:

$$d_1 = 2 \cdot C_D = 2$$
$$d_2 = 2 \cdot C_D + C_E \cdot (1 - \text{sim}(\text{Dribble}, \text{Defend})) + C_{A'} = 2 + 0.5 + 1 = 3.5$$
$$d_3 = 2 \cdot C_D + C_E \cdot (1 - \text{sim}(\text{Go to my goal}, \text{Goal shot})) + C_{A'} = 2 + 1 + 1 = 4$$

As we can see, the result of d_2 is better than d_3 because the labels *Dribble* and *Defend* are more similar than *Go to my goal* and *Goal shot*.

Second approach

This approach is based in the definition of a correspondence between the nodes of the query and the case graphs.

As has been seen in equation (1), in page 9, the intersection with respect to a correspondence C only takes into account the nodes and edges who share identical labels. In the case of finite state machines, it is convenient to consider a more relaxed similarity measure, so we can take into account the nodes that are not equal but similar. To address this problem we add a value β to each tuple in the intersection. This value represents the similarity between the labels of the nodes or edges:

$$\begin{aligned} &descr\,(G_1) \cap_C descr\,(G_2) = \\ &\quad \{(v, v', \beta) \mid (v, v') \in C \wedge (v, l) \in r_{V1} \wedge (v', l') \in r_{V2} \wedge \beta = \text{sim}\,(l, l')\} \cup \\ &\quad \{\big((v_i, v_j), (v'_i, v'_j), \beta\big) \mid (v_i, v'_i) \in C \wedge \big(v_j, v'_j\big) \in C \wedge (v_i, v_j, l) \in r_{E1} \wedge \\ &\quad \big(v'_i, v'_j, l'\big) \in r_{E2} \wedge \beta = \text{sim}\,(l, l')\,\} \end{aligned}$$

$$\text{sim}_C\,(G_1, G_2) = \frac{f\,(descr\,(G_1) \cap_C descr\,(G_2)) - g(splits(C))}{F}$$

The similarity function we use is the functionality based retrieval similarity (Section 4.1).

The similarity value β is used by the function f to obtain the final similarity value, and the constant F is an upper bound of f that maintains the result in the interval $[0, 1]$. For instance, considering the example in figure 3, and the functions:

$$f(I) = \sum_{\text{for each node n in I}} (f_N(n)) + \sum_{\text{for each edge e in I}} (f_E(e))$$
$$f_N((v, v', \beta)) = \beta$$
$$f_E(((v_i, v_j), (v'_i, v'_j), \beta)) = \beta$$
$$g(S) = |S|$$
$$F = \max\{|r_{V1}|, |r_{V2}|\} + \max\{|r_{E1}|, |r_{E2}|\} = 4 + 6 = 10$$

we can have the following similarity values:

– for $C = \{(A,1),(B,1),(C,2),(D,2)\}$:

$$\begin{aligned} descr(G_1) \cap_C descr(G_2) = \{&(A,1,0.5),(B,1,1),(C,2,0.5),(D,2,1),\\ &((B,C),(1,2),1),((B,D),(1,2),1),\\ &((C,B),(2,1),1),((D,B),(2,1),1)\} \end{aligned}$$

$$splits(C) = \{(1,\{A,B\}),(2,\{C,D\})\}$$

$$sim_C(G_1,G_2) = \frac{(3+4)-2}{10} = 0.5$$

– for $C = \{(A,1),(B,\emptyset),(C,1),(D,2)\}$:

$$descr(G_1) \cap_C descr(G_2) = \{(A,1,0.5),(C,1,0.5),(D,2,1),\ ((C,D),(1,2),1)\}$$

$$splits(C) = \{(1,\{A,C\})\}$$

$$sim_C(G_1,G_2) = \frac{(2+1)-1}{10} = 0.2$$

To simplify this approach, we can consider only the nodes and edges whose β is greater than a certain threshold.

Third approach

The third approach is also based in defining the possible correspondences between the graphs being compared. In this case, the calculation includes the comparison of the similarity of labels. To adapt it to our scenario we use the functionality based retrieval similarity function, instead of the one proposed.

As a first approach we give all the nodes and edges the same weight (1). The resulting similarity measure is:

$$\begin{aligned} \mathrm{sim}_C(G_1,G_2) &= \frac{F_n + F_e}{M_n + M_e}\\ F_n + F_e &= \sum_{n \in N_1} \mathrm{sim}(n, C(n)) + \sum_{e \in E_1} \mathrm{sim}(e, C(e))\\ M_n + M_e &= |N_1| + |E_1| \end{aligned}$$

For the example in figure 3 we can have the following results:

– for $C = \{(A,1),(B,1),(C,2),(D,2)\}$:

$$sim_C(G_1,G_2) = \frac{(0.5+1+0.5+1)+(1+1+1+1)}{4+6} = 0.8$$

– for $C = \{(A,1),(B,2),(C,1),(D,2)\}$:

$$sim_C(G_1,G_2) = \frac{(0.5+0+0.5+1)+(1+1+1+1)}{4+6} = 0.6$$

5 Related Work

There exist several tools oriented towards the edition of finite state machines. Most of them are general purpose state machine editors (like Qfsm or FSME), which allow a more or less elastic definition of the inputs and outputs (the sensors and actuators) and the generation of the source code corresponding to the state machine in one or more common languages like C++ or Python. Most of them don't allow the use of HFSMs, nor facilitates the use of CBR or some other tool to favour reusing the state machines.

Regarding game editors, most of them are only applicable to one game or, at the most, to the games implemented by one game engine (as is the case of the Valve Hammer Editor). Besides, the vast majority only allow map edition. The few that allow editing the entity behaviors are usually script based, like the Aurora Toolset for Neverwinter Nights.

Finally, there exist some tools like BrainFrame and, its later version, Simbionic, which are game oriented finite state machine editors. These editors allow the specification of the set of sensors and actuators for the game and the edition of HFSMs using that specification. The HFSMs generated by the editor are interpreted by a runtime engine that must be integrated with the game. Currently, there exist a C++ and a Java version of the runtime engine. There are two crucial differences between our approach and the approach used in Simbionic. First of all, the Simbionic editor doesn't offer any assistance for reusing the behaviors, like the CBR approximate search engine integrated into the eCo editor. And second, to integrate a behavior edited with the Simbionic editor with a game, it is mandatory to integrate the Simbionic runtime engine with the game. On the other hand, the eCo editor can generate the source for behaviors in any language, provided we have implemented the appropriate code generator. Besides, it can generate any kind of file, like image captures, summaries of the behaviors or text files.

6 Conclusions and Future Work

In this paper we have described an ongoing work using CBR to design intelligent behaviors in videogames. We have developed a graphical editor based on HFSM that includes a CBR module to retrieve and reuse stored behaviors.

One of the main advantages of our approach is that the editor and the CBR module are generic and reusable for different games. We have shown the applicability in a soccer simulator environment (SoccerBots) to control the behavior of the players. As part of the testing stage and to check the editor applicability we have proposed the integration of the eCo editor with other games with very different nature: SoccerBots is a sports simulator, Neverwinter Nights is a role playing computer game, JV^{2}M [9] is an action game and AIBO is a real life multipurpose robot) and with different integrating characteristics. For instance, while in JV^{2}M we define the set of sensors and actuators, it is fixed for the other environments; while Neverwinter Nights is highly event-oriented, the rest of the

environments are basically reactive systems. The eCo behavior editor has been easy to use in the different environments and offers a friendly interface. The editor assists the user in the definition of new behaviors through a CBR module that retrieves previously stored behaviors.

In this paper we have described the current state of the work but there are many open lines of work. We have finished the graphical editor, defined the structure of the cases and the game models, and we have been working on case representation, storage and similarity based retrieval. Current lines of work are automatic reuse of behaviors and learning.

By now, the adaptation process is carried out manually by the user, who receives some assistance from the system. The system evaluates the differences between the values of the attributes in the query and the retrieved case and use them to indicate what nodes should be modified.

In the current version, the learning of the CBR system is totally user guided: the user indicates which cases must be stored in the case base and also enters the values for the descriptors. The set of values for each descriptor is a very subjective matter, so it would be a good idea to automatize this process or make the system suggest some suitable values, using machine learning approaches.

Regarding structure based similarity, we have proposed three different approaches to compare finite state machines. Our next step in this issue will be testing them to determine which is the most suitable approach and for what kind of cases.

The use of HFSM offers many possibilities to reuse and combine pieces of behaviors within other more complex behaviors. We are also working on the definition of an ontology about different games genres to be able to reuse behaviors, vocabulary and sets of sensors and actuators between different games of the same genre. This way we can promote the reuse of behaviors, even among different games, while making easier the use of the editor, since the user doesn't need to learn the characteristics of the game model for each game.

There exist numerous techniques, besides HFSMs, to represent behaviors, like decision trees, rule based systems, GOAP or Hierarchical Task Networks, for instance. One of the opened investigation lines is the study of the pros and cons of each one of them and the possibility of combining some of them to create the behaviors (for instance, a HFSM in which the nodes were specified by rule systems, Hierarchical Task Networks or some other technique).

Within more complex and human-like current techniques that are used for controlling game AIs (such as big C functions or finite-state machines) will not scale up. But, just as computer game graphics and physics have moved to more and more realistic modeling of the physical world, we expect that game developers will be forced into more and more realistic modeling of human characters.

References

1. M. Bowling, J. Fürnkranz, T. Graepel, and R. Musick. Machine learning and games. *Machine Learning*, 63(3):211–215, Junio 2006.

2. J. Brownlee. Finite state machines (fsm). Available from http://ai-depot.com/FiniteStateMachines/FSM.html (accessed March 14, 2008).
3. H. Bunke and B. T. Messmer. Similarity measures for structured representations. In *EWCBR '93: Selected papers from the First European Workshop on Topics in Case-Based Reasoning*, pages 106–118, London, UK, 1994. Springer-Verlag.
4. A. J. Champandard. *AI Game Development - Synthetic Creatures with Learning and Reactive Behaviors*. New Riders Games, 2003.
5. P. A. Champin and C. Solnon. Measuring the similarity of labeled graphs. In K. D. Ashley and D. G. Bridge, editors, *5th Int. Conf. On Case-Based Reasoning (ICCBR 2003)*, LNAI, pages 80–95. Springer, June 2003.
6. G. Flórez Puga and B. Díaz-Agudo. Semiautomatic edition of behaviours in videogames. In *Proceedings of AI2007, 12th UK Workshop on Case-Based Reasoning*, dec 2007.
7. D. Fu and R. Houlette. Putting ai in entertainment: An ai authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, 2002.
8. A. Girault, B. Lee, and E. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design*, 18(6):742–760, Junio 1999. Research report UCB/ERL M97/57.
9. P. P. Gómez-Martín, M. A. Gómez-Martín, and P. A. González-Calero. *Javy: Virtual Environment for Case-Based Teaching of Java Virtual Machine*, volume 2773 of *Lecture Notes in Artificial Intelligence, subseries of LNCS*, pages 906–913. Springer Berlin / Heidelberg, 2003.
10. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2007. To appear.
11. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
12. Y. Wang and N. Ishii. A method of similarity metrics for structured representations. *Expert Systems with Applications*, 12(1):89–100, 1997.

6.6. Query Enabled Behaviour Trees

Cita completa

Gonzalo Flórez Puga, Marco Antonio Gómez Martín, Pedro Pablo Gómez Martín, Belén Díaz Agudo y Pedro Antonio González Calero. *Query-Enabled Behavior Trees.* IEEE Transactions on Computational Intelligence and AI in Games. Volume 1, Issue 4. Diciembre, 2009. P. 298–308.

Resumen original de la contribución

Artificial intelligence in games is typically used for creating player's opponents. Manual editing of intelligent behaviours for Non-Player Characters (NPCs) of games is a cumbersome task that needs experienced designers. Our research aims to assist designers in this task. Behaviours typically use recurring patterns, so that experience and reuse are crucial aspects for behaviour design. The use of hierarchical structures like Hierarchical state machines, Behaviour Trees, or Hierarchical Task Networks, allows working on different abstraction levels reusing pieces from the more detailed levels. However, the static nature of the design process does not release the designer from the burden of completely specifying each behaviour. Our approach applies Case-Based Reasoning (CBR) techniques to retrieve and reuse stored behaviours represented as behaviour trees. In this paper we focus on dynamic retrieval and selection of behaviours taking into account the world state and the underlying goals. The global behaviour of the NPC is dynamically built at run time querying the CBR system. We exemplify our approach through a serious game, developed by our research group, with gameplay elements from First-Person Shooter (FPS) games.

Referencia de citas bibliográficas

Rabin (2006), Rabin (2008), Isla (2005), Isla (2008), Krajewski (2009), Gorniak y Davis (2007), Remo y Sheffield (2008), D. Leake (1996), Gilgenback y McIntosh (2006), Borovikov y Kadukin (2008), Harel (1987), Houlette et al. (2001), Atkin et al. (2001), Baader et al. (2003), Gómez-Martín et al. (2007), Flórez Puga et al. (2008), Flórez-Puga et al. (2008), Miles et al. (2004), Aleson y Louis (2006), Olenderski et al. (2006), Nicolescu y Matarić (2002), Mateas y Stern (2002), Loyall y Bates (1991), Ontañón et al. (2008), Hoa (2005), Gorniak y Davis (2007), Ontañón et al. (2008)

Query-Enabled Behavior Trees

Gonzalo Flórez-Puga, Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín, Belén Díaz-Agudo, and Pedro Antonio González-Calero

***Abstract*—Artificial intelligence in games is typically used for creating player's opponents. Manual editing of intelligent behaviors for nonplayer characters (NPCs) of games is a cumbersome task that needs experienced designers. Our research aims to assist designers in this task. Behaviors typically use recurring patterns, so that experience and reuse are crucial aspects for behavior design. The use of hierarchical structures like hierarchical state machines, behavior trees (BTs), or hierarchical task networks, allows working on different abstraction levels reusing pieces from the more detailed levels. However, the static nature of the design process does not release the designer from the burden of completely specifying each behavior. Our approach applies case-based reasoning (CBR) techniques to retrieve and reuse stored behaviors represented as BTs. In this paper, we focus on dynamic retrieval and selection of behaviors taking into account the world state and the underlying goals. The global behavior of the NPC is dynamically built at runtime querying the CBR system. We exemplify our approach through a serious game, developed by our research group, with gameplay elements from first-person shooter (FPS) games.**

***Index Terms*—Behavior trees (BTs), case-based reasoning (CBR), first-person shooter (FPS), nonplayer characters (NPCs).**

I. Motivation

With graphics in video games coming close to photo realistic quality, and multiprocessor architectures becoming common in console and PC game platforms, sophisticated AI has become the focus of the video game industry as the next big thing for enhancing the player experience, while profiting from the number of spare central processing unit (CPU) cycles available in modern hardware. For that reason, industry is growing more interested in academic research in AI to provide rich, robust, and scalable techniques for controlling nonplayer characters (NPCs) and richer narrative schemes in games.

Nevertheless, in spite of the reciprocal interest from academia to demonstrate the applicability of AI research into commercial games, there still exists a big gap between the state of the practice for commercial games and the state of the art in academic AI within research areas potentially relevant to commercial games such as decision making, agent coordination, machine learning, or data mining.

Manuscript received June 22, 2009; revised October 02, 2009; accepted October 29, 2009. First published November 10, 2009; current version published January 08, 2010. This work was supported by the Spanish Ministry of Science and Education (TIN2006-15202-C03-03).

The authors are with the Universidad Complutense de Madrid, Madrid 28040, Spain (e-mail: gflorez@fdi.ucm.es; marcoa@fdi.ucm.es; pedrop@fdi.ucm.es; belend@sip.ucm.es; pedro@sip.ucm.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2009.2036369

In this paper, we propose an extension to behavior trees (BTs), a popular technology for controlling NPCs in modern video games. First, we consider that taking a technique used in industry and extending it with academic results is a more realistic way of closing the industry-academia gap than proposing full replacements for industry-trusted techniques. Second and more importantly, this approach allows us to focus on a key problem when building the AI for a game: the point is not the AI but the player experience, and this has to be authored by a game designer, an expert on game narrative that usually is not proficient at programming computers.

According to the number of papers dedicated to the subject in the editions 3 and 4 of the AI Game Programming Wisdom book series [1], [2], BTs are the technology of choice for designing the AI of NPCs in different game genres. BTs have been proposed as an evolution for hierarchical finite-state machines (HFSM) intended to solve finite-state machine (FSM) scalability problems by emphasizing behavior reuse [3]. In BTs, explicit transitions from state to state are substituted by a number of predefined procedural mechanisms that allow computing the next state. In spite of the added complexities of designing behaviors with implicit transitions, BTs, which were first introduced in game industry as a tool for programmers, have been successfully used by professional game designers in released commercial games [4], [5].

BTs are goal structures that represent how a high-level goal can be decomposed into lower level ones until reaching primitive goals that can be achieved by available actions. In this sense, BTs resemble hierarchical task networks (HTNs) used for planning, although their purpose is totally different. HTNs are used to *generate* plans while BTs are used to *store* and *execute* plans. BTs can be seen as AND-OR trees that store all possible plans that a game entity (basically an NPC or a group of NPCs) can follow to obtain its goals. Although some previous work has been dedicated to applying planning techniques, specially HTNs, for the runtime generation of plans controlling NPC actions [6], [7], the number of practical applications of such approaches that take the human designer out of the loop are greatly outnumbered by others where the designer has a tighter control of the player experience [8].

We propose to extend BTs with simple and controllable runtime planning capabilities taking ideas from case-based reasoning (CBR) [9]. Instead of just storing a complete BT for every type of NPC in a game (the BT for an ogre is different from that of a goblin), we propose to store behavior subtrees designed to achieve particular goals. This way, when building the full BT for an NPC type we can reuse subtrees previously designed. Furthermore, the selection of the particular subtree to reuse can be deferred to runtime. For runtime selection to

work, every reusable subtree (i.e., every case) should be described with the goal it is intended to achieve. Our extension to BTs is called query behavior trees (QBTs) and it proposes a new type of tree nodes representing queries. At design time, designers can choose to include within the BT a query to retrieve a subtree to achieve a particular goal. The query will retrieve at runtime the most appropriate subtree, given the actual state of the game.

Notice that query-enabled BTs can support the runtime generation of the complete BT by just using a query node as the root. Nevertheless, most game designers will prefer to include query nodes in lower levels of the tree, where behavior variability is to be preferred over fine-grained control. Actually, the main benefit of our approach is to provide behavior variability for a particular instance or between different instances of an NPC type, without increasing the authoring effort. Additionally, query-enabled BTs can be automatically updated to incorporate actions that were not created when the tree was designed.

The rest of this paper runs as follows. Section II describes the particular type of BTs that we extend through the mechanisms described in Section III. Section IV steps through a detailed example while Section V estimates the benefits of the proposed extension. Section VI presents related work and concludes the paper.

II. Behavior Trees at a Glance

FSMs have been the technology of choice for AI in games for decades. FSMs are easy to program, fast to execute, and game designers feel comfortable using them. Unfortunately, FSMs do not scale well when the NPC's AI becomes too complex, resulting in a combinatorial explosion of transitions. FSMs do not easily allow either for adding and removing states, or reusing states in different FSMs. For example, if a new AI type can shoot lasers out of its eyes, transitions need to be explicitly added from all the states in which it is valid to go into that state [10].

Two ideas are used in BTs to overcome the scalability problems in FSMs: using procedural mechanisms to determine transitions, turning FSMs into behavior lists; and introducing levels of abstraction, turning behavior lists into BTs.

Behavior *lists* represent the AI for the NPC as the list of states it can be on, providing every state with a condition to check whether the NPC can transit to it, and some algorithm to choose one when several states are runable. Since in this approach we remove transition checking from the state, we may say that the state is just determining what the NPC is doing when in that state, i.e., its *behavior*, and for that reason, in this approach, we do not say that the NPC is in a given state but instead consider that the NPC is executing a given behavior. In this way, to add the *shooting lasers out of its eyes* to an NPC, that state just has to be added into the NPC's list of states along with a condition that becomes true when that behavior can be chosen. Keep in mind that this condition could refer to changes in the NPC state, to player actions, or even to events triggered in the environment.

Although more complex types of behavior selection mechanisms are described in the literature, for the goals of this paper, we only require three of them: sequences, static priority list, and dynamic priority list. A sequence composite behavior executes its children in the order they are defined, succeeding when every child succeeds and failing otherwise. Children behaviors of a sequence are not guarded by conditions. A static priority list is a composite node that evaluates its children conditions in order and activates the first child whose condition is true. A dynamic priority list, in its turn, reevaluates the conditions of a child if it has a higher priority (the first child being the one with highest priority) than the active one, and switches to a higher priority child whenever possible.

The behavior selection mechanism has to be completed with an execution model that determines when to reevaluate guarding conditions for candidate behaviors. Typically, conditions are reevaluated after a given number of game ticks, when certain game events occur or when the active behavior terminates. If a node condition becomes false while its behavior is being executed, it is immediately aborted and fails. Notice that conditions associated with behaviors are not like preconditions that, when fulfilled, guarantee its successful completion, but like guards, as in abstract state machines [11], that indicate that the behavior can be chosen, although it may terminate with failure.

The second ingredient to overcome FSM limitations is that of hierarchy, taking FSMs into HFSMs and behavior lists into BTs. The idea of having abstract states that abstract a whole FSM was first proposed by Harel as part of his *statecharts* specification, a visual formalism, extending that of state diagrams (the visual formalism for FSM), to specify complex systems [12]. HFSMs use a stack to store active states, where only the topmost state represents executable behavior, and, at every cycle, possible transitions from the active states are evaluated [13]. Behavior lists can be extended to BTs by considering that any behavior in the list can be itself a composite behavior with a list of subbehaviors. BTs active states must be in a branch going from the root to a leaf in the tree (multiple branches if several basic actions can be executed at the same time). Notice that, depending on the context, the nodes in a BT can be seen as states, behaviors, or actions. In this context, "behavior" is a synonym of (transitionless) "state," while "action" corresponds to a primitive behavior than can only appear as a leaf in a BT.

The concept of hierarchy in BTs is crucial to overcome the scalability problem in FSMs because it introduces a hierarchy of *goals* that allows determining behavior based on reasoning at different levels of abstraction. Most actions have a primary goal along with a number of additional goals that depend on the action context [14]. For example, the primary goal of the action "move-to" is to change location from x to y, but in an urban fight scenario we can be moving to get under cover from enemy fire or to assist a fallen comrade. Having actions focus only on their primary goal can sometimes lead to unintelligent behavior. For example, if an agent is moving to a destination and is attacked, it will continue to move, even when it would be totally destroyed by doing so. Instead of adding conditional statements to every action that specify all the exceptions to normal behavior we can handle multiple goals and make them part of a hierarchy, which prioritizes goals higher up in the hierarchy, i.e., staying alive is more important than moving to point y, so if some condition higher up in the BT becomes activated for self protection, the whole branch being executed can be pruned.

BTs provide designers with an abstraction that allows them to treat a tree as a new complex behavior implementation that may be attached to other more general BTs. In that sense, during the development phase, designers create a collection of behaviors in the form of BTs, which are later attached to the branches of different BTs for more than one entity. In order to improve the reusability, BTs may be parameterized. For example, designers may build a BT for an enemy that attacks using an available weapon and picks an item up afterwards. Designers may include this BT by hard-coding the values of the parameters (the weapon and enemy) or by selecting these values through other behaviors executed before.

BTs represent the behavior of NPCs, so each BT in execution is associated with the NPC it "belongs" to. When a BT is executed, an execution context is created for it. The context of a BT is made up of a set of variables, each one containing a set of pairs (*attribute name*, *value*). The set of attributes in the context is the portion of the game state that can be accessed by the NPC. Each attribute referenced in the tree has to be present in the context. Likewise, the actions taken in the tree leaves can, sometimes, have an effect on the context, changing the value of its variables. Generally, a BT's context contains at least two variables:

- *?this:* references the NPC executing the behavior; the attributes of this variable describe its properties (e.g., aggressiveness, health, etc.);
- *?world:* references the virtual environment in which the game takes place.

To allow the exchange of information between a behavior and any of its children, a behavior can be associated with a set of input parameters. At the time of the invocation of the behavior, each input parameter is bound to a value, either a literal value or the value of an attribute in the context of the current BT. The value bound to the parameter is then included in the context of the newly invoked BT, and can be used there. The information exchange in the opposite direction (from children to parents) is done using the context variables that act in some way like global variables.

III. Query Nodes in Behavior Trees

Reusability and modularity are important advantages of using BTs. Each BT represents an abstraction that can be reused as a composing piece of other BTs. Different BTs are created independently during the game design phase and they can be assembled as pieces of other existing BTs. The collection of game BTs includes different ways of solving a certain goal, e.g. different ways of *getting food*. To make the process consistent and useful it is important to review the preexisting BTs that include a certain goal to check if it is convenient to assemble the new BT (representing a new way to solve a certain goal). As we discuss in Section VI, this consistency process generates an extra effort that is sometimes skipped. That means that the behaviors added in the late design phases are not taken into account by the behaviors that were included in the early design phases.

The approach presented in this paper proposes the use of case-based reasoning (CBR) as a dynamic way to generate behavior that prevents this problem. CBR is based on the intuition that new problems are often similar to previously encountered problems, and therefore, that past solutions may be reused, directly or through adaptation, in other situations. CBR systems typically apply retrieval and matching algorithms to a case base of past problem–solution pairs. Another very important feature of CBR is its coupling to learning. A strong effort has been done in the CBR community to solve the problems of similarity and adaptation in different contexts, with different approaches to case representation, organization and storage, and amount of knowledge, from knowledge-intensive to data-intensive approaches.

We propose a dynamic approach where the CBR system is queried at runtime to find the most appropriate behavior from a case base of implemented behaviors using behaviors trees. The CBR processes work with an updated behavior case base that allows retrieving the most convenient behavior according to a certain query using the whole collection of designed behaviors and avoiding the extra cost of prechecking its adequacy with other behaviors.

Complex behaviors are built by assembling simple behaviors that are combined to form a complex BT implementing this behavior. The first challenge is defining the knowledge of the system, mainly the vocabulary to define the case structure and the experience behavior case base. Then, we define the retrieval and reuse processes that deal with this knowledge.

Each case in the case base represents a behavior implementation using a BT that is described through a semantic label from a behavior ontology B, a set of variables, and a set of variable constraints. A behavior ontology classifies and allows the annotation of the individual behavior implementations that can be retrieved and reused. We use a basic classification of FPS oriented behaviors to manage resources, to confront other entities, either attacking or defending, to move or transport other entities, and to hunt or chase other entities. Each individual behavior can be implemented through primitive actions or through BTs.

A *query* is formalized in a *query node* and it would be essentially a partial description of a behavior implementation that may include the desired behavior, along with a number of variables and variable constraints using the same vocabulary used for describing applicability conditions of behavior implementations. The vocabulary is based on a behavior ontology that provides different *classes* used to categorize the behaviors. We used both *primitive classes* such as *annoy*, *move*, or *stock-up* and *defined classes* such as *hunt* or *harvest* that symbolize behaviors that can be defined using the primitive ones. Concrete behaviors are represented as *individuals* (instances of classes) in the ontology. Defined classes help us to automatically classify some of the behaviors added as instances to the ontology. Automatic concept and instance classification, as well as the similarity between behaviors in the hierarchy, rely on the subsumption mechanism defined in description logics systems for knowledge representation [15].

Apart from the *behavior ontology*, we have an *entity ontology*, which classifies the entities present in the game. Relations between concrete behaviors and classes in the behavior ontology are created to model the *behaviors' input parameters*. For instance, any *attack* behavior has a target of class *entity*, or the destination of a *move* behavior has to be a *location* entity. Using

these relations, the designers can build queries more adequate to their necessities.

Our proposal is to let the designer define the query using the attributes listed as follows.

1) *Header:* the header of the query shows the name of the intended behavior goals represented by this query node [e.g., *attack (?entity)*] and a short descriptive text.
2) *Domain:* is a class of the behavior *ontology*. The retrieved behavior has to belong to this ontology domain class (e.g. class *attack*).
3) *Parameters:* parameters are bound from the current context to the retrieved behavior context. The function of a parameter in the retrieved behavior is expressed using the relations defined in the ontology.
4) *Relevant descriptors:* here, the designer selects from the whole set of variables that describe the game state, which ones are relevant for the query and, therefore, should be used.
5) *Similarity:* using this parameter, the designer can set the importance of each descriptor in the calculation of the similarity between the query and the retrieved behaviors.
6) *Requery:* once the behavior has been retrieved and is running, there may occur changes in the game state that would make another behavior more suitable for the current situation. Using the *requery* parameters, the designer can specify the conditions or changes in the game state that should make the system repeat the query. Note that, although the query is done again, the results can be the same. In that case, the behavior being executed is not restarted.
7) *Selection:* it indicates how many behaviors are retrieved when the query is executed. For instance, for a value of 3, the query would return the three most similar behaviors. The first one in order of similarity would be executed in the first place, but, if it fails, execution continues with the second and third ones.

Section IV shows an example of a query defined using this set of attributes. Depending on the moment the query is executed, the game state will be different, and so will the results of the query.

The set of attributes used to *define the cases* are very similar to those used to describe the query as they will be compared during the retrieval phase.

1) *Header*: it is composed of a *case number*, used to identify the case in the case base and a natural language *description* that describes the behavior represented by the case.
2) *Classes*: the behavior represented by a case can belong to one or more classes from the behavior ontology. This parameter enumerates them.
3) *Parameters*: it is the set of parameters received by the behavior, along with the restrictions of type of each one of them. The type is built from the classes in the ontology which an individual belongs to.
   The case can only be retrieved if the type of the parameter in the query is compatible with the type of the parameter in the case description.
4) *Descriptors*: it is a set of restrictions declared over the game state variables. The values of the descriptors can be symbolic or numeric.

The retrieval process consists of obtaining the most appropriate BT from the case base, based on the query data. To achieve this goal, we compare the attributes from the query with the attributes describing the behaviors trees in the case base using a similarity function. Given a query Q and a case from the case base C, the similarity value is obtained as follows:

$$\mathrm{sim}(Q,C) = \begin{cases} \bullet\ Q.\text{domain} \not\sqsupseteq C.\text{class} \Rightarrow 0 \\ \bullet\ \text{The restrictions on parameters in } Q \text{ do not hold in } C \Rightarrow 0 \\ \bullet\ \text{Otherwise} \Rightarrow \mathrm{sim}_{\mathrm{atr}}(Q,C) \end{cases}$$

$$\mathrm{sim}_{\mathrm{atr}}(Q,C) = \sum_{d \in D(Q,C)} w_d \cdot \mathrm{sim}_{\mathrm{loc}}(Q_d, C_d)$$

$$D(Q,C) = Q.\text{descriptors} \cap C.\text{descriptors}$$

$$\mathrm{sim}_{\mathrm{loc}}(Q_d, C_d) = 1 - \frac{|Q_d.\text{value} - C_d.\text{value}|}{\text{size}_d}$$

$D(Q,C)$ is the intersection of the sets of descriptors of Q and C and size_d is the size of the interval of valid values for a descriptor d. The weights w_d are the weights specified in the query, under the section *similarity*.

IV. Example

To show the advantages of our *query node*, we will detail an example where a BT will be built in the usual incremental way used in game production. This process is revealed as repetitive and, sometimes causes incoherences. In comparison, the use of our query node avoids these problems using a preclassification of the behaviors and delaying their selection until runtime. The BTs are extracted from JV^{2}M,[1] a serious game developed to teach the inner workings of the Java Virtual Machine, using gameplay elements from first-person shooters (FPSs) [16].

Fig. 1 shows the BT corresponding to the goal *steal resource from the weakest player*. In this case, the execution context is composed of three variables: *?this* and *?world*, which were mentioned in Section II, and *?player*, which refers to the human player who is competing against the computer.

This goal is expanded to another two subgoals that are executed in sequence. In the first place, the goal *select entity* is invoked. This goal is fulfilled by the atomic action of the same name. This atomic action tries to locate an entity belonging to the category and class specified by the input parameters. If it fails, then the whole *steal resource from the weakest player* goal fails, propagating this failure outside to the invoker behavior. Otherwise, it updates the value of the variable *?this.target* to the identifier of the selected entity and the execution continues to the next goal *steal resource*.

Steal resource has an input parameter *entity*, which is bounded at this point to the variable *?this.target*. This way, the goal *steal resource* receives the target entity from which it should try to steal the resources: the entity selected by *select entity*.

Steal resource is expanded to a new sequence of three goals: *search*, *attack*, and *pickup nearest resource*. *Search* tries to locate the entity in the game world. If it succeeds, *attack* will try to eliminate the target—it can be done in several different ways, as we will see. Last, *pickup nearest resource* tries to collect the

[1]http://www.gaia.fdi.ucm.es/projects/javy/

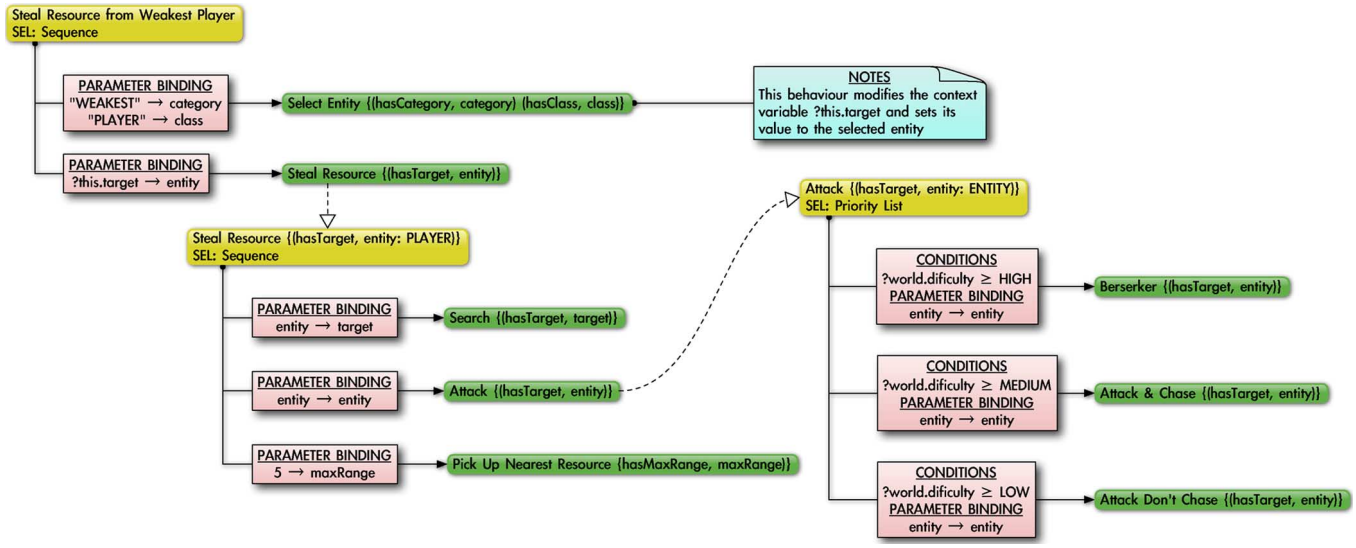


Fig. 1. Behavior tree for *steal resource from the weakest player*.

fallen resources, if any. If any of these goals fail, the goal *steal resource* fails, making *steal resource from the weakest player* also fail.

The *attack* goal is expanded to a priority list made up of three subgoals, each one with its own execution conditions. This means that *attack* can be solved in three different ways. The execution conditions indicate that the subgoals are selected depending on the difficulty level of the game. The subgoal with the higher priority is *berserker*, solved by the action *berserker*, which is a very aggressive and strong attack, and is executed in the first place. If this subgoal fails or the execution condition *?world.difficulty* $\geq$ *HIGH* does not hold, the action that solves the next subgoal in the list is executed: *attack and chase*. The entity that executes this action will try to attack a target and, if the target runs away, will chase it. If this subgoal also fails, the next one executed is *attack do not chase*, which attacks the target entity but does not chase it if it tries to escape. If all three subgoals fail, the *attack* goal fails, and this fail is propagated up the BT. On the other side, if any of them succeeds, then *attack* also succeeds.

The obtained *attack* tree only takes into account the parameter *?world.difficulty* to choose the way the entity executing the behavior is going to attack. Results would be more accurate if some other parameters were used to decide how to attack. For instance, if the health of the NPC executing the behavior is low it will be better to use a safer attack, even though the difficulty level is high. What we can do is to add a new reference to a more suitable goal—*attack do not chase*—before *berserker*. In this way, we are giving more priority to the newly added goal. We also have to add an execution condition to prevent executing the new goal if the health is not low.

Now, suppose that, once we have implemented *attack*, a new atomic action is developed by the programmers' team, *stealth attack*, that consists of approaching a target from behind and attacking it without being noticed. This action that would fulfill a new goal called also *stealth attack* would fit as a new way to complete *attack*. It would be applicable for a medium difficulty level and if the health was medium or low. It is suitable for entities that are not very aggressive but is not suitable if the entity executing it has been discovered. To check these conditions, we use the parameters *?this.aggressive* that measures the aggressiveness level of the entity and *?this.underattack*, a boolean that checks if the entity perceives any attacker nearby.

Our designers found this new behavior interesting, and decided to revise the previous *attack* BT. Fig. 2 shows the result, where, in order to maintain the priorities of the existing subgoals, several references to *stealth attack* have been added with different execution conditions.

When the *attack* goal is expanded, the subgoal with the highest priority is *attack do not chase*. If the health of the entity executing it is not low or if its execution fails, the next goal *stealth attack* is checked. *Stealth attack* will be executed at this point if the NPC is not very aggressive, if the level of difficulty is medium or high, and if the NPC has not been spotted yet. On failing of any of these conditions, or of the subgoal itself, *berserker* will be executed if the difficulty level is high. Otherwise, if the aggressiveness is low and the NPC has not been detected, the subgoal *stealth attack* will be executed next. If everything else fails, the difficulty is checked again and, if it is low, *attack* will be solved by the subgoal *attack and chase*. If it is not, *stealth attack* will be checked again. Last, in the case of all the former conditions and goals failed, subgoal *attack do not chase* is executed.

As we can see, with the addition of new subgoals and parameters, the complexity of the behavior grows greatly, along with the number of subgoals. The resulting behavior can be very difficult to interpret by a designer.

Another problem is that, if we include new subgoals during the development process, we have to modify the parent BT, adding the new subgoal and all the necessary references to the existing subgoals to maintain the intended priority order, like we did with *attack* in this example. Being *attack* a *priority list node*, the condition of each child is, in fact, made up of its own node condition (shown in the figures) and the negated conditions of all the preceding nodes, because all of them must be false in order to reach the current node in the first place. An undesirable

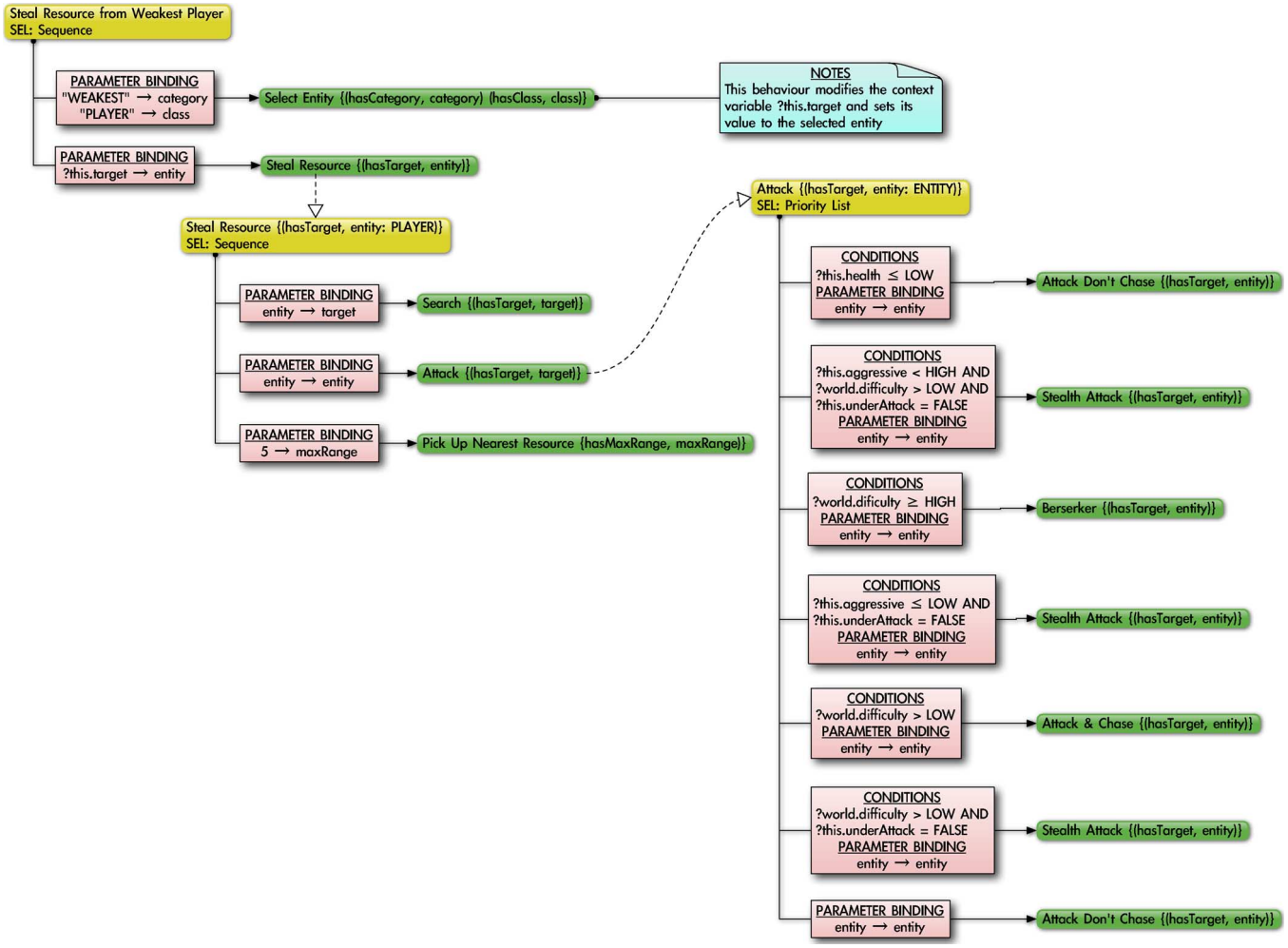


Fig. 2. Behavior tree for *steal resource from the weakest player* with the new subbehavior *stealth attack*.

consequence of the complexity in the conditions could be that some of the actions never get to execute, because of the accumulation of conditions as we go down the tree. As an attentive reader may have noticed, an example of this kind of conflict can be seen in the last reference to *stealth attack* in Fig. 2.

In query nodes, we use CBR techniques to retrieve behaviors from a domain, according to a set of parameters specified by the designer. These parameters describe the behavior that should be retrieved and attached to the same place where the query is. The system tries to find behaviors whose description is similar to the one in the query.

Fig. 3 shows an example behavior with a query node. The domain from where the behaviors must be retrieved is specified by means of a behavior *ontology* described in Section III.

When the execution flow reaches the query node *attack*, it has to be expanded into a BT or an atomic behavior. The expansion implies executing the query associated and retrieving the behaviors from the case base. The retrieved behaviors' parameters are bound to the values in the context (in this case, the variable *entity* from *steal resource* is bound to the input parameter *?entity* of the retrieved behavior) and executed.

Depending on the moment the query is executed, the game state will be different, as can the results of the query. Table I shows two hypothetical snapshots of the game state in two different instants. The increment in the value of the variables, shown in the third column, is used to check the requery conditions, as explained in Section III.

For this example, we use a small sample of the case base. Table II gathers the cases and descriptions we are using. The results of the similarity measure calculated as explained in Section III are shown in Table III. Results show how changes in the values of the parameters that describe the game state at different instants can lead to changes in the most suitable behavior.

For instance, if run under the conditions in Table I, the execution trace for a classic BT like the one in Fig. 2 would be the one shown in Table IV. The behavior executed at instant t_0 as an expansion of *attack* is *stealth attack*. On the other hand, if we use the query node in the same place and game state, the behavior retrieved, as shown in Table III, is C_4 that is *stealth attack*. The same thing happens at t_1 with behavior *attack and chase*.

V. Discussion

Query nodes can save development effort by being expanded at design [17] or at runtime [18]. As was stated in the paper, during the game production, *designers* create BTs mixing the basic behaviors with the aggregation nodes described in Section II. At the same time, *developers* create new basic

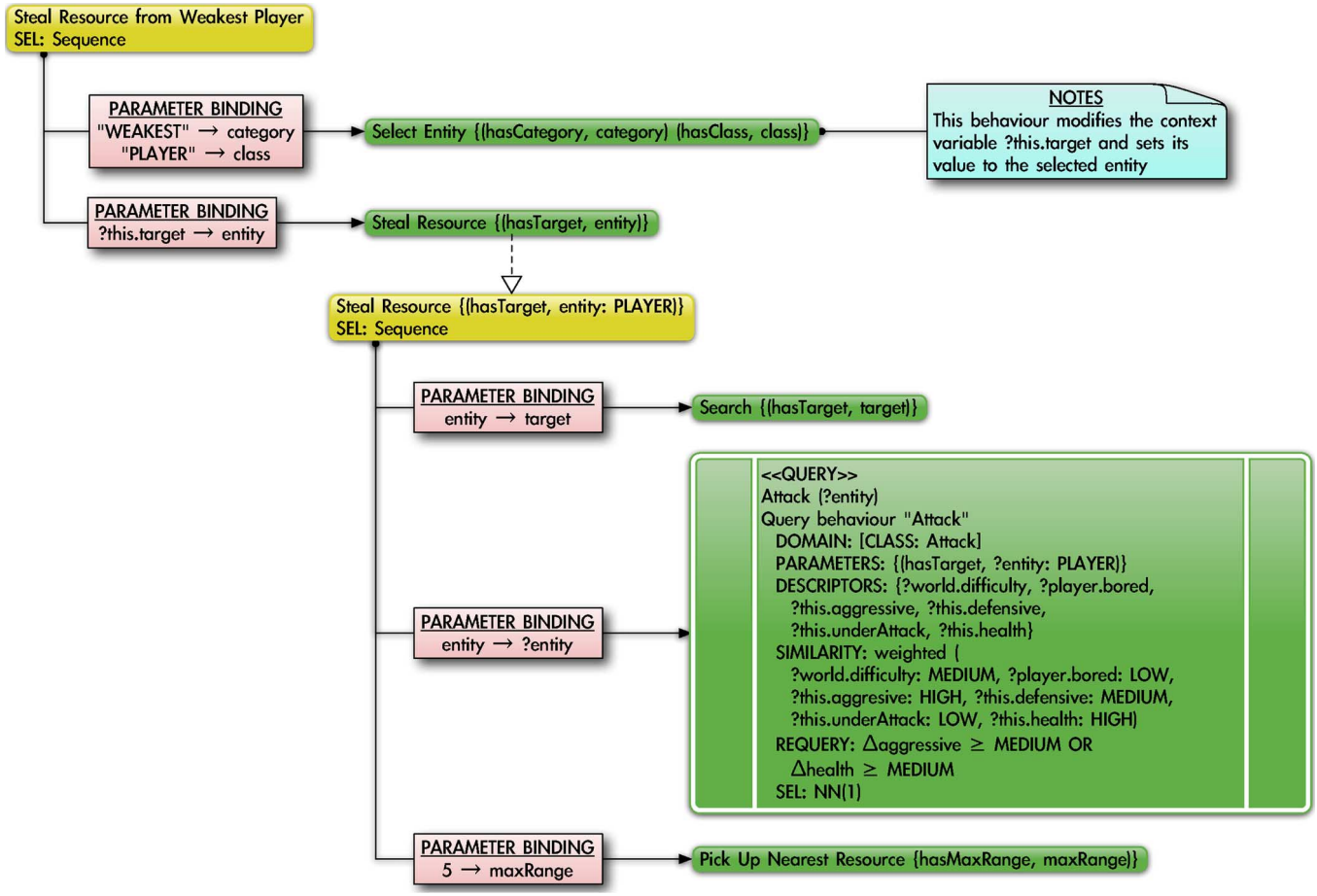


Fig. 3. Behavior tree for *steal resource from the weakest player* using a query node.

behaviors depending on the ongoing necessities (the *stealth attack* of our example would be one of them). As a result, designers will have more basic behaviors to play with at the end of the production, and the last created BTs will be richer than the first ones.

The *ad hoc* solution for this consistence problem is to revise the older BTs for detecting if they could be improved using the more recent basic behaviors created by the development team. Unfortunately, this revision effort needs a lot of time and should be performed during all the game production timeline.

Using our *query nodes*, on the contrary, old BTs automatically benefit from new behaviors if they are correctly stored and annotated in the case base. The example has shown that, when using our technique in the *attack* node, no revision is needed if a new *stealth attack* behavior is developed.

The main advantage of our proposal is that the number of *basic behaviors* can grow throughout all the game development and, even so, be quite sure that they will be used in older complex behaviors. Having this confidence when using static BTs requires a manual revision of the previously developed BTs, something only affordable if the number of added behaviors is kept low. Consequently, our proposal provides a better scalability for the *growth of basic behaviors*.

The cost for this saving is, obviously, categorizing each new basic behavior in order for the query node to recover it in the correct moments. Behavior and entities ontologies (the vocabulary for describing our cases) must also be created, although they could be reused between projects (after all, reuse is one of the goals of ontologies).

TABLE I
GAME STATE IN TWO INSTANTS

Variables	$Value\,(t_0)$	$Value\,(t_1)$	$\Delta\,(t_1 - t_0)$
?world			
difficulty	0.5 (MEDIUM)	0.4 (MEDIUM)	0.1 (VERY LOW)
?this			
health	0.8 (HIGH)	0.4 (MEDIUM)	0.4 (MEDIUM)
aggresive	0.2 (LOW)	0.8 (HIGH)	0.6 (MEDIUM)
defensive	0.7 (HIGH)	0.3 (MEDIUM)	0.4 (MEDIUM)
underAttack	0.0 (FALSE)	1.0 (TRUE)	1.0 (VERY HIGH)
?player			
bored	0.8 (HIGH)	0.3 (LOW)	0.5 (MEDIUM)

TABLE II
SMALL SAMPLE OF THE BEHAVIOR CASE BASE

Case	Parameters	Classes	Descriptors	Description
C_1	**Attack Don't Chase**			
	(hasTarget, entity: ALIVE)	Attack	?world.difficulty = L ?player.bored = L ?this.aggressive = L ?this.deffensive = H	Tries to attack an entity. If the entity tries to escape, the NPC doesn't chase it. It is suitable for low difficulty games or for NPCs not very aggressive or very defensive. It's not advisable when the player is bored.
C_2	**Attack And Chase**			
	(hasTarget, entity: ALIVE)	Attack Annoy	?world.difficulty = M ?player.bored = M ?this.aggressive = M ?this.defensive = M ?this.health = M	Attacks an entity and chases it if it tries to escape. Is valid for almost any situation as it is a very general way of attacking. Less recommendable if the health is low.
C_3	**Berserker**			
	(hasTarget, entity: ALIVE)	Attack	?world.difficulty = VH ?this.aggressive = H ?this.defensive = VL ?this.health = H	Attacks an entity with the most powerful weapon available and without caring about own safety. This behaviour is used for very aggressive entities and when the health is high. A defensive entity won't show this behaviour.
C_4	**Stealth Attack**			
	(hasTarget, entity: PLAYER)	Attack	?world.difficulty = M ?this.aggressive = L ?this.defensive = H ?this.underAttack = F ?this.health = M	Tries to approach an enemy without being noticed. The entity executing it has to remain undetected for the behaviour to be effective. It's more appropriate for medium or high difficulty games, and for entities more defensive than aggressive.
C_5	**Elusive Attack**			
	(hasTarget, entity: ALIVE)	Attack	?this.aggressive = M ?this.defensive = H ?this.underAttack = T ?this.health = L	Approaches the enemy and shoots him while trying to cover behind the objects in the game world and zigzag to avoid being hit. It's a defensive behaviour useful when the entity is being attacked or when the health is low.
C_6	**Take Cover**			
		Defend	?this.aggressive = L ?this.defensive = H ?this.underAttack = T ?this.health = L	When a NPC is attacked, it tries to defend itself by covering with walls, columns and scenery props. The NPC stays under cover without attacking until the enemy is close.

TABLE III
QUERY RESULTS

Query	Similarity					
	C_1	C_2	C_3	C_4	C_5	C_6
t_0	0.84	0.77	0.58	**0.90**	0.59	0.00
t_1	0.64	**0.82**	0.70	0.63	0.75	0.00

At runtime, our query node will spend more time *the first time* for extracting the appropriate basic behavior if comparing with a priority list. But, due to the *requery* attribute in the query node, we avoid spending time every AI cycle to change the first election, something that priority lists do not do. On the other hand, debug behaviors using our query nodes will be a bit more complex due to the new uncertainty ingredient added to the behavior selection. This problem can, in fact, be seen as an advantage, because some *emergent behavior* usually is considered to provide game variability.

We can estimate the benefits of our approach in the scalability for the growth of basic behaviors using some production numbers of Halo 2 available in [3]. The game has 115 basic behaviors and an average of 60 of them (≈50%) for each

TABLE IV
EXECUTION RESULTS FOR THE *attack* BEHAVIOR TREE

Execution results in t_0	**Execution results in t_1**
1. Attack Don't Chase {(hasTarget, PLAYER_INSTANCE_001)}	
FAIL – ?this.health (t_0) = VERY HIGH	**FAIL** – ?this.health (t_1) = MED ≠ LOW
2. Stealth Attack {(hasTarget, PLAYER_INSTANCE_001)}	
FAIL – ?world.difficulty (t_0) = LOW	**FAIL** – ?this.underAttack (t_1) = TRUE
3. Berserker {(hasTarget, PLAYER_INSTANCE_001)}	
FAIL – ?world.difficulty (t_0) = LOW	**FAIL** – ?world.difficulty (t_0) = MED
4. Stealth Attack {(hasTarget, PLAYER_INSTANCE_001)}	
SUCCESS – ?this.aggressive (t_0) = LOW ∧ ?this.underAttack (t_0) = FALSE	**FAIL** – ?this.underAttack (t_1) = TRUE
5. Attack And Chase {(hasTarget, PLAYER_INSTANCE_001)}	
– *Not evaluated* –	**SUCCESS** – ?world.difficulty (t_1) = MED

four-nodes-deep BTs of their 30 characters types. We can formulate some hypothesis.

1) The game production lasted two years.
2) At the beginning, designers had 43 basic behaviors, developed during the preproduction phase. Three new behaviors were made available each month.
3) On the other hand, during the preproduction phase, designers had assembled six BTs, with 20 basic behaviors (≈50%), and finished one more per month.

When AI designers start the BT of a new character type, they have available more basic behaviors than for the previous races. Revising the previous BTs is so time consuming that it becomes impossible. *Query nodes* allow *automatic revision* of nodes as new behaviors appear. Using the previous hypothesis, if Halo 2 had used it in the 20% of their three-deep nodes, it would have this automatic (accumulated) revision for 6700 nodes that would have always used the best basic behaviors available.

Such development improvements still have to be tried out in actual commercial games, but by introducing them as a well-defined extension to a technique widely used in industry we facilitate its adoption by practitioners.

VI. RELATED WORK AND CONCLUSION

Some related approaches can be found in the robotics and simulation domains. In [19], authors apply a case injected genetic algorithm to optimize the allocation of a collection of military strike assets to a set of targets, in the context of a strategic simulation game. Case injection consists of replacing the worst members of the population with individuals chosen from a case base, where cases are made up of chromosomes representing past strategies used by human experts. A key difference from their work is that in our approach the designers have a greater control over the NPC behavior by specifying the attributes of the query to retrieve cases at a given node in a BT.

Also in the simulation domain, Aleson and Louis [20] and Olenderski *et al.* [21] describe a training simulator to train officers in the tactical aspects of shiphandling. It uses a behavior-based control architecture. This kind of system employs a collection of concurrently executing processes, called behaviors, that receive an input from the sensors or other behaviors and send commands to the actuators. Behaviors are represented as abstract behavior networks [22], which are networks in which the links between behaviors represent precondition–postcondition dependencies. Behavior networks are basically an extension to HFSMs to allow for the concurrent execution of several behaviors, and, as such, suffer from the scalability problems found in FSMs when used to control complex dynamics in modern video games.

There is also some related work within the interactive storytelling and video games' arena. A behavior language (ABL) is presented in [23] as an extension to Hap [24], the agent architecture designed as part of the Oz project at Carnegie Mellon University (Pittsburgh, PA). Hap is basically a plan execution engine that, given a memory of previously designed plans, maintains a plan tree at runtime, with goals to be pursued and basic actions to be executed. Given an open goal, Hap selects an applicable plan from the plan memory, based on the state of the world and plan preconditions, and when more than one plan is applicable, it selects the most specific one, based on a specificity level (an integer) hand-coded in the plan. ABL extends Hap to allow for the specification of coordinated plans involving several agents (actually, two characters in the Façade interactive drama). The work described in [25] extends ABL to integrate case-based planning in the process of solving an open goal in the plan tree. Plans in the plan memory are traces of an expert playing the game (a real-time strategy game in this case), annotated with the goals the expert was pursuing and described through the state of the game when those actions took place. Given an open goal in the goal tree, a case is retrieved that is annotated to solve that same goal.

Purely goal-oriented formalisms such as [23] and [24] provide a declarative way for designers to express NPC behavior, allowing for an underlying search algorithm to explore a potentially huge solution space, and have already been used both

in academic [6] and industrial systems [7]. Nevertheless, the way to educate game designers to be able to write domain descriptions and plan preconditions and postconditions still has to be found. The case-based approach described in [25] faces this problem by providing an author-by-demonstration process, although it remains to be proved that such an approach can scaleup to take control of NPCs interacting in complex reactive environments such as those found in FPSs games.

In this paper, we have proposed the use of QBTs as a middle point between fully specified BTs and search-based goal-oriented formalisms. Through QBT, a designer can reuse low-level behaviors without actually knowing in advance every possible implementation for a given functionality, but being able to specify the features of the desired behavior, using a domain language. The ontology used to describe behavior queries and implementations is the contract between high-level and low-level behaviors, and it will evolve as the set of behaviors grows. Nevertheless, for this approach to be successful in practice, designers have to understand that a richer AI implies some degree of emergent behavior, where not everything can be a 100% predicted beforehand.

REFERENCES

[1] S. Rabin, Ed., *AI Game Programming Wisdom 3*. Boston, MA: Charles River Media, 2006.

[2] S. Rabin, Ed., *AI Game Programming Wisdom 4*. Boston, MA: Charles River Media, 2008.

[3] D. Isla, "Handling complexity in the Halo 2 AI," in *Proc. Game Developers Conf.*, 2005 [Online]. Available: http://www.gamasutra.com/view/feature/2250/gdc_2005_proceeding_handling_.php

[4] D. Isla, "Halo 3—Building a better battle," in *Proc. Game Developers Conf.*, 2008 [Online]. Available: http://www.bungie.net/Inside/publications.aspx

[5] J. Krajewski, "Creating all humans: A data-driven AI framework for open game worlds," *Gamasutra*, Feb. 2009 [Online]. Available: http://www.gamasutra.com/view/feature/1862/creating_all_humans_a_datadriven_.php

[6] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila, "Hierarchical plan representations for encoding strategic game AI," in *Proc. 1st Artif Intell. Interactive Digit. Entertainment Conf.*, R. M. Young and J. E. Laird, Eds., 2005, pp. 63–68.

[7] P. Gorniak and I. Davis, "Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters," in *Proc. Artif Intell. Interactive Digit. Entertainment Conf.*, J. Schaeffer and M. Mateas, Eds., 2007, pp. 14–19.

[8] C. Remo and B. Sheffield, "Redefining game narrative: Ubisoft's Patrick Redding on far cry 2," *Gamasutra*, Jul. 2008 [Online]. Available: http://www.gamasutra.com/view/feature/3727/redefining_game_narrative_.php

[9] E. D. Leake, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Menlo Park, CA: AAAI/MIT Press, 1996.

[10] M. Gilgenback and T. McIntosh, "A flexible AI system through behavior compositing," in *AI Game Programming Wisdom 3*. Boston, MA: Charles River Media, 2006, pp. 289–300.

[11] I. Borovikov and A. Kadukin, "Building a behavior editor for abstract state machines," in *AI Game Programming Wisdom 4*. Boston, MA: Charles River Media, 2008, pp. 333–346.

[12] D. Harel, "Statecharts: A visual formulation for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[13] R. Houlette, D. Fu, and D. Ross, "Towards an AI behavior toolkit for games," in *Proc. AAAI Spring Symp. Artif. Intell. Interactive Entertain.*, J. Laird and M. van Lent, Eds., 2001, pp. 50–53.

[14] M. S. Atkin, G. W. King, D. L. Westbrook, B. Heeringa, and P. R. Cohen, "Hierarchical agent control: A framework for defining agent behavior," in *Proc. 5th Int. Conf. Autonom. Agents*, 2001, pp. 425–432.

[15] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge, U.K.: Cambridge Univ. Press, 2003.

[16] P. P. Gómez-Martín, M. A. Gómez-Martín, P. A. González-Calero, and P. Palmier-Campos, "Using metaphors in game-based education," in *Lecture Notes in Computer Science*, K.-C. Hui, Z. Pan, R. C.-K. Chung, C. C.-L. Wang, X. Jin, S. Göbel, and E. C.-L. Li, Eds. Berlin, Germany: Springer-Verlag, 2007, vol. 4469, pp. 477–488.

[17] G. Flórez-Puga, B. Díaz-Agudo, and P. A. González-Calero, "Experience-based design of behaviors in videogames," in *Lecture Notes in Computer Science*, K.-D. Althoff, R. Bergmann, M. Minor, and A. Hanft, Eds. Berlin, Germany: Springer-Verlag, 2008, vol. 5239, pp. 180–194.

[18] G. Flórez-Puga, M. A. Gómez-Martín, B. Díaz-Agudo, and P. A. González-Calero, "Dynamic expansion of behaviour trees," in *Proc. Artif Intell. Interactive Digit. Entertainment Conf.*, C. Darken and M. Mateas, Eds., 2008, pp. 36–41.

[19] C. Miles, S. J. Louis, N. Cole, and J. McDonnell, "Learning to play like a human: Case injected genetic algorithms for strategic computer gaming," in *Proc. Congr. Evol. Comput.*, Jun. 2004, vol. 2, pp. 1441–1448.

[20] R. S. Aleson and S. J. Louis, "Developing adaptive tactical aggressors," in *Proc. Interservice/Industry Training Simulation Education Conf.*, 2006 [Online]. Available: http://ntsa.metapress.com/app/home/contribution.asp?referrer=parent&backto=issue,103,158;journal,3,12;linkingpublicationresults,1:113340,1

[21] A. Olenderski, M. N. Nicolescu, and S. J. Louis, "A behavior-based architecture for realistic autonomous ship control," in *Proc. IEEE Symp. Comput. Intell. Games*, S. J. Louis and G. Kendall, Eds., 2006, pp. 148–155.

[22] M. N. Nicolescu and M. J. Matarić, "A hierarchical architecture for behavior-based robots," in *Proc. 1st Int. Joint Conf. Autonom. Agents Multiagent Syst.*, New York, 2002, pp. 227–233 [Online]. Available: http://www.dx.doi.org/10.1145/544741.544798

[23] M. Mateas and A. Stern, "A behavior language for story-based believable agents," *IEEE Intell. Syst.*, vol. 17, no. 4, pp. 39–47, Jul./Aug. 2002.

[24] A. B. Loyall and J. Bates, "Hap—A reactive, adaptive architecture for agents," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-91-147, Jun. 1991.

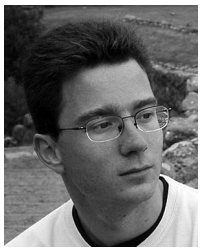
[25] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Learning from demonstration and case-based planning for real-time strategy games," in *Soft Computing Applications in Industry*, B. Prasad, Ed. New York: Springer-Verlag, 2008, pp. 293–310.

Gonzalo Flórez-Puga received the B.S. degree in computer science and the M.S. degree in intelligent systems from the Universidad Complutense de Madrid, Madrid, Spain, in 2006 and 2007, respectively, where he is currently working towards the Ph.D. degree at the Artificial Intelligence Applications Group (GAIA).

His research interests include case-based reasoning, games AI, and intelligent behavior authoring and design.

Marco Antonio Gómez-Martín received the Ph.D. degree in computer science from the Universidad Complutense de Madrid, Madrid, Spain, in 2008.

Currently, he is a Lecturer in Computing at the Universidad Complutense de Madrid in subjects related to videogame architecture, artificial intelligence for videogames, and scripting in games among others. His research interests center around general architectures for virtual environments, such as entertainment applications and serious games, and automatic generation of NPC behavior and game levels.



Pedro Pablo Gómez-Martín received the M.S. and Ph.D. degrees in computer science from the Universidad Complutense de Madrid, Madrid, Spain, in 2000 and 2008, respectively.

He has taught in the Master on Videogame Development at the Universidad Complutense de Madrid since its first edition in 2004. His research involves AI and software engineering aspects for the development of educational video games, searching for new ways to improve how this software guides student learning while keeping development costs under control.

Belén Díaz-Agudo received the Ph.D. degree in computer science from the Universidad Complutense de Madrid, Madrid, Spain, in 2002.

Currently, she is an Associate Professor in the Faculty of Informatics, Universidad Complutense de Madrid. She is teaching AI and knowledge-based system courses at different levels of expertise. Her research interests are knowledge-based systems, case-based reasoning, ontologies, and the application of AI techniques to videogames.

Pedro Antonio González-Calero received the Ph.D. degree in physics from the Universidad Complutense de Madrid, Madrid, Spain, in 1997.

Currently, he is an Associate Professor of Computer Science at the Universidad Complutense de Madrid, where he has been teaching in the Faculty of Informatics since its creation in 1991. He has been the Director of Universidad Complutense de Madrid Master on Videogame Development since its creation in 2004 and he is member of the IFIP Technical Committee on Entertainment Computing. His research has focused on the confluence of software engineering and AI, with contributions in the areas of knowledge-based software engineering, software reuse, and case-based reasoning.

6.7. Evaluating Sketch-based Retrieval Speed-up for Behaviour Design in Soccerbots

Cita completa

Gonzalo Flórez Puga, Belén Díaz Agudo y Pedro González Calero. *Evaluating Sketch-based Retrieval Speed-up for Behaviour Design in Soccerbots.* 25th International Florida Artificial Intelligence Research Society Conference. En proceso de revisión.

Resumen original de la contribución

Sketch-based retrieval is a technique that supports the design of behaviour for game characters by reusing previously designed behaviours. Most techniques for specifying behaviour for game characters use some kind of graph-based formalism to represent such behaviour. Through graph-matching techniques, sketch-based retrieval allows to use any intermediate graph generated along the design process, a *sketch* of the final behaviour, as a query to retrieve similar behaviours from a library of complete behaviours. In this paper we describe the design and results from an experiment designed to measure to what extent having a library of reusable behaviours accessed through sketch-based retrieval can speed-up the behaviour design process in the Soccerbots game.

Referencia de citas bibliográficas

Isla (2005), Millington (2006), Bourg y Seemann (2004), Flórez-Puga et al. (2011), Flórez-Puga et al. (2012a), Jiménez-Díaz et al. (2011), Bunke y Messmer (1994), Riesen y Bunke (2009), Flórez-Puga et al. (2012a), Flórez-Puga et al. (2012a)

Evaluating Sketch-based Retrieval Speed-up for Behaviour Design in Soccerbots [*†]

Gonzalo Flórez-Puga and **Belén Díaz-Agudo** and **Pedro González-Calero**
Complutense University of Madrid. Madrid. Spain
{gflorez, belend, pedro}@fdi.ucm.es

Abstract

Sketch-based retrieval is a technique that supports the design of behaviour for game characters by reusing previously designed behaviours. Most techniques for specifying behaviour for game characters use some kind of graph-based formalism to represent such behaviour. Through graph-matching techniques, sketch-based retrieval allows to use any intermediate graph generated along the design process, a *sketch* of the final behaviour, as a query to retrieve similar behaviours from a library of complete behaviours. In this paper we describe the design and results from an experiment designed to measure to what extent having a library of reusable behaviours accessed through sketch-based retrieval can speed-up the behaviour design process in the Soccerbots game.

Introduction

We present a new approach to the authoring of behaviours for non-player characters (NPCs) in videogames based on retrieval and reuse from a collection of reusable behaviours. The motivation behind our approach is that typically in a large game we can find simple behaviours that are replicated within different complex behaviours. For instance, in a soccer game, `Defend` could be a complex behaviour that is composed of two simpler behaviours like `Go to the ball` and `Clear`; meanwhile `Attack` could be made up of `Go to the ball`, `Dribbling` and `Shoot`. However, the actual process of AAA game development, where a group of game designers and programmers collaborate over a long period of time to iteratively design a large number of complex behaviours (for instance, Halo 2 had an average of 60 different behaviours arranged in 4 layers (Isla 2005)), does not currently rely on behaviour reuse. Without supporting tools and technology, reuse is not an option, and game designers tend to develop new behaviours from scratch, resulting in variations of similar behaviours coexisting in the same game, ignoring the benefits of reuse in terms of quality and scalability.

To help designers in the task of building and reusing behaviours we have developed *eCo*[1], a visual editor capable of storing, indexing, retrieving and reusing previously designed behaviours. Although in this paper we exemplify the approach with behaviours represented as hierarchical finite state machines (HFSMs) (Millington 2006), the editor can deal with other formalisms typically employed for designing behaviour in videogames, such as finite state machines (FSMs) (Bourg and Seemann 2004), and behaviour trees (BTs) (Flórez-Puga et al. 2011).

One of the most notable features of our editor is its capability for sketch-based retrieval. In the image retrieval domain, sketch-based retrieval consists in finding a complex image using an approximate representation of it (a sketch) as a query. We can translate that idea to the behaviour domain, where a sketch is a partial representation of a behaviour (for instance, a FSM that is missing some edges or where the behaviour of a node has not been specified). In sketch-based retrieval of behaviours we search in a repository for behaviours that are similar to the one the user is drawing, making suggestions about how to complete it.

The question we address in this paper is to what extent sketch-based retrieval can speed-up the process of designing a new behaviour when the target behaviour is already in the library of reusable behaviours. In essence we measure at what point in the design process our algorithm would retrieve the completed target behaviour and how much effort is saved in terms of edition steps.

The rest of the paper runs as follows. Next section describes the main ideas of sketch-based retrieval and its implementation in the *eCo* behaviour editor. Next we describe the experiment setup before going into the section describing the results. Finally, we draw some conclusions.

eCo Behaviour Editor

eCo is a visual editor that helps game designers in the task of developing behaviours for NPCs in games. In particular, the version presented in this paper allows creating HSFMs that implement behaviours for Soccerbots robots, but the editor can be configured to be used with other games. For a more detailed description of *eCo* we refer to (Flórez-Puga et al. 2013).

[*]Supported by Spanish Ministry of Economy and Competitiveness under grant TIN2009-13692-C03-03.

[†]Funded by Complutense University of Madrid.

[1]eCo: `http://gaia.fdi.ucm.es/research/eco-behaviour-editor`

Soccerbots[2] is a well-known simulation environment that simulates the dynamics and dimensions of a regulation RoboCup[3] small size robot league game. Two teams of five robots compete on a soccer field by pushing and kicking a ball into the opponent's goal.

To execute the matches we rely on SBTournament[4] (Jiménez-Díaz et al. 2011), an enhanced environment to run Soccerbots matches. SBTournament offers different interfaces that allow to configure and run automatically multiple matches between two sets of teams. Besides, it generates a very useful complete log with statistics regarding the matches played.

Aside from assistance for configuring and launching large sets of matches, SBTournament provides users with a set of sensors and actuators, which are an enhanced superset of those provided by SoccerBots. Actuators are the most simple actions that a robot can execute, while sensors are the pieces of information that a robot can gather from the game world. For example, actuators in SBTournament allow users to kick the ball or set the desired heading and speed for a robot. Likewise, sensors provide information about the ball position or the position of the opponent's goal. The editor uses sensors and actuators to build the HFSMs that our robots will execute. On one hand, sensors are used to build the conditions for the edges of the HFSMs. On the other hand, actuators are used to build the *basic behaviours*, i.e. the basic building blocks for the robot's behaviour. Basic behaviours are the simplest actions that can be executed in a node of a robot's HFSM. These basic behaviours generally consist of a sequence of calls to different actuators.

eCo allows users to "draw" HFSMs to specify the behaviour of SBTournament's robots and teams. As the user draws, the partially completed HFSM is used as a sketch to retrieve previously created behaviours. The users can reuse pieces of the retrieved behaviours to complete the one being edited. Once a HFSM is finished it can be exported and executed in SBTournament. Finished behaviours are added to a library of created behaviours in order to be reused later.

Figure 1 shows the editor's graphic interface that allows users to design individual players and teams. The area in the middle is a canvas where users can draw the HFSMs that represent the robots' behaviours. Under the drawing canvas there is a text editor where users can write code to create and/or modify the basic behaviours.

The most important feature of the editor is its capability to retrieve behaviours stored in a library using two kinds of searches: *attribute-based* retrieval and *sketch-based* retrieval. For attribute-based retrieval we make the behaviours in the library play several matches versus a set of preselected *trainer* behaviours. From each match we gather some statistics that describe the gameplay of the behaviour (for instance, the number of goals scored, the covered distance, the average distance to each goal, etc.). To issue a query, the user assigns values to a subset of the statistics. The behaviours with the most similar statistics are retrieved from the library. Sketch-based retrieval allows designers to retrieve behaviours using as a query a partial graphical representation of the desired behaviours as we describe next.

[2]SoccerBots: `http://www.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots`

[3]Robocup: `http://www.robocup.org/`

[4]SBTournament: `http://gaia.fdi.ucm.es/research/sbtournament`

Sketch-based Retrieval

In sketch-based retrieval designers can use a sketch of the desired behaviour as a query. A sketch is a partial or unfinished representation of the behaviour. The sketch is typically a behaviour that is being drawn by the designer but it is still not finished (that is, an intermediate step in the process of creating a behaviour). This allows the editor to automatically suggest different ways to finish the behaviour.

This approach requires an appropriate similarity function. For complete behaviours like the ones in the library, we can make them play and gather statistics about their gameplay to see if they are similar because they behave similarly. But in the case of a sketch, that is not possible, because the behaviour is not finished yet. Instead, we have to rely on another similarity metric that allows us to compare behaviours and predict which of them behave similarly. In particular we use the graph edit distance to compare the underlying graphs of the sketch and the cases in the library. The problem with the, so to speak, "standard" graph edit distance is that its cost is exponential on the number of nodes of the graph (Bunke and Messmer 1994). For this reason we have used the heuristic proposed in (Riesen and Bunke 2009), adapting it to HFSMs. As we have shown in past studies (Flórez-Puga et al. 2013) using this heuristic we obtain a result set that is almost indistinguishable from the original similarity function, but at a rather reduced cost in time.

The suggestion feature in the editor works as follows: while the designer is drawing the behaviour, the editor uses the current (probably unfinished) behaviour as a query. If the designer takes a new edition step, like adding a node or changing the label on an edge, a new query is issued with the changed sketch. The top results of the query are shown in the suggestions panel, which is at the left side of the drawing canvas in Figure 1. The designer can use any of the results instead the current sketch, or can combine them with the sketch being edited. When the user selects a behaviour suggested from the library, the editor shows some statistics about it in the table below. The statistics are gathered by making the teams play versus a predefined set of *trainer teams*.

The adaptation process is not automatized, but the system offers some assistance for manual adaptation. Information regarding the gameplay of the teams suggested can be employed by the users to adjust the team being built. For instance, if the user wants to develop a team that has a defensive gameplay she could compare her team with the teams suggested. She could then find a more defensive team (with few goals against or matches lost) but still similar to hers, and use it as a model to modify its configuration.

Experiment setup

As we have seen, sketch-based retrieval can help game AI designers to create behaviours for NPCs by providing candi-

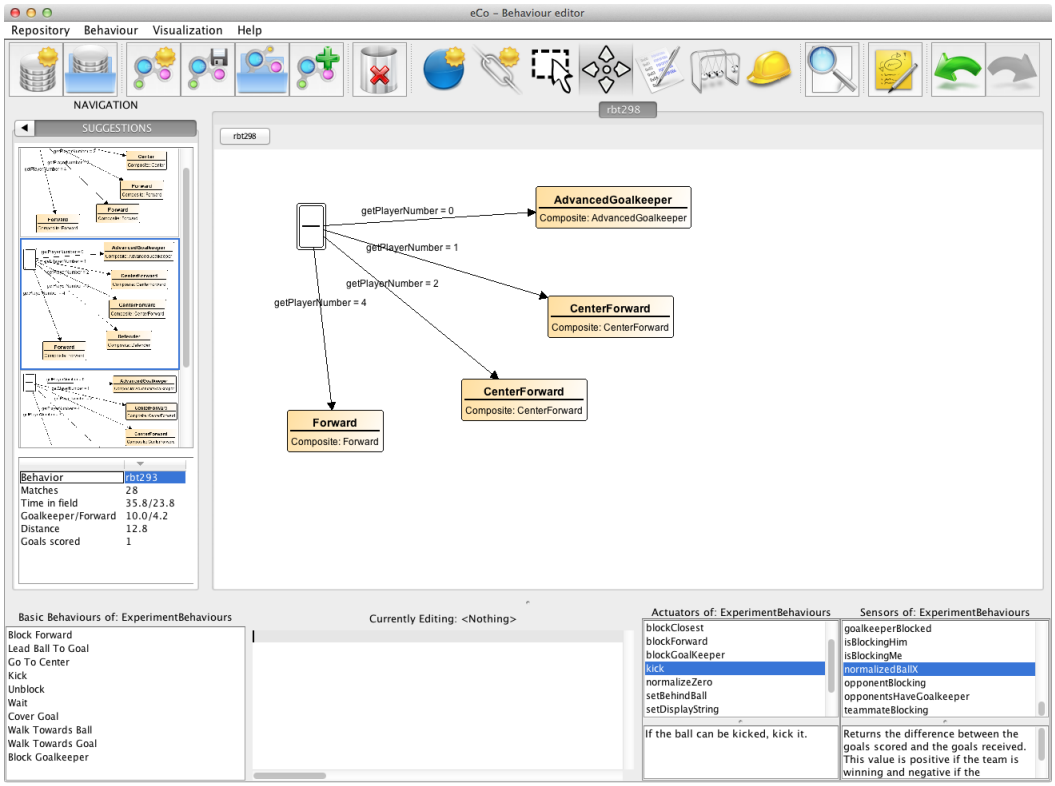


Figure 1: Capture of eCo

date behaviours to be reused. This way, development time of new behaviours can be reduced. In this section we describe an experiment we have conducted to measure the savings in development time or, more precisely, in development steps.

The idea behind the experiment is to compare the number of steps taken by a user to create the same behaviour with and without the sketch-based retrieval feature. The experiment was conducted with 43 students from the Knowledge Based Systems course of the Complutense University, divided in 16 groups, during two sessions of two hours. In the first session we taught the users how to build the behaviours using the editor. To that end we asked them to follow a tutorial that taught them how to build an example goalkeeper behaviour and then a team using different simple behaviours. As they were asked to build the same behaviours, we did not keep the behaviours created by the users in this session.

Before the second session they had one week to design several roles (e.g. goalkeepers, forwards, defenders, etc.) for a Soccerbots team using "pen and paper". For the second session we asked them to implement those roles in the editor and build a team combining them. For this second session they weren't allowed to use the retrieval features of the editor.

Once they had finished, we collected the behaviours they built. In total we collected 95 behaviours with an amount of nodes that ranged from 2 in the simpler behaviours (e.g. "Go to my goal" or "Kick ball") to more than 20 for the team behaviours.

Together with the behaviours, we collected an execution trace generated by the editor, that contained all the edition steps that the users had followed. We consider an edition step any operation that introduces a change in the behaviour being edited: adding or deleting a node or edge, editing the label associated to a node or an edge or changing the initial node of a behaviour. We don't consider edition steps, for instance, the creation of new basic behaviours or adding a behaviour to the library. Unsurprisingly, the quantity of edition steps is related to the number of nodes of the behaviour. In the behaviours we collected we found that the number of steps ranges from around 10 for the smaller behaviours (with 2 or 3 nodes) to more than 300 for the teams. Using this trace we were able to rebuild the original behaviours.

For each behaviour implemented by the users, B_i, we used its trace to obtain a set of intermediate steps, which are incomplete versions of the behaviour B_i. We called this set the intermediate behaviours $I_i = \{I_{i,0}, \ldots, I_{i,si}\}$, where si is the total amount of steps taken to obtain B_i. Hence, the set I_i ranges from the empty behaviour $I_{i,0}$ (a behaviour without any nodes or edges) to the final behaviour that was implemented $I_{i,si} = B_i$. Each $I_{i,j}$ is the intermediate behaviour obtained after applying step j.

To run the experiment we also needed a case base. Our case base is composed of all the the final behaviours from the users, plus a set of 205 behaviours that were created by ran-

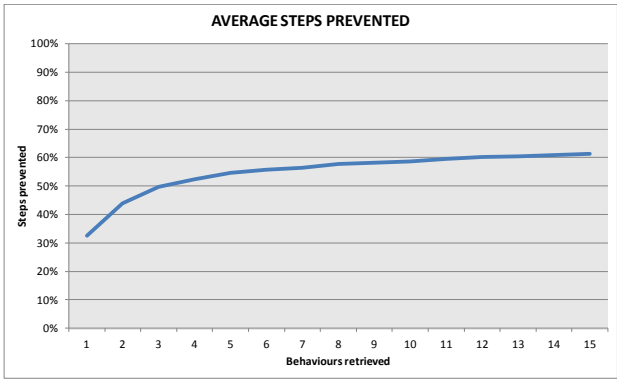


Figure 2: Average steps prevented when using sketch-based retrieval

domly composing different roles we already had from past experiments (Flórez-Puga et al. 2013), which makes a total of 300 behaviours in our case base. The size of the added behaviours goes from 14 to 35 nodes.

To determine the number of steps spared by the sketch-based retrieval for each B_i, we have used as a query each of the intermediate behaviours $I_{i,j}$, with $0 \leq j \leq si$, retrieving from the case base the k-nearest neighbours. In this way we are mimicking the behaviour of the editor when using the sketch-based retrieval feature. As we will see in the following section, the number of behaviours retrieved, k, has a great influence on the final results.

Then, we checked if the retrieved list contains the final behaviour B_i. If B_i belongs to the list of behaviours retrieved by query $I_{i,j}$ it means that using sketch-based retrieval the user can obtain the desired behaviour at step j, sparing the remaining steps until si.

Results

In the Figure 2 we show the average number of steps spared for each value of k up to 15 behaviours retrieved. We can see that, when we increase the value of k, the number of steps spared is also increased. Although we have registered better values of spared steps for values of k higher than 15, it is not practical for the users to have a big list of retrieved behaviours and, in any case, they are most likely to analyse only the first few of them.

We observe that when we only show the most similar behaviour to the user (that is, when $k = 1$), the user can prevent one third of the total amount of steps. If we show more behaviours to the user, the number of prevented steps rises to around 50% for $k = 3$. From there on the improvement is more gradual, reaching the 60% when $k = 11$. This observation indicates that, in average, users can prevent a great number of edition steps (up to 50%) when showing them only a few results from the query.

The standard deviation σ for each value of k remains almost constant around 20% for all the values shown. This indicates that for most of the cases, the number of prevented steps are in a $\pm 20\%$ range from the average.

To narrow this range and get a better idea of the overall results we also have studied the frequencies of the results for different values of k, as shown in the Figure 3. Each pie chart represents the results of retrieval for a different value of k. Each section in the chart represents the proportion of retrieved behaviours for which we spare at most the percentage of steps indicated. Table 1 shows the specific values of frequency. Each column in the table represents a value for k, while each row is a range of spared steps. The value in each cell shows the number of behaviours for which we prevent a number of steps in the corresponding range.

We note that for $k = 1$ there is a saving of more than 50% of the steps needed to create the behaviour in 16 out of the 98 behaviours studied. This value grows to 34 when $k = 2$ and to 44 (almost half of the total) when $k = 3$. This upward trend is steady for bigger values of k, but with a gentler increase. We also can see that the section labelled with 0% is present only when $k = 1$. This means that, when the number of retrieved behaviours is 2 or more, there is no case in the case base for which we don't save any steps.

Another factor to take into account is from what step the results retrieved are reliable. If a query is issued after too few steps, the sketch is less likely to summarize the structure of the desired behaviour and, hence, the retrieved behaviours won't be what the user expects. For that reason, before issuing any query it is advisable that the user takes some edition steps to reach a more detailed sketch. We have observed that the number of edition steps needed before obtaining de desired behaviour depends on the final size of the behaviour and on the number of elements retrieved in each query.

Table 2 shows the number of edition steps the user needed to take to retrieve the desired behaviour using sketch-based retrieval. We divided the set of behaviours in three groups according to their size: *small* behaviours with a total of 2 or 3 nodes(41 of them), *medium*, with 4 to 7 nodes (also 41) and *large* behaviours, with 8 and 9 nodes (from which we have 5). The remaining 8 behaviours are too scattered to be grouped. The second column shows the average number of edition steps that the users needed to create the behaviours in that group. This gives an upper bound of the number of steps needed in the worst case (that is, without using sketch-based

	Retrieved behaviours (k)					
Spared steps	1	2	3	5	10	15
0 %	7	0	0	0	0	0
(0, 10] %	8	3	1	0	0	0
(10, 20] %	21	11	6	3	2	2
(20, 30] %	15	18	17	14	10	8
(30, 40] %	12	17	14	11	9	9
(40, 50] %	16	12	13	15	14	13
(50, 60] %	9	13	11	12	14	11
(60, 70] %	4	11	18	20	21	23
(70, 80] %	0	5	7	9	8	8
(80, 90] %	3	5	8	9	12	12
(90, 100] %	0	0	0	0	5	9

Table 1: Frequencies for different ranges of spared steps

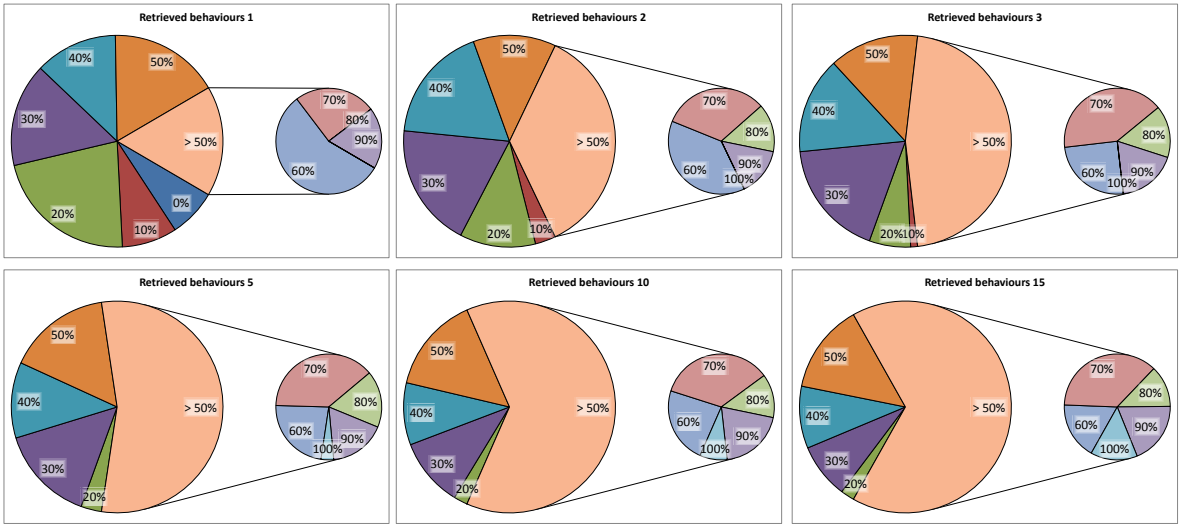


Figure 3: Steps spared for different values of k

Size	**Total steps**	**Steps needed for k**					
		1	2	3	5	10	15
Small (2 or 3 nodes)	18	10.88	8.51	7.17	6.10	5.15	4.19
Medium (4 to 7 nodes)	56	40.44	33.73	31.63	29.44	28.07	27.44
Large (8 or 9 nodes)	80	58.40	54.20	50.20	38.40	33.20	32.80

Table 2: Number of edition steps needed in average to obtain the final behaviour in a query

retrieval). The remaining columns show the average number of steps needed to obtain the desired behaviour using sketch-based retrieval for different values of k.

We can see again that when we increase the number of behaviours shown to the user (k) the number of steps needed to obtain the behaviour decreases, fast for the first values of k but in a smoother manner for values over 5. We can also see that, for bigger behaviours the user needs to take more steps to obtain the expected result.

In Figure 4 we show the evolution of the position of the desired behaviour in the results list for three different behaviours of different characteristics. Although they are three particular behaviours, we have chosen them in such a way that they are representative of each of the classes described before. The first case corresponds to a *small* behaviour, of only 2 nodes, that was completed by the user in 27 steps. The second one is *medium* sized, has 5 nodes and took the user 40 steps. The last one is *large*, has 8 nodes and took 56 steps. The horizontal axis of the graphs represents the number of steps taken by the user and the vertical axis the position of the desired behaviour in the results for that step. We have placed vertical dotted lines dividing the number of steps at 25%, 50% and 75%. The desirable result is a graph that goes down fast (meaning that the desired behaviour is found after a few steps) and then stays stable at a low position (this way, although the user has missed the behaviour in the first positions of the results list, he can retrieve it again in a later query). That is the case of the first example. We can see that, although at the very first steps it is retrieved in a high position, the position goes down to 3 at step 11 and is retrieved the first after step 12, staying there for the remaining steps. This means that using sketch-based retrieval, the appropriate behaviour is retrieved using the 40% of the steps that were needed to create it in the first place.

The second example behaves similarly. In this case we see that the result needs more steps to stabilize (16 steps to reach position 3 and 24 to reach the first position), but if we attend to the percentage of steps, we are also around 40%. In the results shown in the third graph we can see that it takes still more steps to find the behaviour and also to stabilize. The percentage of steps needed in this case has also grown up to 55%.

Analysing the graphs of these and other similar behaviours we can conclude that the number of steps the user needs to take before issuing a query grows along with the size of the behaviour but decreases when we increment the number of behaviours retrieved in each query (k). In general terms, the user needs to take around 40% of the total number of steps to be sure that the system retrieves the adequate behaviour. This percentage is greater for the biggest behaviours in our collection.

Regarding execution times, as we mentioned earlier, we used a heuristic similarity function to avoid the exponential cost inherent to structural similarity functions for graphs. Using this heuristic, each query was resolved in an average time of 0.117 ms. Therefore it is possible for eCo to auto-

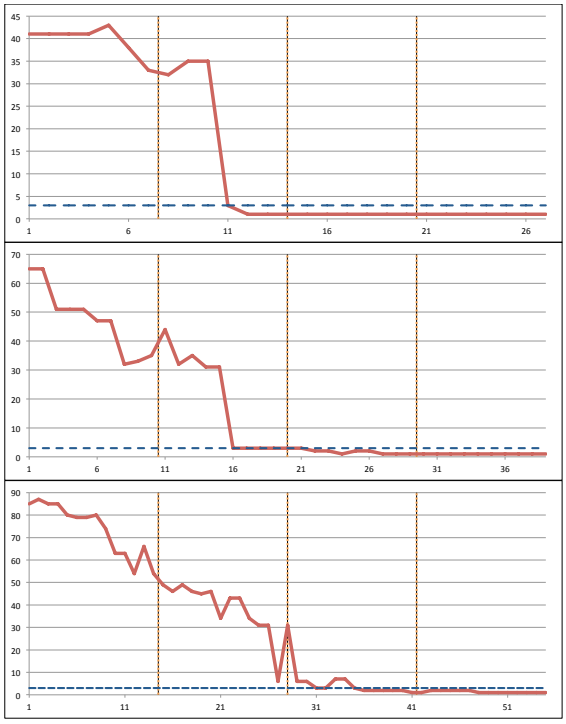


Figure 4: Evolution of the position of the target behaviour in the results list for three example queries

matically issue a new query after each edition step without imposing any delays for the user interaction, at least for the size of the cases bases that we have tried.

Conclusions and Future Work

In this paper we have presented the evaluation of a novel approach to the creation of intelligent behaviours that is based in reuse. We have called this approach *sketch-based retrieval*. In sketch-based retrieval designers use a sketch of the desired behaviour to retrieve from a library previously created behaviours that are similar to it. The similar behaviours are shown to the user who can then select a complete behaviour or a fragment to complete the one used as a query. It is interesting to note that the querying process doesn't require user intervention.

Using the retrieved behaviours to finish the one being developed designers can save time in the edition of behaviours while, at the same time, reduce the possible errors, because they are using behaviours or pieces of behaviour that have been previously tested. We have demonstrated experimentally that using sketch-based retrieval actually reduces the number of edition steps the designer has to take to obtain the desired behaviour, hence reducing the development time. The amount of steps prevented is dependant on the number of behaviours retrieved (k): when we increase the number of behaviours retrieved the number of steps prevented also grows.

But retrieving too many behaviours is not useful for the user, because he would have to search in a long list for the behaviours he is interested in. We propose to use a value of $k = 3$. In our experiment we have shown that using this value we prevent at least 50% in 47 behaviours out of the total of 98 cases evaluated. We have also found that for this value of k we obtain good results for small behaviours (2 or 3 nodes) after the user has taken around 7 steps. For bigger behaviours (between 4 and 7 nodes) we needed around 30 steps to get the result. For behaviours of 8 or 9 nodes, our approach needed around 50 steps to return the relevant result between the first 3. In the examples of Figure 4 we have drawn a dashed horizontal line in $k = 3$. We can see that for the small behaviour, we obtain a good result after step 11, in the second example, the medium sized, after step 16 and in the last one after step 31.

As future work, we plan to extend eCo in order to allow the creation of Behaviour Trees (BTs), which is the formalism of choice in most commercial videogames for representing NPC behaviour. Since BTs are also graphs, as HFSMs, the underlying technology should work as is now and only changes to the interface should be required before going into new experiments.

References

Bourg, D. M., and Seemann, G. 2004. *AI for Game Developers*. O'Reilly Media, Inc.

Bunke, H., and Messmer, B. T. 1994. Similarity measures for structured representations. In *EWCBR '93: Selected papers from the First European Workshop on Topics in Case-Based Reasoning*, 106–118. London, UK: Springer-Verlag.

Flórez-Puga, G.; Llansó, D.; Gómez-Martín, M. A.; Gómez-Martín, P. P.; Díaz-Agudo, B.; and González-Calero, P. A. 2011. Empowering designers with libraries of self-validated query-enabled behaviour trees. In González-Calero, P. A., and Gómez-Martín, M. A., eds., *Artificial Intelligence for Computer Games*. Springer. 55–82.

Flórez-Puga, G.; González-Calero, P. A.; Jiménez-Díaz, G.; and Díaz-Agudo, B. 2013. Supporting sketch-based retrieval from a library of reusable behaviours. *Expert Systems with Applications* 40(2):531–542.

Flórez-Puga, G.; Díaz-Agudo, B.; and González-Calero, P. A. 2010. Similarity measures in hierarchical behaviours from a structural point of view. In Guesgen, H. W., and Murray, R. C., eds., *FLAIRS Conference*. AAAI Press.

Isla, D. 2005. Handling Complexity in the Halo 2 AI. In *Game Developers Conference*.

Jiménez-Díaz, G.; Menéndez, H. D.; Camacho, D.; and González-Calero, P. A. 2011. Predicting performance in team games. the automatic coach. In *3rd International Conference on Agents and Artificial Intelligence (ICAART 2011)*, 401 – 406. Rome, Italy: SciTePress.

Millington, I. 2006. *Artificial intelligence for games*. The Morgan Kaufmann series in interactive 3D technology. Morgan Kaufmann Publishers Inc.

Riesen, K., and Bunke, H. 2009. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.* 27(7):950–959.

Apéndice A

El Editor de Comportamientos *eCo*

En paralelo a la investigación realizada, se ha desarrollado una herramienta software de edición de comportamientos llamada *eCo*[1]. *eCo* es un editor visual altamente configurable, que permite la creación de comportamientos para prácticamente cualquier juego o entorno de simulación. En la versión actual del editor, los comportamientos son especificados como HFSM, pero el editor está diseñado para poder añadir otros tipos de representación como BT realizando cambios mínimos, que tendrán lugar en futuras versiones. El editor ha servido para un doble propósito: por un lado, ha sido un banco de pruebas para comprobar la efectividad de los algoritmos y las técnicas presentadas en este trabajo y, además, ha facilitado en gran medida el proceso de adquisición del conocimiento, ya que los casos con los que trabajamos son comportamientos que han sido creados utilizando el editor.

eCo es una pieza fundamental dentro del modelo de diseño de comportamientos basado en la reutilización ya que, no sólo se encarga de la edición visual de comportamientos; también incluye las herramientas necesarias para gestionar las bibliotecas de comportamientos creadas y para realizar en ellas las búsquedas descritas en la sección anterior.

La Figura A.1 muestra una captura del editor de comportamientos. Como se puede ver en la figura, la GUI (*Graphic User Interface*, Interfaz Gráfica de Usuario) está dividida en cuatro partes. En la parte superior se encuentra la barra de herramientas, que muestra las herramientas que se pueden utilizar para gestionar las bibliotecas y editar los comportamientos. La parte inferior muestra el editor de comportamientos básicos, utilizado principalmente por los programadores (o por diseñadores con conocimientos de programación) para crear y modificar los comportamientos básicos asociados al modelo de juego. En la parte central, a la derecha se puede ver el lienzo de edición,

[1] *eCo*: `http://gaia.fdi.ucm.es/research/eco-behaviour-editor`

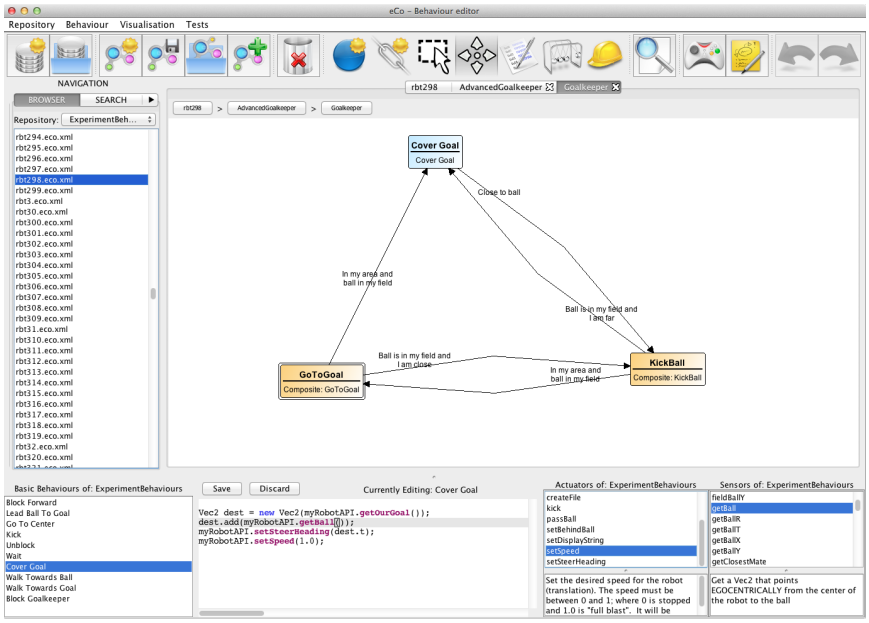


Figura A.1: Interfaz de usuario de *eCo*

donde los diseñadores pueden dibujar los comportamientos. A la izquierda del lienzo se encuentra el panel de navegación. Este panel está compuesto por tres pestañas que permiten realizar acciones relacionadas con la gestión de las bibliotecas y las búsquedas de comportamientos. A lo largo del resto de la sección explicaremos cada una de las partes de la GUI, relacionándolas con los elementos del modelo propuesto en la Sección 3.2.

Como hemos dicho, *eCo* puede utilizarse para la creación de comportamientos en prácticamente cualquier juego o simulador. Para ello, necesita un *modelo de juego* que describa la interfaz de comunicación de los comportamientos con la IA del juego concreto. En la práctica, el modelo de juego es un fichero `xml` en el que se especifican los sensores y los actuadores propios del juego. Para cada uno de estos elementos se añade una entrada en el modelo de juego.

En el caso de los actuadores es necesario especificar los siguientes datos:

- **Nombre** del actuador.
- **Descripción** textual acerca de para qué sirve el actuador.
- **Comando**: código fuente necesario para realizar en el juego la acción correspondiente al actuador.
- Número de **parámetros** que recibe.

La información necesaria para los sensores es la siguiente:

- **Nombre** del sensor.
- **Descripción** textual acerca de para qué sirve el sensor.
- **Comando**: fragmento de código fuente necesario para realizar la llamada que devuelve los datos de este sensor.
- **Tipo** de datos del valor devuelto.

Para permitir mayor complejidad en los comportamientos, añadimos una capa más sobre los sensores y los actuadores: los *comportamientos básicos*. Un comportamiento básico no es más que un fragmento de código que realiza una acción simple invocando a varios actuadores y sensores, y opcionalmente a otras construcciones del lenguaje de destino.

Por ejemplo, en SBTournament podríamos usar los actuadores `setSteerHeading` y `setSpeed` y el sensor `getPosition` para construir un comportamiento básico capaz de mover al jugador hasta el centro del campo. Para definir el comportamiento básico solamente necesitaríamos dos propiedades: su nombre (*Ir al centro*), y el código correspondiente:

```
// El punto de destino es el centro (0, 0)
Vec2 destino = new Vec2(0.0, 0.0);
// Calculamos el vector hasta el destino
destino.sub(myRobotAPI.getPosition());
// Giramos el robot en esa dirección
myRobotAPI.setSteerHeading(destino.t);
// Movimiento a la máxima velocidad
myRobotAPI.setSpeed(1.0);
```

En el modelo de juego también se describen los atributos que se utilizarán para realizar las consultas. Estos atributos describen la funcionalidad del comportamiento y pueden ser introducidos por el usuario o pueden ser estadísticas que se capturan automáticamente al ejecutar cada comportamiento.

En la versión actual del editor los atributos deben ser monovaluados y su valor debe ser numérico. Para definir cada atributo hay que especificar el **nombre** del atributo y los valores **mínimo** y **máximo** que puede tener.

Para realizar las pruebas sobre el editor, hemos construido modelos de juego para SoccerBots[2] y SBTournament[3]. Durante el desarrollo de la primera versión de *eCo* también construimos un modelo de juego para controlar un robot Aibo[4]. Para utilizar el editor con otro juego, sólo es necesario cambiar el modelo de juego por el del juego correspondiente.

Un elemento importante en nuestro modelo son las bibliotecas de comportamientos. El editor se encarga de gestionar la creación y eliminación de las bibliotecas, así como de su mantenimiento, cuidando en todo momento

[2]SoccerBots: `http://www.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots`
[3]SBTournament: `http://gaia.fdi.ucm.es/research/sbtournament`
[4]Aibo: `http://es.wikipedia.org/wiki/Aibo`

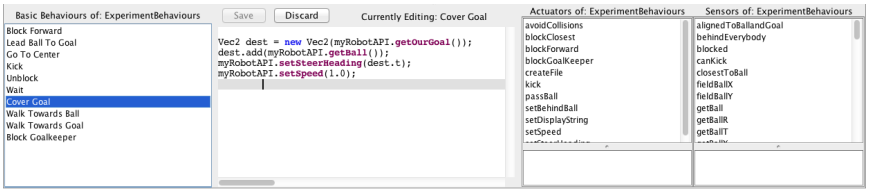


Figura A.2: El editor de comportamientos básicos

de que se conserve la coherencia entre los comportamientos contenidos en ella y el modelo de juego.

Como hemos dicho antes, una biblioteca contiene una colección de comportamientos correspondientes a un juego. Por lo tanto, para crear una nueva biblioteca, el editor necesita recibir el modelo del juego para el que se crea. Este modelo de juego se almacenará en la biblioteca y se utilizará en todos los comportamientos que se añadan a ella. La biblioteca, además, se encarga de gestionar otra información referente a los comportamientos. En concreto, en ella se almacenan las estadísticas recopiladas al ejecutar los comportamientos, los comportamientos ejecutables generados y otros ficheros intermedios.

En las primeras versiones del editor no existía la figura de la biblioteca como tal. Las bibliotecas, en realidad, eran directorios en el sistema de archivos, que el usuario tenía que administrar de manera manual. El modelo de juego, en lugar de estar almacenado en la biblioteca, estaba guardado en cada comportamiento. Esto daba lugar a problemas de coherencia entre comportamientos que, aunque habían sido creados para el mismo juego, no compartían, por ejemplo, los mismo comportamientos básicos. También resultaba problemático realizar cualquier cambio o actualización sobre el conjunto de sensores o actuadores del juego, puesto que había que hacerlo comportamiento por comportamiento. Al introducir la figura de la biblioteca como un "ciudadano de pleno derecho" conseguimos mejorar la coherencia de las colecciones de comportamientos y facilitar los procesos de refactorización del conjunto de sensores y actuadores si estos son necesarios.

Para construir los comportamientos básicos de la biblioteca, los usuarios utilizarán el *editor de comportamientos básicos* (Figura A.2), que se encuentra en la parte de abajo del editor, debajo del lienzo de dibujo. Este editor permite a los usuarios con conocimientos de programación crear nuevos comportamientos básicos o modificar los existentes. Los comportamientos básicos son pequeños fragmentos de código que representan acciones simples que pueden ejecutar las entidades del juego. Los comportamientos básicos se almacenan dentro del modelo de juego y pueden, por lo tanto, ser utilizados por cualquier comportamiento de la biblioteca correspondiente a ese modelo de juego.

La parte izquierda del editor de comportamientos básicos muestra una lista con todos los comportamientos básicos existentes en el modelo de juego

Figura A.3: Barra de herramientas de *eCo*

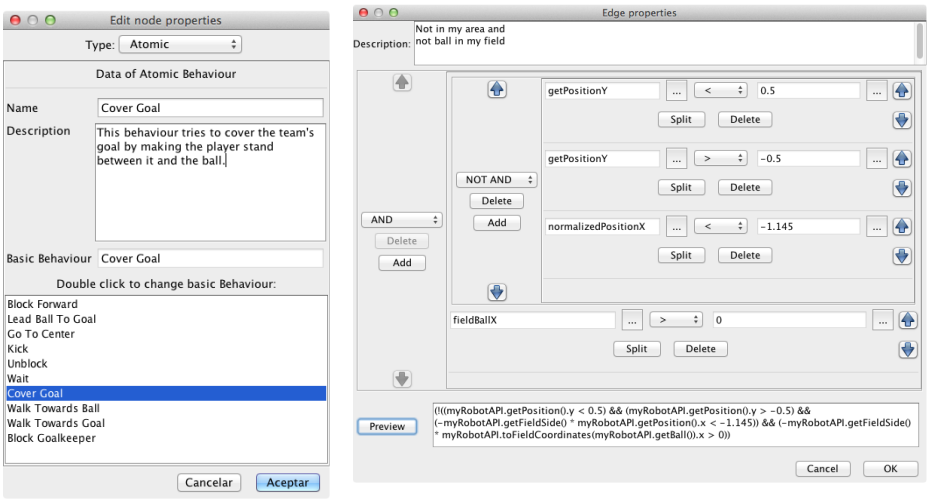
de la biblioteca actual. Un menú contextual permite abrirlos, modificarlos o crear comportamientos nuevos. En el editor de texto que hay a continuación, el usuario puede escribir el código correspondiente al comportamiento básico que está editando en ese momento. Por último, en la parte derecha hay dos listas en las que se muestran todos los sensores y actuadores del modelo de juego. Estas listas sirven como referencia al programador de comportamientos básicos, ya que muestran los sensores y actuadores existentes junto con su descripción. El usuario puede, además, arrastrar los elementos de estas listas hasta el editor de código para insertar una llamada al sensor o actuador correspondiente junto con los parámetros necesarios.

Para editar un comportamiento básico, en primer lugar el usuario debe cargarlo haciendo doble clic sobre él o crear uno nuevo usando el menú contextual. A continuación escribe el código en el editor de texto. Si quiere hacer referencia a algún actuador o sensor del modelo de juego, puede escribirlo directamente en el editor o arrastrarlo desde las listas de la derecha.

La barra de herramientas (Figura A.3), situada en la parte superior de la ventana principal de la GUI, permite realizar la mayoría de operaciones para la administración y edición de bibliotecas y comportamientos. Las dos primeras herramientas permiten crear una nueva biblioteca (1) o cargarla de disco (2). Como hemos dicho anteriormente, para crear una nueva biblioteca es necesario proporcionar al editor un modelo de juego.

Una vez que tenemos una biblioteca de comportamientos, podemos comenzar a crear comportamientos usando el editor. Las siguientes cuatro herramientas se ocupan de la gestión de los comportamientos, permitiendo crear un nuevo comportamiento (3), lo que añadirá un comportamiento vacío a la biblioteca (es decir, sin nodos ni aristas), guardar el comportamiento actual en la biblioteca (4), cargar un comportamiento de la biblioteca (5) e importar un comportamiento sobre el actual (6). La diferencia entre las herramientas de carga y de importación es que, al cargar se cierra el comportamiento actual y se abre el nuevo comportamiento, mientras que en la importación, el comportamiento seleccionado se añade al actual como un subcomportamiento. Esta herramienta resulta útil, por ejemplo, para construir comportamientos complejos a partir de otros comportamientos existentes en la biblioteca.

Mientras la biblioteca de comportamientos está vacía, no se pueden usar las herramientas para reutilizar comportamientos creados y todo el diseño tiene que llevarse a cabo de manera manual. Durante esta etapa, *eCo* facilita las tareas de diseño permitiendo al usuario "dibujar" en el *lienzo de dibujo* que ocupa la parte central del editor los elementos que forman el comportamiento,



(a) Comportamiento asociado a un nodo

(b) Fórmula asociada a una arista

Figura A.4: Edición de las propiedades de las etiquetas asociadas a los nodos y las aristas

es decir, los nodos y las aristas.

Las herramientas que se encuentran a continuación permiten realizar estas tareas de dibujo en el lienzo. Por orden, las herramientas permiten borrar elementos del comportamiento (7), añadir nuevos nodos (8) y aristas (9), seleccionar elementos (10), mover los elementos seleccionados (11), modificar las etiquetas asociadas a los nodos y las aristas (12) y cambiar el nodo inicial (13).

Para añadir los nodos el usuario puede usar la herramienta de creación de nodos de la barra de herramientas. Con ella puede añadir nodos vacíos a su diseño para, a continuación, usando la herramienta de edición de etiquetas, asignarles el contenido. El contenido de un nodo puede ser un comportamiento básico de entre los que han creado los programadores, representado en el lienzo en color azul, u otra máquina de estados subordinada, que se dibujará en color naranja. En la Figura A.4(a) se muestra la página de propiedades de un nodo. En este caso, es un nodo atómico, es decir que contiene un comportamiento básico. El comportamiento básico que se ha asociado al nodo en este caso es *Cover goal*.

Alternativamente, el usuario puede arrastrar el comportamiento deseado hasta el lienzo para añadir de manera más rápida y sencilla un nodo que contenga ese comportamiento. El tipo de nodo que se añadirá depende del origen de la operación de arrastrar. Así, si se arrastra un elemento de la lista de comportamientos básicos, se añadirá un nodo con ese comportamiento

básico. Si se arrastra un elemento de la lista de actuadores, se añadirá un nodo con un nuevo comportamiento básico que corresponde a una llamada al actuador. El comportamiento básico correspondiente al actuador también se añadirá a la lista de comportamientos básicos. Por último, si el origen es un comportamiento de cualquiera de los paneles de búsquedas (navegación, consultas por atributo o sugerencias) se añadirá un nodo compuesto con el comportamiento seleccionado.

Para añadir nuevas aristas a la máquina de estado, el diseñador debe seleccionar la herramienta de creación de aristas y, a continuación, dibujar una línea desde el nodo de origen hasta el de destino. Deberá usar la herramienta de edición de etiquetas para asignar una descripción y una condición a la etiqueta. La Figura A.4(b) muestra la página de propiedades de una arista. Para asignar la condición, el usuario dispone de un editor de fórmulas simples, que permite construir árboles AND/OR (Nilsson, 1998). Cada nodo intermedio de un árbol AND/OR es una conjunción (AND) o una disyunción (OR). Las hojas son proposiciones lógicas del tipo `<valor><comparador><valor>`, donde cada `<valor>` es, o bien una constante, o bien la evaluación de un sensor, y el `<comparador>` puede ser `>`, `>=`, `<`, `<=`, `==` (igual) o `!=` (diferente).

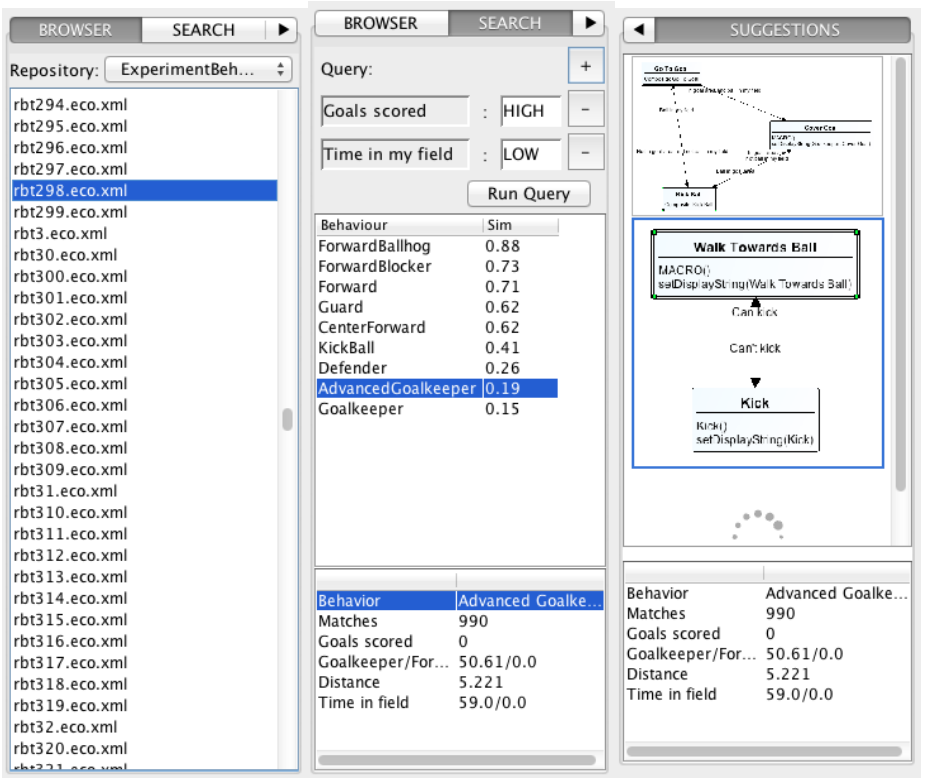
En el formulario de propiedades, el árbol tiene su raíz a la izquierda y va creciendo hacia la derecha. Por ejemplo, el árbol de la figura equivale a la siguiente fórmula:

$$\begin{aligned}\{\neg \quad & \{(\text{getPositionY} < 0{,}5) \quad \wedge \quad (\text{getPositionY} > -0{,}5) \\ & \qquad\qquad\qquad\quad \wedge \quad (\text{normalizedPositionX} > -1{,}145)\}\} \\ \wedge \quad & \{(fieldBallX > 0)\}\end{aligned}$$

El lienzo de dibujo permite, además, explorar la jerarquía de subcomportamientos del coportamiento cargado actualmente. Para profundizar en la jerarquía, el usuario puede utilizar la herramienta de inspección (14) que hemos mencionado antes. Usando esta herramienta en los nodos compuestos (dibujados en naranja en el lienzo) puede abrir en una nueva pestaña el subcomportamiento contenido en ellos. Para saber en todo momento a qué nivel de la jerarquía se encuentra y para poder volver hacia atrás, el usuario cuenta con una estructura tipo *breadcrumb* (Blustein et al., 2005) en la parte superior del lienzo.

Cuando el usuario ha creado suficientes comportamientos en la biblioteca, está listo para utilizar las diferentes herramientas de búsqueda que proporciona el editor. El editor permite realizar tres tipos de búsqueda, representadas en cada una de las pestañas del panel de búsquedas que aparece a la izquierda del lienzo en la Figura A.1.

La primera de las pestañas es la de navegación en bibliotecas (Figura A.5(a)). Esta pestaña sirve para navegar y buscar manualmente comportamientos en las bibliotecas cargadas en el editor. En la parte superior se puede ver un desplegable en el que se puede seleccionar de entre todas las



(a) Bibliotecas (b) Atributos (c) Sugerencias

Figura A.5: Pestañas del panel de búsquedas

bibliotecas cargadas, la biblioteca actual o biblioteca de trabajo. Esta será la biblioteca sobre la que se realizaran todas las operaciones tales como añadir o cargar comportamientos. Debajo del desplegable hay una lista que muestra todos los comportamientos disponibles en la biblioteca actual.

En la Figura A.5(b) se muestra la pestaña de *consultas por atributo*. Esta pestaña permite construir y ejecutar consultas por atributo y muestra los resultados. La parte superior de la pestaña permite construir la consulta añadiendo los diferentes atributos y asignándoles un valor. Como se aprecia en la figura, el usuario puede usar valores numéricos o utilizar valores simbólicos que se traducen en un intervalo. Para representar estos valores simbólicos usamos las 5 palabras clave `LOW`, `MEDIUM LOW`, `MEDIUM`, `MEDIUM HIGH` y `HIGH`. Cada una de ellas representa un intervalo cuyo tamaño es la quinta parte del dominio de valores que puede tomar el atributo. El dominio se especifica en el modelo de juego, indicando los valores mínimo y máximo del atributo. Por ejemplo, para el atributo *Time in my field*, que mide el tiempo que el jugador ha pasado en su propio campo, los valores mínimo y máximo son, respectivamente, 0 y 60, ya que todos los partidos jugados en

la fase de recogida de estadísticas duran 60 segundos. Un valor `LOW` para este atributo indica que buscamos valores en el intervalo $[0, 12]$, o cercanos a él.

Una vez ejecutada la consulta, los resultados se muestran en la lista que hay debajo. Al seleccionar uno de los comportamientos de la lista, en la parte de abajo se muestran sus estadísticas. De esta manera, el usuario puede comparar los comportamientos recuperados entre sí.

La tercera pestaña es la de *sugerencias*, que se muestra en la Figura A.5(c). Cuando esta pestaña está activa, el sistema utiliza la recuperación basada en bocetos para realizar sugerencias sobre comportamientos similares al que está siendo editado actualmente. El funcionamiento es el siguiente: cada cierto tiempo o cuando hay ciertos cambios en el comportamiento editado, el sistema genera una consulta para buscar comportamientos parecidos a él. En realidad, la generación de la consulta, como hemos dicho es sencilla, puesto que, al usar la similitud estructural, la consulta es el propio comportamiento que se está editando. A continuación se compara la consulta con cada uno de los comportamientos de la biblioteca y se añaden a la lista de resultados aquellos que tienen un valor de similitud mayor que una constante definida en las preferencias del usuario.

Los resultados de la consulta se van actualizando mientras el usuario sigue trabajando en el comportamiento actual. Como se ve en la figura, en la parte superior de la pestaña se muestran los comportamientos y debajo, las estadísticas del comportamiento seleccionado.

En los tres casos anteriores, el usuario puede incorporar un comportamiento de manera rápida como subcomportamiento al que está editando simplemente arrastrándolo desde cualquiera de las listas anteriores. También puede abrirlo para editarlo haciendo doble clic sobre él.

El editor incorpora en la barra de herramientas, además, una herramienta de búsqueda (15) que permite realizar consultas por estructura a petición del usuario. En lugar de mantener activa la pestaña de sugerencias, el usuario puede realizar una consulta independiente usando esta herramienta.

Al activar la herramienta se muestra un formulario como el que aparece en primer plano en la Figura A.6 donde el usuario puede introducir el número máximo de resultados que va a devolver la consulta. Además puede activar la opción de usar todos los subcomportamientos como casos. Si la opción no está activada, la consulta se compara con cada uno de los comportamientos de la biblioteca, pero si se activa esta opción, además se comparará con cada uno de los subcomportamientos.

Al ejecutar la consulta, se muestra un formulario de resultados como el que aparece en segundo plano en la figura. Usando este formulario, el usuario puede navegar a través de los comportamientos recuperados usando la herramienta de inspección que mencionamos antes para seleccionar un comportamiento o un subcomportamiento. También puede seleccionar un subconjunto de nodos y aristas que pertenezcan a un comportamiento. Cuando el usuario

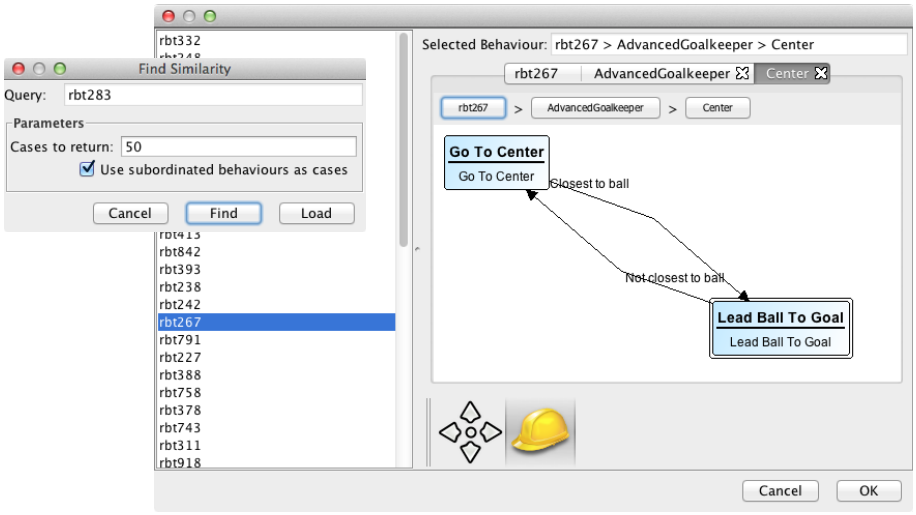


Figura A.6: Formulario de parámetros y formulario de resultados para la herramienta de consultas

acepta, el comportamiento, subcomportamiento o selección que ha realizado se añade al comportamiento actual.

Las restantes herramientas de la barra realizan tareas que no afectan directamente a la edición o gestión de comportamientos o bibliotecas, pero que son igualmente importantes durante el proceso de diseño de comportamientos basado en la reutilización.

La herramienta de ejecución (16) invoca a un *ejecutor de comportamientos* apropiado para el comportamiento y el juego. La tarea del ejecutor es realizar una generación automática del comportamiento ejecutable y lanzarlo en el juego, permitiendo realizar pruebas de manera rápida sin necesidad de realizar todo el proceso de generación.

Por ejemplo, en el caso de SBTournament, cada vez que quisieramos realizar una prueba, sería necesario generar el comportamiento ejecutable, que es un paquete Java formado por tres clases. A continuación habría que compilarlo junto con SBTournament y copiarlo en la carpeta de comportamientos de SBTournament. Por último, habría que arrancar SBTournament y configurar el partido. Esta herramienta realiza automáticamente todas estas tareas, ahorrando tiempo a los diseñadores. El inconveniente de los ejecutores es que deben ser programados a medida para el juego para el que estamos diseñando los comportamientos y es posible que, en algunos casos, el proceso de lanzar el juego y ejecutar el comportamiento automáticamente no se pueda realizar.

En la Figura A.7, se muestra el formulario de configuración del ejecutor de SBTournament. En el formulario se pueden introducir unos valores de configuración para el partido, que el editor almacenará de una ejecución a

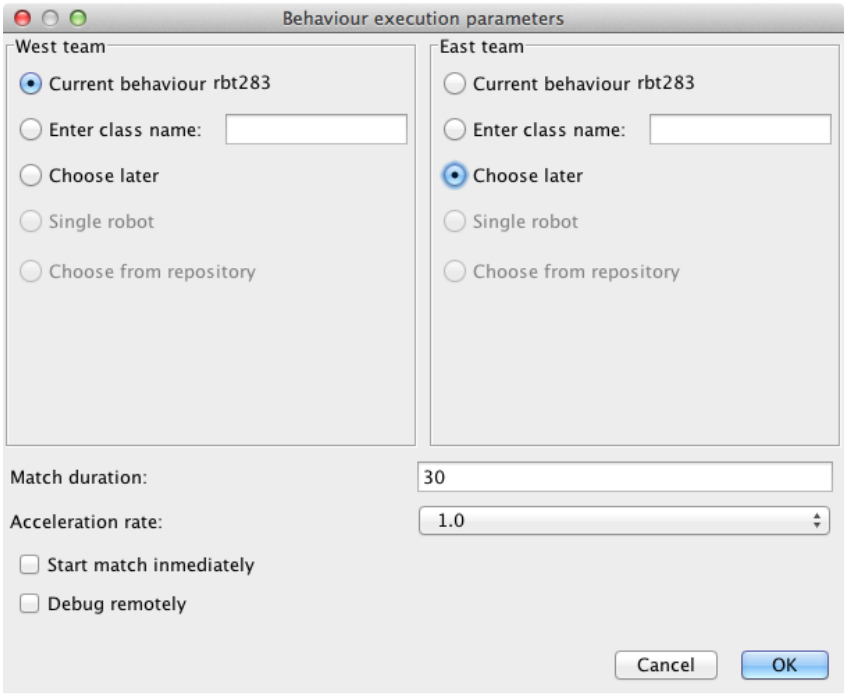


Figura A.7: Ejecutor de comportamientos para SBTournament

la siguiente, de manera que el usuario pueda realizar una batería de pruebas similares seguidas.

Para configurar el partido, lo más importante es introducir los equipos que van a jugar. El ejecutor permite seleccionar el equipo editado actualmente, uno de los equipos incluidos en SBTournament, escribiendo el nombre de su clase Java, permitir al usuario seleccionar el equipo cuando se ejecute SBTournament, un equipo formado por un solo robot con el comportamiento actual (los restantes no tendrán ningún comportamiento) u otro comportamiento incluido en el repositorio actual.

Por último se pueden ajustar tanto el tiempo de duración del partido como la tasa de aceleración. Si se marca la opción de comienzo inmediato, el partido se lanza inmediatamente sin que el usuario tenga que activar ningún control en SBTournament. La opción de depuración remota permite depurar los comportamientos usando las opciones de depuración remota de la JVM (*Java Virtual Machine*, Máquina Virtual Java).

La última de las herramientas de la que vamos a hablar es el generador de código (17) (las dos restantes son los botones de deshacer y rehacer). Mediante esta herramienta se genera el comportamiento ejecutable correspondiente

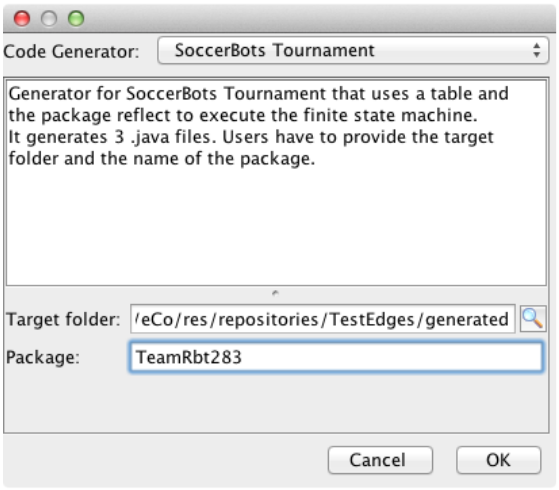


Figura A.8: Generador de comportamientos para SBTournament

al comportamiento actual. Igual que en el caso anterior, esta herramienta depende del juego para el que se están desarrollando los comportamientos. Por lo tanto, tendremos, al menos, un generador por cada modelo de juego. Al igual que con los ejecutores, los generadores también tienen que ser programados *ad hoc* para el juego en cuestión.

Una vez que el usuario ha finalizado la edición del comportamiento, utilizará el generador para obtener el comportamiento ejecutable que será incluido en el juego. En la Figura A.8 se muestra el formulario de configuración para el generador de SBTournament.

En conclusión, el editor ofrece herramientas para trabajar a 3 niveles.

- A nivel de los repositorios: el editor permite crearlos en disco, cambiar su ubicación y eliminarlos. También permite combinar varios repositorios utilizando un asistente mediante el cual se puede comprobar qué sensores, actuadores y, sobre todo, qué comportamientos básicos son comunes a los repositorios combinados y, para los que no lo son, seleccionar uno de ellos o modificarlos. De esta manera se obtendrá un nuevo repositorio con los comportamientos de los repositorios originales y con una combinación de los elementos que componen el modelo de juego de manera que los comportamientos sean coherentes con este. El objetivo de este asistente es facilitar el diseño colaborativo y la actualización de comportamientos a diferentes versiones del mismo juego.

- A nivel de los comportamientos básicos: los usuarios disponen de he-

rramientas para añadirlos y eliminarlos del modelo de juego del repositorio actual. Además, existe un editor de código integrado donde pueden programar nuevos comportamientos básicos.

- A nivel de los comportamientos: existen herramientas para crear, cargar y guardar los comportamientos en un repositorio. Para poder editar los comportamientos también existen herramientas que permiten "dibujarlos", añadiendo y borrando los nodos y aristas que los componen, o modificar las etiquetas asociadas a estos elementos. El editor permite también importar en el comportamiento que está siendo editado partes de otro comportamiento del repositorio actual. Para localizar los comportamientos que pueden ser importados el editor ofrece diferentes herramientas de búsqueda.

Bibliografía

Hierarchical plan representations for encoding strategic game ai. En *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. AAAI Press, 2005.

Real time game loop models for single-player computer games. 2005.

Aamodt, A. y Plaza, E. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, vol. 7, páginas 39–59, 1994.

Aleson, R. S. y Louis, S. J. Developing adaptive tactical aggressors. En *The Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*. 2006.

Alexander, G., Youngblood, G. M., Heckel, F. W. P., Hale, D. H. y Ketkar, N. S. Rapid development of intelligent agents in first/third-person training simulations via behavior-based control. En *Proceedings of the 19th Behavior Representation in Modeling and Simulation Conference (BRIMS)*. 2010.

Arévalo, J. Videojuegos: Arte, diversión, ingeniería, negocio. `http://www.iguanademos.com/Jare/docs/2005UCMVideojuegos.zip`, 2005. Último acceso: 2012-08-22. Archivado como `http://www.webcitation.org/6A6ksYj1F`.

Atkin, M. S., King, G. W., Westbrook, D. L., Heeringa, B. y Cohen, P. R. Hierarchical agent control: a framework for defining agent behavior. En *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, páginas 425–432. 2001.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. y Patel-Schneider, P. F., editores. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. ISBN 0-521-78176-0.

Bates, B. *Game Design, 2nd Edition*. Course Technology Press, Boston, MA, United States, 2004. ISBN 1592004938.

BLUSTEIN, J., AHMED, I. y INSTONE, K. An evaluation of look-ahead breadcrumbs for the www. En *Proceedings of the sixteenth ACM conference on Hypertext and hypermedia*, HYPERTEXT '05, páginas 202–204. ACM, New York, NY, USA, 2005. ISBN 1-59593-168-6.

BOLLMANN, P. y RAGHAVAN, V. V. A utility-theoretic analysis of expected search length. En *Proceedings of the 11th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '88, páginas 245–256. ACM, New York, NY, USA, 1988. ISBN 2-7061-0309-4.

BOROVIKOV, I. y KADUKIN, A. Building a behavior editor for abstract state machines. En *AI Game Programming Wisdom 4*, páginas 333–346. Charles River Media, 2008.

BOURG, D. M. y SEEMANN, G. *AI for Game Developers*. O'Reilly Media, Inc., 2004. ISBN 0596005555.

BOWLING, M. H., FÜRNKRANZ, J., GRAEPEL, T. y MUSICK, R. Machine learning and games. *Machine Learning*, vol. 63(3), páginas 211–215, 2006.

BROOKS, R. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, vol. 2(1), páginas 14–23, 1986. ISSN 0882-4967.

BROWNLEE, J. Finite state machines. `http://ai-depot.com/FiniteStateMachines/FSM.html`, 2002. Último acceso: 2012-12-03. Archivado como `http://www.webcitation.org/6CdbAG2w0`.

BUCHANAN, W. *Game Programming Gems 5*, capítulo A Generic Component Library. Charles River Media, 2005. ISBN 1-584-50352-1.

BUNKE, H. y MESSMER, B. T. Similarity measures for structured representations. En *EWCBR '93: Selected papers from the First European Workshop on Topics in Case-Based Reasoning*, páginas 106–118. Springer-Verlag, London, UK, 1994. ISBN 3-540-58330-0.

BURKARD, R. E., DELLÁMICO, M. y MARTELLO, S. *Assignment Problems*. SIAM, 2009. ISBN 978-0-89871-663-4.

CHAMPANDARD, A. *Ai Game Development: Synthetic Creatures With Learning and Reactive Behaviors*. New Riders Games Series. New Riders, 2003. ISBN 9781592730049.

CHAMPANDARD, A. J. Choosing a hierarchical fsm or a hierarchy of nested fsms? `http://aigamedev.com/open/article/hierarchical-or-nested-fsm/`, 2007. Último acceso: 2012-08-31. Archivado como `http://www.webcitation.org/6AKdzkv0a`.

CHAMPIN, P. A. y SOLNON, C. Measuring the similarity of labeled graphs. En *5th Int. Conf. On Case-Based Reasoning (ICCBR 2003)* (editado por K. D. Ashley y D. G. Bridge), LNAI, páginas 80–95. Springer, 2003.

CHANDLER, H. *The Game Production Handbook (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA, 2005. ISBN 1584504161.

CRYTEK. CryENGINE 3 SDK, Sandbox Editor. 2010. Http://mycryengine.com.

CUNNINGHAM, P. A taxonomy of similarity mechanisms for case-based reasoning. *IEEE Transactions on Knowledge and Data Engineering*, vol. 21(11), páginas 1532–1543, 2009. ISSN 1041-4347.

D. LEAKE, E. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Menlo Park, CA: AAAI Press/MIT Press, Menlo Park, CA., 1996.

ESMURDOC, C. Head games: Double fine's psychonautic break. *Game Developer Magazine*, vol. 12(7), páginas 30–38, 2005.

FLÓREZ-PUGA, G., BELÉN DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Evaluating sketch-based retrieval speed-up for behaviour design in soccerbots. En *Proceedings of the 26 International Florida Artificial Intelligence Research Society Conference*. 2013. (enviado).

FLÓREZ-PUGA, G. y DÍAZ-AGUDO, B. Semiautomatic edition of behaviours in videogames. En *Proceedings of AI2007, 12th UK Workshop on Case-Based Reasoning*. 2007.

FLÓREZ PUGA, G., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. Experience-based design of behaviors in videogames. En *Proceedings of the 9th European conference on Advances in Case-Based Reasoning*, ECCBR '08, páginas 180–194. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85501-9.

FLÓREZ-PUGA, G., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Similarity measures in hierarchical behaviours from a structural point of view. En *FLAIRS 2010* (editado por H. W. Guesgen y R. C. Murray), páginas 330–335. AAAI Press, 2010.

FLÓREZ-PUGA, G., GÓMEZ-MARTÍN, M. A., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Dynamic expansion of behaviour trees. En *AIIDE* (editado por C. Darken y M. Mateas), páginas 36–41. The AAAI Press, 2008. ISBN 978-1-57735-391-1.

FLÓREZ-PUGA, G., GÓMEZ-MARTÍN, M. A., GÓMEZ-MARTÍN, P. P., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Query-enabled behavior trees. *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 1(4), páginas 298–308, 2009.

FLÓREZ-PUGA, G., GONZÁLEZ-CALERO, P. A., JIMÉNEZ-DÍAZ, G. y DÍAZ-AGUDO, B. Supporting sketch-based retrieval from a library of reusable behaviours. *Expert Systems with Applications*, vol. 40(2), páginas 531–542, 2012a. ISSN 0957-4174.

FLÓREZ-PUGA, G., JIMÉNEZ-DÍAZ, G. y GONZÁLEZ-CALERO, P. A. eco: Managing a library of reusable behaviours. En *Proceedings of the 20th International Conference on Case-Based Reasoning* (editado por B. Díaz-Agudo y I. Watson), vol. 7466 de *Lecture Notes in Computer Science*. Springer, 2012b.

FLÓREZ-PUGA, G., LLANSÓ, D., GÓMEZ-MARTÍN, M. A., GÓMEZ-MARTÍN, P. P., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. *Empowering Designers with Libraries of Self-validated Query-enabled BehaviourTrees*, páginas 55–82. Springer, 2011. ISBN 978-1-4419-8187-5.

FU, D. y HOULETTE, R. T. Putting ai in entertainment: An ai authoring tool for simulation and games. *IEEE Intelligent Systems*, vol. 17(4), páginas 81–84, 2002.

GAMES, E. Unreal Development Kit. 2009. Http://www.udk.com/.

GARCÉS, S. Strategies for multiprocessor AI. En *AI Game Programming Wisdom 3*, páginas 65–76. Charles River Media, 2006. ISBN 1-584-50457-9.

GAREY, M. R. y JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

GERBAUD, S., MOLLET, N., GANIER, F., ARNALDI, B. y TISSEAU, J. Gvt: a platform to create virtual environments for procedural training. En *IEEE Virtual Reality Conference 2008 (VR 2008)*, páginas 225–232. 2008.

GIL, Y., KIM, J., FLÓREZ-PUGA, G., RATNAKAR, V. y GONZÁLEZ-CALERO, P. A. Workflow matching using semantic metadata. En *Proceedings of the Fifth International Conference on Knowledge Capture (K-CAP)*, páginas 121–128. Redondo Beach, CA, 2009.

GILGENBACK, M. y MCINTOSH, T. A flexible AI system through behavior compositing. En *AI Game Programming Wisdom 3*, páginas 289–300. Charles River Media, 2006.

GIRAULT, A., LEE, B. y LEE, E. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design*, vol. 18(6), páginas 742–760, 1999. Research report UCB/ERL M97/57.

GLICKMAN, M. E. Chess rating systems. *American Chess Journal*, vol. 3, páginas 59–102, 1995. ISSN 1066-8292.

GÓMEZ-MARTÍN, P. P., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Javy: Virtual environment for case-based teaching of java virtual machine. En *KES* (editado por V. Palade, R. J. Howlett y L. C. Jain), vol. 2773 de *Lecture Notes in Computer Science*, páginas 906–913. Springer, 2003. ISBN 3-540-40803-7.

GÓMEZ-MARTÍN, P. P., GÓMEZ-MARTÍN, M. A., GONZÁLEZ-CALERO, P. A. y PALMIER-CAMPOS, P. Using metaphors in game-based education. En *Edutainment'07* (editado por K. chuen Hui, Z. Pan, R. C. kit Chung, C. C. Wang, X. Jin, S. Göbel y E. C.-L. Li), vol. 4469 de *LNCS*, páginas 477–488. SV, 2007. ISBN 3-540-73010-9.

GORNIAK, P. y DAVIS, I. Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters. En *AIIDE* (editado por J. Schaeffer y M. Mateas), páginas 14–19. The AAAI Press, 2007. ISBN 978-1-57735-325-6.

HARCHAOUI, Z. y BACH, F. Image classification with segmentation graph kernels. En *CVPR*. IEEE Computer Society, 2007.

HAREL, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, vol. 8(3), páginas 231–274, 1987. ISSN 0167-6423.

HECKEL, F. W. P., YOUNGBLOOD, G. M. y HALE, D. H. Behaviorshop: An intuitive interface for interactive character design. En *AIIDE* (editado por C. Darken y G. M. Youngblood). The AAAI Press, 2009. ISBN 978-1-57735-431-4.

HECKEL, F. W. P., YOUNGBLOOD, G. M. y KETKAR, N. S. Representational complexity of reactive agents. En *CIG* (editado por G. N. Yannakakis y J. Togelius), páginas 257–264. IEEE, 2010. ISBN 978-1-4244-6295-7.

HOCKING, C. Ubisoft Montreal's Far Cry 2 Postmortem. *Game Developer Magazine*, páginas 30–38, 2009.

HOULETTE, R., FU, D. y ROSS, D. Towards an ai behavior toolkit for games. En *Papers from 2001 AAAI Spring Symposium in Artificial Intelligence and Interactive Entertainment* (editado por J. Laird y M. van Lent), páginas 50–53. 2001.

IERUSALIMSCHY, R. *Programming in Lua*. Lua.org, second edición, 2006. ISBN 9788590379829.

ISLA, D. Handling Complexity in the Halo 2 AI. En *Game Developers Conference*. 2005.

ISLA, D. Halo 3 - building a better battle. En *Game Developers Conference*. 2008.

JIMÉNEZ-DÍAZ, G. y DÍAZ-AGUDO, B. SB Tournament: Competiciones de robots en asignaturas de Inteligencia Artificial. En *Procs. of the 9th edition of the International Symposium on Computers in Education. SIIE*. 2007. ISBN 978-972-8969-04-2.

JIMÉNEZ-DÍAZ, G., MENÉNDEZ, H. D., CAMACHO, D. y GONZÁLEZ-CALERO, P. A. Predicting performance in team games. the automatic coach. En *3rd International Conference on Agents and Artificial Intelligence (ICAART 2011)*, páginas 401 – 406. SciTePress, Rome, Italy, 2011.

KO, A. J., ABRAHAM, R., BECKWITH, L., BLACKWELL, A. F., BURNETT, M. M., ERWIG, M., SCAFFIDI, C., LAWRANCE, J., LIEBERMAN, H., MYERS, B. A., ROSSON, M. B., ROTHERMEL, G., SHAW, M. y WIEDENBECK, S. The state of the art in end-user software engineering. *ACM Comput. Surv.*, vol. 43(3), página 21, 2011.

KRAJEWSKI, J. Creating all humans: A data-driven AI framework for open game worlds. *Gamasutra*, 2009.

KÜCKLICH, J. Precarious Playbour: Modders and the Digital Games Industry. *Fibreculture*, (5), 2005.

LAKOS, J. *Large Scale C++ Software Design*. Addison Wesley, 1996. ISBN 0-201-63362-0.

LEONARD, T. Building an ai sensory system: Examining the design of thief: The dark project. En *Game Developer's Conference Proceedingsz*. 2003.

LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, vol. 10(8), páginas 707–710, 1966.

LLANSÓ, D., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Self-validated behaviour trees through reflective components. En *Proceedings, The Fifth AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, páginas 76–81. AAAI Press, 2009. ISBN 978-1-57735-431-4.

LOYALL, A. y BATES, J. *Hap: a reactive, adaptive architecture for agents*. Número v. 91-147 en Research paper. School of Computer Science, Carnegie Mellon University, 1991.

MANNING, C. D., RAGHAVAN, P. y SCHTZE, H. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

MATEAS, M. y STERN, A. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, vol. 17(4), páginas 39–47, 2002.

MILES, C., LOUIS, S. J., COLE, N. y MCDONNELL, J. Learning to play like a human: case injected genetic algorithms for strategic computer gaming. En *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 2, páginas 1441–1448 Vol.2. 2004.

MILLINGTON, I. *Artificial intelligence for games*. The Morgan Kaufmann series in interactive 3D technology. Morgan Kaufmann Publishers Inc., 2006. ISBN 9780124977822.

NICOLESCU, M. N. y MATARIĆ, M. J. A hierarchical architecture for behavior-based robots. En *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, páginas 227–233. ACM, New York, NY, USA, 2002. ISBN 1581134800.

NILSSON, N. J. *Artificial intelligence: a new synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-467-7.

OLENDERSKI, A., NICOLESCU, M. N. y LOUIS, S. J. A behavior-based architecture for realistic autonomous ship control. En *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG06)* (editado por S. J. Louis y G. Kendall), páginas 148–155. IEEE, 2006.

ONTAÑÓN, S., MISHRA, K., SUGANDH, N. y RAM, A. Learning from demonstration and case-based planning for real-time strategy games. En *Soft Computing Applications in Industry* (editado por B. Prasad), páginas 293–310. Springer, 2008.

PILLOSU, R. Coordinating agents with behaviour trees. `http://staff.science.uva.nl/~aldersho/GameProgramming/Papers/Coordinating_Agents_with_Behaviour_Trees.pdf`, 2009. Último acceso: 2012-08-22. Archivado como `http://www.webcitation.org/6A6iIWHPP`.

RABIN, S., editor. *AI Game Programming Wisdom 3*. Charles River Media, 2006.

RABIN, S., editor. *AI Game Programming Wisdom 4*. Charles River Media, 2008.

REMO, C. y SHEFFIELD, B. Redefining game narrative: Ubisoft's patrick redding on far cry 2. *Gamasutra*, 2008.

RENE, B. *Game Programming Gems 5*, capítulo Component Based Object Management. Charles River Media, 2005. ISBN 1-584-50352-1.

RIESEN, K. y BUNKE, H. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, vol. 27(7), páginas 950–959, 2009.

ROUSE, R. y OGDEN, S. *Game design: theory & practice*. Wordware Game Developer's Library. Wordware Publishing Inc., 2nd edición, 2005. ISBN 9781556229121.

SMITH, G., WHITEHEAD, J., MATEAS, M., TREANOR, M., MARCH, J. y CHA, M. Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 3(1), páginas 1–16, 2011.

SMYTH, B. y CUNNINGHAM, P. The utility problem analysed: A case-based reasoning perspective. En *EWCBR*, páginas 392–399. 1996.

SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997. ISBN 0201178885.

TAI, K.-C. The tree-to-tree correction problem. *J. ACM*, vol. 26(3), páginas 422–433, 1979. ISSN 0004-5411.

UBISOFT MONTREAL STUDIOS. Far Cry 2. 2008.

WAGNER, R. A. y FISCHER, M. J. The string-to-string correction problem. *J. ACM*, vol. 21(1), páginas 168–173, 1974. ISSN 0004-5411.

WANG, J. T. L., SHAPIRO, B. A., SHASHA, D., ZHANG, K. y CURREY, K. M. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, páginas 889–895, 1998.

WANG, Y. y ISHII, N. A method of similarity metrics for structured representations. *Expert Systems with Applications*, vol. 12(1), páginas 89–100, 1997.

WEST, M. Evolve your hierarchy. *Game Developer*, vol. 13(3), páginas 51–54, 2006.

WITTEN, I. H. y FRANK, E. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2 edición, 2005. ISBN 0120884070.

YU, L. y LIU, H. Feature selection for high-dimensional data: A fast correlation-based filter solution. En *in ICML*, páginas 856–863. 2003.

ZHANG, K. y SHASHA, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, vol. 18(6), páginas 1245–1262, 1989. ISSN 0097-5397.

Lista de acrónimos

BT	*Behaviour Tree*, Árbol de Comportamiento
CBR	*Case Based Reasoning*, Razonamiento Basado en Casos
FPS	*First Person Shooter*
FSM	*Finite State Machine*, Máquina de Estado Finito
GUI	*Graphic User Interface*, Interfaz Gráfica de Usuario
HFSM	*Hierarchical Finite State Machine*, Máquina de Estado Finito Jerárquica
IA	Inteligencia Artificial
JVM	*Java Virtual Machine*, Máquina Virtual Java
kNN	*k Nearest Neighbours*, k vecinos más cercanos
NPC	*Non Playing Character*, Personaje No Jugador
UDK	*Unreal Development Kit*