



**UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR**

INGENIERÍA EN INFORMÁTICA

PROYECTO FIN DE CARRERA

**ARAN, UN SISTEMA DE
PLANIFICACIÓN ORIENTADO A
VIDEOJUEGOS**

Autor: Abel García Plaza
Tutores: Daniel Borrajo Millán
Carlos Linares López

Mayo de 2008

Para ti, Arancha

**ARAN, UN SISTEMA DE
PLANIFICACIÓN ORIENTADO A
VIDEOJUEGOS**

Lo bien hecho, bien parece

Agradecimientos

A Arancha, mi novia. Porque, sin saber nada de informática, ha sabido formar parte de un equipo que, sin su ayuda, no habría logrado los mismos resultados. Su colaboración con las traducciones al español queda en nada al lado de las aportaciones que sólo una personalidad como la suya puede ofrecer. Su paciencia, su comprensión y su apoyo incondicional quedarán siempre en el lugar más visible de este proyecto: su nombre.

A mi familia en general y a mis padres en particular, por haberme inculcado la cultura del esfuerzo y la superación, siendo ellos el mejor de los ejemplos.

A todos mis compañeros de universidad, por haber luchado a mi lado en las batallas de la guerra por la consecución del título de Ingeniero en Informática.

A todos los compañeros de trabajo que he tenido en cada empresa por la que he pasado, por su aportación a mi mejora profesional y, sobre todo, personal.

A todos los que me han criticado alguna vez, porque me han ayudado a seguir superándome.

A todos los profesores que he tenido a lo largo de la carrera, porque lo exigente de sus métodos ha aprobado con nota la asignatura de la enseñanza. A mis tutores en particular, Daniel Borrajo y Carlos Linares, por darme la libertad y la confianza que necesitaba, y a Tomás de la Rosa por la ayuda prestada explicándome SAYPHI.

A la Ingeniería en Informática como disciplina, a la Universidad como institución de su difusión y a la Carlos III como representante de ésta.

A todos los que, de algún modo, han compartido conmigo su conocimiento.



Índice

1	Introducción.....	9
2	Estado de la cuestión	13
2.1	Videojuegos.....	13
2.1.1	Historia y evolución	13
2.1.2	Desarrollo no profesional en Xbox 360.....	18
2.2	Inteligencia Artificial en videojuegos.....	25
2.2.1	Técnicas comunes.....	26
2.2.2	Técnicas prometedoras	31
2.3	Planificación automática.....	36
2.3.1	Noción básica	36
2.3.2	Planificación clásica	40
3	Objetivos.....	45
4	Desarrollo	49
4.1	Metodología.....	49
4.2	Análisis	50
4.2.1	Actores.....	51
4.2.2	Casos de uso	52
4.3	Diseño conceptual.....	54
4.3.1	Arquitectura del sistema	54
4.3.2	Subsistema de representación genérica	60
4.3.3	Subsistema de representación específica	69
4.3.4	Subsistema núcleo	76
4.3.5	Subsistema de extensión funcional.....	92
4.3.6	Subsistema de búsqueda genérica	96
4.3.7	Subsistema de búsqueda específica	100
4.3.8	Subsistema de traducción	104
4.3.9	Subsistema de planificación	116
4.3.10	Subsistema de interacción	119
4.4	Diseño detallado	121
4.4.1	Espacios de nombres	122
4.4.2	Espacio de nombres Aran.Representation.....	123
4.4.3	Espacio de nombres Aran.Representation.Xml.....	125
4.4.4	Espacio de nombres Aran.Core	127
4.4.5	Espacio de nombres Aran.Functional.....	131
4.4.6	Espacio de nombres Aran.Searching.....	133
4.4.7	Espacio de nombres Aran.Searching.Specialized.....	135
4.4.8	Espacio de nombres Aran.Translation.....	137
4.4.9	Espacio de nombres Aran.Planning.Xml.....	140
4.4.10	Espacio de nombres Aran.Planning.Xml.Command.....	141
4.5	Implementación	142
4.5.1	Entorno de desarrollo.....	142
4.5.2	Ficheros fuente	144
4.5.3	Optimización del rendimiento	148
4.5.4	Diferencias entre versiones de PC y Xbox 360	152
4.5.5	Productos generados	154
5	Resultados.....	161
5.1	Entorno de pruebas	161
5.2	Descripción del dominio.....	162



5.3	Descripción de los problemas.....	163
5.4	Presentación de los datos obtenidos	168
5.5	Análisis de los datos obtenidos.....	175
6	Conclusiones.....	193
7	Líneas futuras	201
8	Referencias	209
9	Anexo A: Gramática del lenguaje	217
9.1	Notación	217
9.2	Reglas	217
10	Anexo B: Ejemplos de utilización del lenguaje.....	225
10.1	Ejemplo de dominio	225
10.2	Ejemplo de problema.....	230
11	Anexo C: Manual de usuario	237
11.1	Utilización del comando desde la consola del sistema.....	237
11.2	Utilización de la funcionalidad del comando desde una aplicación o videojuego	239
11.3	Utilización directa de los componentes del sistema desde una aplicación o videojuego	241
11.3.1	Creación del sistema de representación.....	243
11.3.2	Creación del traductor	243
11.3.3	Ejecución del traductor.....	245
11.3.4	Creación del algoritmo de búsqueda	245
11.3.5	Ejecución del algoritmo de búsqueda.....	248
11.3.6	Extracción del plan	249



Índice de ilustraciones

Ilustración 1: Videojuegos Spacewar y Assassin's Creed	14
Ilustración 2: Videoconsolas Odyssey y Xbox 360	15
Ilustración 3: Políticos haciendo uso de los videojuegos	16
Ilustración 4: Videoconsola Xbox y logotipo de DirectX	19
Ilustración 5: Cuotas de mercado en la 7ª generación de videoconsolas	19
Ilustración 6: Arquitectura de desarrollo para Xbox 360	20
Ilustración 7: Logotipo de XNA	22
Ilustración 8: Pasos a seguir en Xbox LIVE Community Games	24
Ilustración 9: Diagrama intuitivo de un proceso de planificación	37
Ilustración 10: Diagrama intuitivo, de mayor detalle, de un proceso de planificación ..	40
Ilustración 11: Diagrama de casos de uso	53
Ilustración 12: Diagrama de componentes del sistema	55
Ilustración 13: Diagrama de interacciones entre los componentes del sistema	57
Ilustración 14: Diagrama de flujo del sistema	58
Ilustración 15: Niveles de abstracción	59
Ilustración 16: Representación conceptual de un estado	78
Ilustración 17: Representación conceptual de una acción	81
Ilustración 18: Primer paso de la selección de acciones	88
Ilustración 19: Grafo de planificación relajada	91
Ilustración 20: Representación conceptual de una operación de bajo nivel	93
Ilustración 21: Tipos de referencia a variable, directa e indirecta	95
Ilustración 22: Pseudocódigo del algoritmo Best First	98
Ilustración 23: Pseudocódigo del algoritmo Hill Climbing	99
Ilustración 24: Pseudocódigo del algoritmo Beam Search	99
Ilustración 25: Pseudocódigo del algoritmo Enforced Hill Climbing	103
Ilustración 26: Pseudocódigo del algoritmo de búsqueda de un mejor sucesor	103
Ilustración 27: Traducción de objetos a números	107
Ilustración 28: Traducción visual de argumentos de una <i>instancia</i>	108
Ilustración 29: Decodificación de argumentos de una <i>instancia</i>	110
Ilustración 30: Agrupación de números de <i>instancia</i>	110
Ilustración 31: Reasignación de números de <i>instancias</i> máximas	113
Ilustración 32: Reasignación de números de <i>instancias</i> mínimas	113
Ilustración 33: Diagrama de clases de Aran.Representation	123
Ilustración 34: Diagrama de clases de Aran.Representation.Xml	126
Ilustración 35: Diagrama de clases de Aran.Core	128
Ilustración 36: Diagrama de clases de Aran.Functional	131
Ilustración 37: Diagrama de clases de Aran.Searching	133
Ilustración 38: Diagrama de clases de los algoritmos de Aran.Searching	134
Ilustración 39: Diagrama de clases de Aran.Searching.Specialized	135
Ilustración 40: Diagrama de clases de Aran.Translation	137
Ilustración 41: Diagrama de la clase Aran.Translation.MessageManager ..	139
Ilustración 42: Diagrama de clases de mensajes en Aran.Translation	139
Ilustración 43: Diagrama de clases de Aran.Planning.Xml	141
Ilustración 44: Diagrama de clases de Aran.Planning.Xml.Command	142
Ilustración 45: Documentación de referencia	156
Ilustración 46: Instalador para Windows	157
Ilustración 47: Integración de Aran con Visual Studio	158



Ilustración 48: Logotipo de Aran.....	158
Ilustración 49: Ejemplo de problema para el dominio de pruebas	162
Ilustración 50: Captura de pantalla del problema 1	164
Ilustración 51: Captura de pantalla del problema 2	164
Ilustración 52: Captura de pantalla del problema 3	164
Ilustración 53: Captura de pantalla del problema 4	165
Ilustración 54: Captura de pantalla del problema 5	165
Ilustración 55: Captura de pantalla del problema 6	165
Ilustración 56: Captura de pantalla del problema 7	166
Ilustración 57: Captura de pantalla del problema 8	166
Ilustración 58: Captura de pantalla del problema 9	167
Ilustración 59: Captura de pantalla del problema 10	167
Ilustración 60: Rendimiento en tiempo del traductor	176
Ilustración 61: Relación entre complejidad de problema y tiempo de traducción	177
Ilustración 62: Variables en el problema frente a representadas en el estado	178
Ilustración 63: Hechos en el problema frente a representados en el estado	179
Ilustración 64: Impacto del número de hechos en el tamaño del estado	179
Ilustración 65: Relación entre el número de acciones y la memoria	180
Ilustración 66: Comparativa entre soluciones encontradas y óptimas.....	182
Ilustración 67: Diferencias porcentuales entre soluciones y óptimos.....	183
Ilustración 68: Optimalidad de los algoritmos en nodos, tiempo y coste.....	184
Ilustración 69: Mejor tiempo de búsqueda por problema.....	185
Ilustración 70: Memoria consumida por cada mejor tiempo de búsqueda.....	185
Ilustración 71: Rendimiento de los algoritmos en nodos / ms.....	186
Ilustración 72: Tiempo de búsqueda para los problemas 7 y 8	187
Ilustración 73: Diferencias entre soluciones y óptimo para el problema 8	188
Ilustración 74: Ejecución por defecto del comando aran.exe	237
Ilustración 75: Ejemplos de uso del comando aran.exe.....	239
Ilustración 76: Ejemplo de uso de la biblioteca Aran.Planning.Xml.dll	240
Ilustración 77: Ejemplo de parametrización de Aran.Planning.Xml.dll.....	241
Ilustración 78: Ejemplo de creación de un sistema de representación	243
Ilustración 79: Ejemplo de creación del traductor.....	244
Ilustración 80: Ejemplo de configuración del traductor	244
Ilustración 81: Ejemplo de ejecución del traductor	245
Ilustración 82: Ejemplo de creación del proveedor del estado inicial.....	246
Ilustración 83: Ejemplo de creación de la función de validación de meta	246
Ilustración 84: Ejemplo de creación del generador de sucesores	247
Ilustración 85: Ejemplo de creación de la función de coste	247
Ilustración 86: Ejemplo de creación de la función heurística.....	247
Ilustración 87: Ejemplo de creación de la función de asignación de padre.....	247
Ilustración 88: Ejemplo de creación del comparador de igualdad entre estados.....	248
Ilustración 89: Ejemplo de creación del algoritmo de búsqueda.....	248
Ilustración 90: Ejemplo de ejecución del algoritmo de búsqueda.....	248
Ilustración 91: Ejemplo de extracción del plan	249



Índice de tablas

Tabla 1: Planes de ejecución intuitivos	38
Tabla 2: Tipos de representación de operaciones.....	84
Tabla 3: Correspondencias entre subsistemas y espacios de nombres	122
Tabla 4: Valores del enumerado <code>Aran.Representation.OperationType</code> ...	124
Tabla 5: Correspondencias entre espacios de nombres y ensamblados.....	155
Tabla 6: Resultados de traducción para los problemas 1 al 5.....	170
Tabla 7: Resultados de traducción para los problemas 6 al 10.....	170
Tabla 8: Resultados de búsqueda para el problema 1.....	171
Tabla 9: Resultados de búsqueda para el problema 2.....	171
Tabla 10: Resultados de búsqueda para el problema 3.....	172
Tabla 11: Resultados de búsqueda para el problema 4.....	172
Tabla 12: Resultados de búsqueda para el problema 5.....	173
Tabla 13: Resultados de búsqueda para el problema 6.....	173
Tabla 14: Resultados de búsqueda para el problema 7.....	174
Tabla 15: Resultados de búsqueda para el problema 8.....	174
Tabla 16: Resultados de búsqueda para el problema 9.....	175
Tabla 17: Resultados de búsqueda para el problema 10.....	175
Tabla 18: Correspondencia entre las opciones del comando y la biblioteca.....	241

CAPÍTULO 1

INTRODUCCIÓN

Saber y saberlo demostrar es valer dos veces



1 Introducción

Inteligencia Artificial práctica. Así denominan muchos a la vertiente de la Inteligencia Artificial (IA) que se ocupa de resolver casos prácticos reales en contraposición a los resueltos por los sistemas desarrollados en entornos de laboratorio, típicamente limitados con el fin de simplificar la cantidad de elementos que influyen en los resultados y permitir así al investigador centrarse en unos objetivos concretos que rara vez se encuentran aislados en el mundo real.

La IA práctica viene a desmentir la falsa creencia popular acerca de esta disciplina. Y es que lejos de suponer la construcción de cerebros artificiales encargados de tomar las decisiones de robots humanoides propios de la ciencia-ficción, la IA práctica resuelve problemas reales y cotidianos, como el control de los limpiaparabrisas automáticos de un vehículo, la detección de pagos mediante tarjeta de crédito que se corresponden con casos de robo, la clasificación de correo electrónico como no deseado o la búsqueda de secuencias de acciones que permitan alcanzar un fin, como sucede en los dispositivos GPS que tanta popularidad han adquirido de un tiempo a esta parte.

Los casos expuestos pretenden ejemplificar que la IA es tan aplicable como cualquier otra rama de la Ingeniería en Informática. Es, sin embargo, sorprendente ver cómo aún hoy es una disciplina rara vez empleada en la resolución de problemas hasta el punto de que sólo su nombre consigue, siempre por desconocimiento, bien elevar las expectativas de su utilización más allá de lo razonable, bien ser ignorada bajo acusación de exceso de pretensiones.

Si ha habido un área que ha sabido entender la importancia y aplicabilidad de la IA ésta ha sido la **construcción de videojuegos** [Laird & van Lent, 2001], donde desarrolladores y usuarios coinciden en calificar la existencia de ese concepto como imprescindible. Lo que muchos desconocen es que lo que se ha venido empleando hasta ahora no es más que la punta del iceberg. Las limitaciones tecnológicas por un lado, el desconocimiento de los propios desarrolladores por otro y, por último, la poca apreciación del usuario de algo invisible e inaudible han retrasado el verdadero desarrollo de la disciplina hasta nuestros días. Pero el techo audiovisual está próximo a ser alcanzado y, tras cuatro décadas, a los desarrolladores cada vez les cuesta más sorprender a un usuario más y más exigente, ávido de nuevo contenido.



Precisamente esa falta de ideas, unida a la inherente curiosidad del ser humano por emular todo lo que percibe, ha hecho surgir en los últimos años una importante comunidad de **desarrolladores no profesionales**, unas veces impulsados por las propias compañías de videojuegos y otras por el mero hecho de querer imitar a aquellos a los que, en cierto modo, idolatran o envidian. Pero pocos de los que colaboran en esa comunidad tienen los conocimientos técnicos apropiados para el desarrollo de sistemas informáticos complejos, como los de IA, lo cual lastra el desarrollo de obras que, comúnmente, cumplen una labor social derivada de su difusión pública gratuita.

En octubre de 2005 el videojuego **F.E.A.R.** (Monolith Productions, 2005) se convertía en estandarte de la IA práctica erigiéndose como uno de los mejores títulos del año, siendo una de sus aportaciones más valoradas su avanzado sistema de IA. La Game Developers Conference¹ de 2006 sería testigo meses más tarde de la explicación de tan afamado sistema. Un planificador en tiempo real ingeniosamente aderezado con “*humo y espejos*” [Orkin, 2006] sorprendía a propios y extraños por su relativa sencillez hasta el punto de provocar la aparición de titulares tales como “*Desmitificada la IA de F.E.A.R.*” [GameSpy, 2006]. Ineficientes implementaciones de la misma idea, como la llevada a cabo por el equipo de desarrollo del proyecto Delta3D [McDowell *et al.*, 2006], evidenciaban poco más tarde que, por sencilla que se le haga parecer, la IA es una disciplina que requiere conocimientos técnicos avanzados.

Inspirado en el ejemplo de F.E.A.R., el presente proyecto pretende colaborar a la expansión de la IA práctica atacando uno de sus frentes de batalla: la deficiencia de herramientas de desarrollo. Sin ellas, un desarrollador tiende a rehusar la IA cuando se encuentra con que debe partir de cero en un área en la que no se siente a gusto. **El objetivo final de este trabajo es la construcción de un sistema de planificación automática** (ver “Noción básica”, página 36) con una tecnología, .NET, en la que parecen no existir precedentes. Se pondrá especial énfasis en su aplicación a videojuegos no profesionales desarrollados **para la videoconsola Xbox 360** pero se aprovechará la compatibilidad de la tecnología empleada para permitir también su uso en videojuegos y aplicaciones comunes de PC, sin que ello impida su ejecución en otras plataformas compatibles. El proyecto, de vocación internacional, será publicado en Internet **como software libre**.

¹ Más información en <http://www.gdconf.com>

CAPÍTULO 2
ESTADO DE LA CUESTIÓN

*Vale más saber alguna cosa de todo,
que saberlo todo de una sola cosa*



2 Estado de la cuestión

En este apartado se introducirán los conceptos básicos necesarios para entender el proceso de construcción del sistema objetivo desde diferentes puntos de vista, como puedan ser el histórico, el tecnológico o el técnico.

2.1 Videojuegos

El papel de la Inteligencia Artificial en los videojuegos actuales y futuros no puede entenderse sin conocer la historia y la evolución de esta industria hasta nuestros días. En este apartado se hará un recorrido global desde los inicios de la misma hasta la actualidad, pasando por las tendencias futuras. Llegado al presente, se introducirá la videoconsola actual sobre la que se ejecutará el producto de este proyecto, la Xbox 360, exponiendo las posibilidades que ésta ofrece para el desarrollo no profesional, en las cuales se fundamenta la construcción de esta idea.

2.1.1 Historia y evolución

Mucho han cambiado los tiempos y, con ellos, el concepto del entretenimiento desde 1889. En aquél entonces nada podía hacer pensar al japonés Fusajiro Yamauchi que estaba fundando la que sería una de las mayores compañías de videojuegos del mundo: Nintendo. Los naipes japoneses a cuya fabricación se dedicaría la entonces denominada Marukufu Company para el mercado nacional [DeMaria & Wilson, 2002] nada tienen que ver con los productos desarrollados por la gran multinacional que es en la actualidad, segunda empresa por cotización bursátil de Japón, sólo por detrás de la marca automovilística Toyota [Reuters, 2007a]. De los naipes a los videojuegos, Nintendo ha ido conservando su idea original, el qué, entretener a sus clientes. El resto, el cómo, ha evolucionado con el paso de los años de modo que, habiendo llegado a ser poco menos que sinónimo de videojuego, ha necesitado reconducir sus ideas en pro de una mayor aceptación por parte de un mercado saturado. Hoy tanto su videoconsola de sobremesa, Wii (Nintendo, 2006), como la portátil, Nintendo DS (Nintendo, 2004), lideran sus respectivos mercados en número de unidades vendidas, avalando el giro que la compañía ha dado hacia una nueva visión del negocio que se desmarca de las tendencias clásicas predominantes aún en la actualidad.

Pero para poder hablar de las tendencias actuales es necesario remontarse de nuevo a los **orígenes**. Y es que, por curioso que parezca, la filosofía predominante en la

actualidad no difiere tanto de la iniciada en 1961 por Steve Russell en el MIT¹, al crear Spacewar, considerado el primer videojuego de la historia. Entonces, como ahora, el juego perseguía llevar al límite las capacidades de una nueva máquina, la PSP-1, usando para ello un programa que consiguiera la implicación del espectador de forma activa y entretenida [DeMaria & Wilson, 2002]. El tiempo ha pasado y el aspecto del rudimentario Spacewar nada tiene que ver con la belleza de los juegos más punteros de la actualidad, como Assassin's Creed (Ubisoft, 2007).

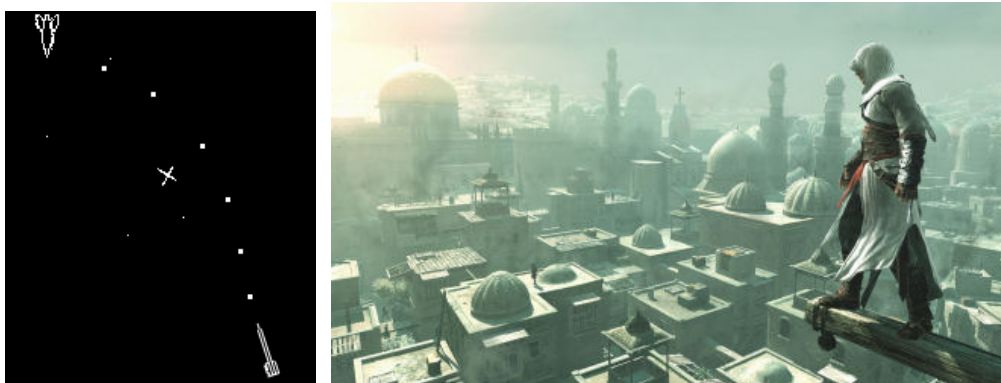


Ilustración 1: Videojuegos Spacewar y Assassin's Creed

La evidencia técnica que supone la Ilustración 1 oculta otra invisible: casi medio siglo después, la filosofía que rige uno y otro sigue siendo la misma. Ahora bien, **aprovechar al máximo las capacidades de una máquina** de última generación de 1972 como la Odyssey de Magnavox no permitía siquiera imaginar lo que en la actualidad podría hacerse con otra máquina puntera como la Xbox 360 (Microsoft, 2005), la PlayStation 3 (Sony, 2006) o un PC de gama alta. Y es que dos máquinas empleadas para dar vida a una misma filosofía, incluso con un aspecto exterior que sigue guardando similitudes, difieren sobremanera en las necesidades que crean a las empresas dedicadas al desarrollo. Atrás quedaron los años iniciales de esta aún prometedora industria, en los que una sola persona, o un grupo muy reducido de ellas, podía producir un título con no demasiados recursos. Hoy se mueven sumas de dinero que pueden hacer temblar los cimientos de la empresa más sólida si la inversión no produce los resultados esperados. Como puede suponerse, esto también sucede a la inversa. Por poner un ejemplo positivo, Halo 3 (Bungie Studios, 2007), considerado uno de los videojuegos más mediáticos del momento, supuso una recaudación de 300 millones de dólares para su editora, Microsoft, en su primera semana a la venta

¹ Instituto Tecnológico de Massachusetts, EE.UU.



[Microsoft, 2007] consiguiendo, por sí sólo, sacar definitivamente a la división de videojuegos de la compañía de los números rojos que lastraba desde sus inicios.



Ilustración 2: Videoconsolas Odyssey y Xbox 360

Y es que hoy la industria de los videojuegos se parece más que nunca a la del cine. Tanto es así que por muchos es ya considerado **el 8º arte**. Los videojuegos actuales comparten con el cine numerosas características como presupuestos de decenas de millones de dólares, equipos multidisciplinares de centenares de personas y problemas similares que resolver, como guiones, escenografía, dirección artística (puede haber incluso más artistas que programadores [Sweeney, 2006]), doblaje, etc. Tal es ya el parecido que algunos directores de cine trabajan en la industria [Microsoft, 2006a] o bien ruedan películas basadas en los personajes más populares del momento, como Hitman (20th Century Fox, 2007). Y es que los videojuegos **hace tiempo que superaron al cine como negocio**¹. Pero las actuales producciones del mundo de los videojuegos no se parecen a las del cine sólo en los términos señalados sino, también, en el **estancamiento que se está produciendo en cuanto a ideas**. Más dinero significa también más riesgo, por lo que las productoras dudan bastante a la hora de apostar por introducir novedades en el sector.

Como todo negocio de masas [Reuters, 2007b], los videojuegos lo son gracias a una gran base de clientes. Desde esos años 70 hasta la actualidad ha tenido lugar un largo proceso de **aceptación social** que está próximo a culminar. Aunque aún hoy mucha gente sigue pensando que los videojuegos son cosas de niños, lo cierto es que la media del jugador nacional es de 22 años [aDeSe, 2006], inferior aún a la de otros países como EE.UU., y no es raro encontrar a gente de 30, 40 y 50 años jugando. Eso sí, las preferencias de estos jugadores no son las mismas que los de corta edad. Su

¹ En 2007, en España vendieron más que la industria cinematográfica y la musical juntas [aDeSe, 2008]

experiencia les hace ser **más exigentes** y son, al tiempo y por contradictorio que parezca, los que más notan la **falta de novedades** pero los más críticos con éstas si se salen demasiado de las pautas habituales. Este sector poblacional representa la vertiente aún predominante, heredera de la filosofía de 1961 en plataformas como el PC y las videoconsolas Xbox 360, PlayStation 3 y PlayStation Portable (Sony, 2004).

Por otro lado, el afán de **expansión de mercado** ha hecho que en los últimos años surjan videojuegos basados en nuevos conceptos de diversión simple y directa, muchos de ellos con un marcado **carácter social**. A esta visión sólo pertenecen plataformas de Nintendo como Wii y Nintendo DS. En cuestión de software, si bien sigue siendo Nintendo la dominante, otras plataformas también tienen algunos ejemplos que son referentes en su género o, incluso, creadores del mismo. Juegos como Buzz! (Sony, 2005), simulador de concurso de preguntas, Guitar Hero (Activision, 2006) o Jam Sessions (Ubisoft, 2007), simuladores de guitarras, Donkey Kong Jungle Beat (Nintendo, 2005), controlado con unos bongos, o LittleBigPlanet (Media Molecule, 2008¹), en el que prima la creatividad del usuario a la hora de hacer sus propias pantallas y compartirlas con el resto, son claros ejemplos de una filosofía destinada a extender los videojuegos a sectores sociales antaño apartados de todo esto, como las mujeres o las personas mayores. La aceptación social de los videojuegos es tal que la política no es ajena al fenómeno [La Vanguardia, 2007].



Ilustración 3: Políticos haciendo uso de los videojuegos

Los videojuegos han supuesto siempre un campo de entrenamiento muy interesante para los investigadores de IA [Muñoz-Avila & Fisher, 2004] pero en el ámbito comercial ésta ha sido hasta ahora una disciplina secundaria, siempre ensombrecida por la parte audiovisual y sólo tenida en verdadera consideración por algunas producciones innovadoras. Ahora que esa parte audiovisual ha tocado techo por

¹ Año de publicación estimado



un lado o no tiene la importancia de antaño por otro, el sector se propone apostar definitivamente por la IA como nuevo elemento distintivo entre un videojuego del montón y un superventas.

Los dos planteamientos actuales de los videojuegos requieren IA para poder avanzar. Por ejemplo, la reciente y minoritaria, la de la diversión sin complicaciones, necesita la IA como aliado en su lucha por la simplificación de los controles, pilar base sobre el que se sustenta esta corriente. Y es que detrás de los equipos de fútbol controlados sólo con órdenes generales del entrenador de PC Fútbol (Gaelco Multimedia, 2007), las ciudades autogestionadas de Civilization IV (Firaxis Games, 2005) o los controles basados en la detección de movimientos bidimensionales de Nintendo DS o tridimensionales de Wii y PlayStation 3 está algún tipo de IA. Pueden verse ya, de hecho, **empresas dedicadas en exclusiva a ofrecer productos de IA orientada a videojuegos**, como AiLive¹ [Gamasutra, 2007b] o Xaitment², algo impensable hace pocos años.

En la otra corriente, predominante y única hasta hace poco, la de la explotación al máximo de las máquinas, la industria ha hecho ya grandes **apuestas por la IA que han conseguido llevar a las producciones que lo hicieron a lo más alto**. Un ejemplo claro de ello es el videojuego Los Sims (Maxis, 2000), el más vendido de la historia. Otro claro ejemplo es el videojuego Black & White (Lionhead Studios, 2001), considerado en su momento por el libro Guinness de los Records como el poseedor de la IA más compleja. Otro ejemplo más reciente es el videojuego F.E.A.R. (Monolith Productions, 2005), cuya IA fue alabada por crítica y usuarios y considerada un avance significativo con respecto a la mostrada por otros juegos del mismo género.

Hoy día, en todos estos juegos de todas las corrientes se busca un añadido que les diferencie del resto, que aporte algo que los demás no tienen, y ese algo es, cada vez más, IA. La IA permite, por ejemplo, dotar a los personajes de ese componente humano, emocional, que aún les falta, dar un paso más hacia la superación de su particular test de Turing [Turing, 1950] y, al tiempo, permite **simplificar los procesos de desarrollo** [Orkin, 2006].

¹ Más información en <http://www.ailive.net>

² Más información en <http://www.x-aitment.net>



2.1.2 Desarrollo no profesional en Xbox 360

Si hay algo que nadie niega a Microsoft es la capacidad que ha tenido siempre para atraer a los desarrolladores de videojuegos hacia sus plataformas. Es, de hecho, uno de los pocos motivos reconocidos ampliamente por los detractores del sistema operativo Windows como un punto a favor en estos momentos sobre otros como GNU/Linux. Gran parte de ese éxito se debe a **DirectX**, una biblioteca de desarrollo de videojuegos que la compañía de Redmon lanzó en 1995 con el fin de facilitar la programación de este tipo de aplicaciones en su entonces nuevo Windows 95. Aunque existen alternativas libres a DirectX, lo cierto es que ésta supone una biblioteca común para el manejo de todo lo relacionado con el desarrollo de videojuegos, como gráficos, sonido, entrada/salida, comunicaciones en red, etc. Mientras, su contrapartida libre exige la combinación de varias soluciones [Barrapunto, 2007] como, por ejemplo, OpenGL¹ para gráficos y OpenAL² para sonido, con la desventaja de estar muchas veces poco integradas entre sí y/u orientadas a desarrolladores con más experiencia pero con la ventaja, eso sí, de ser multisistema. Ya sea por esta u otras razones, lo cierto es que DirectX es hoy día el **estándar empleado por las desarrolladoras de videojuegos para PC**, siendo muy pocas las que emplean sus alternativas e, incluso, anunciando el cese del soporte comercial a las mismas en proyectos futuros, como el reciente caso de id Software [Carmack, 2007].

No contenta con su dominio en el mundo del PC, y preocupada porque el éxito de la PlayStation (Sony, 1994) le quitase cuota de mercado a éste, Microsoft se introdujo en el mundo de las videoconsolas en noviembre de 2001, en plena sexta generación, intentando así diversificar su negocio, basado hasta entonces sólo en *software*. Nació **Xbox**, un sistema de videojuegos basado en una arquitectura similar a la de un PC y **cuyo modelo de desarrollo se basaba en DirectX**, facilitando la producción de videojuegos para ambas plataformas. De hecho, originalmente pretendió llamársele DirectX-box y, aunque su nombre fue embellecido por razones de *marketing*, conservó la X tanto en el nombre como en la apariencia³.

¹ Más información en <http://www.opengl.org>

² Más información en <http://www.openal.org>

³ Información extraída de <http://en.wikipedia.org/wiki/Xbox>



Ilustración 4: Videoconsola Xbox y logotipo de DirectX

Aún siendo la videoconsola más potente de su generación y la primera en disponer de un servicio de comunicación en red de banda ancha (Xbox LIVE¹), la Xbox se vio ensombrecida por el poder mediático de la PlayStation 2 (Sony, 2000) por lo que Microsoft decidió adelantarse a su rival lanzando a la sucesora de su primer sistema, la **Xbox 360**², en noviembre de 2005. La nueva videoconsola **sigue basando su modelo de desarrollo en DirectX**, empleando esta vez un *hardware* bien distinto al de un PC pero cuya arquitectura es similar. La Xbox 360 dispone de un procesador con 3 núcleos simétricos de propósito general, cada uno a 3,2 Ghz y con soporte para 2 *threads hardware*, sumando un total de 6, a los que hay que añadir una tarjeta gráfica de última generación que comparte los 512 MB de RAM³ con el procesador. Estas características sitúan a la videoconsola en un empate técnico con su nueva rival tecnológica, la PlayStation 3 (Sony, 2006) [The Inquirer, 2007], que emplea una arquitectura muy diferente y difícil de programar, a lo que no ayudan sus herramientas de desarrollo [Gamasutra, 2007a]. Ambas videoconsolas presentan un ritmo mundial de ventas similar, si bien el año que las separa hace que la Xbox 360 tenga más cuota de un mercado dominado por la Wii⁴, tecnológicamente inferior a ellas.

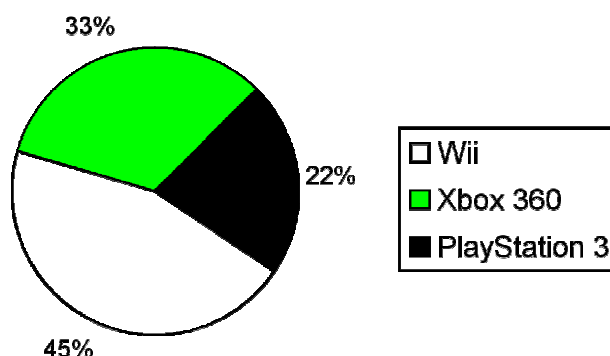


Ilustración 5: Cuotas de mercado en la 7ª generación de videoconsolas

¹ Más información en <http://www.xbox.com/es-es/live>

² Fotografía en página 15, Ilustración 2

³ Suficientes para estar al nivel de un PC de gama alta, con 1-2 GB de RAM

⁴ Información extraída de <http://www.vgchartz.com> a 11 de mayo de 2008

El mismo año en el que su primera videoconsola iniciaba la batalla en el mundo del *hardware*, Microsoft contraatacaba en el mundo que domina, el *software*, a la plataforma Java de Sun Microsystems. 2001 sería también el año del **nacimiento de la tecnología .NET**.

.NET es un proyecto de Microsoft para crear una nueva plataforma de desarrollo de *software* con énfasis en la transparencia de redes, con independencia de plataforma y que permita un rápido desarrollo de aplicaciones. Basado en esta plataforma, Microsoft intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el sistema operativo hasta las herramientas de mercado¹.

Java y .NET comparten una concepción similar y un mismo modelo de desarrollo, basándose en la independencia de plataforma mediante la **abstracción del hardware subyacente por medio de una máquina virtual** (comúnmente implementada en *software*) a pila, orientada a objetos y con distintos servicios comunes a toda aplicación, como la gestión automática de memoria, que permite al desarrollador olvidarse de los aspectos más rutinarios y centrarse en la materialización de sus ideas.



Ilustración 6: Arquitectura de desarrollo para Xbox 360

El modelo de abstracción antes comentado les permite trabajar en arquitecturas bien diferentes basándose en una **doble compilación**. Los ficheros generados con sus compiladores contienen instrucciones para la **máquina virtual** (que en .NET recibe el nombre de **CLR**²) cuyo compilador interno, denominado JIT³, vuelve a compilar en tiempo de ejecución dando como resultado el código máquina del *hardware* subyacente.

¹ Definición extraída de <http://es.wikipedia.org/wiki/.NET>

² Siglas de “Common Language Runtime”

³ Siglas de “Just In Time”



Siendo la máquina virtual el componente fundamental sobre el que se sustenta este modelo de programación, la otra piedra angular del mismo es la **biblioteca estándar de tipos**. Tanto Java como .NET disponen de una serie de tipos preprogramados y pensados para ser reutilizados, de modo que ahorren recursos de desarrollo al evitar “reinventar la rueda”. La máquina virtual y la biblioteca estándar (que en .NET se denomina **BCL**¹) conforman, junto con otros elementos que no interesa aquí describir, el **entorno de ejecución** (que en .NET se denomina **.NET Framework**). Ese entorno de ejecución suele tener distintas configuraciones según el dispositivo que lo implemente. En ese sentido, en .NET existe una versión ligera (el **.NET Compact Framework**) que, por ejemplo, no admite todo el conjunto de instrucciones de la máquina virtual completa ni todo el conjunto de tipos o miembros de tipo de la biblioteca estándar.

Acabando ya con las comparaciones, .NET, al contrario que Java, es una plataforma ideada para ser empleada por diferentes compiladores de **diferentes lenguajes**, creados o no por Microsoft². No es que Java no pueda soportar idéntico uso³, sino que en .NET, atendiendo a las diferencias existentes entre lenguajes, se han cuidado de especificar un conjunto de **normas que se recomiendan usar para asegurar la interoperatividad entre lenguajes**. A ese conjunto de normas se le denomina **CLS**⁴ y permiten, por ejemplo, que lenguajes como **VB.NET** (que no distingue mayúsculas) puedan utilizar sin ambigüedades tipos nombrados desde **C#** (que sí las distingue). Estos dos lenguajes son los más utilizados de la plataforma y el primero de ellos requiere de unas bibliotecas que extienden la BCL y que no están presentes en todos los entornos de ejecución.

Microsoft ha puesto mucho énfasis desde que lanzara .NET en que ésta fuera una **plataforma estándar**, por lo que la especificación de su infraestructura y lenguajes principales está recogida en documentos de organizaciones internacionales de estandarización como **ECMA** o **ISO** [MSDN, 2006a]. Ello permite que existan implementaciones distintas a la de Microsoft, como los proyectos Mono⁵ y DotGNU⁶,

¹ Siglas de “Base Class Library”

² En <http://www.dotnetpowered.com/languages.aspx> hay una extensa lista de lenguajes de .NET

³ El lenguaje Groovy es un ejemplo de ello, <http://groovy.codehaus.org>

⁴ Siglas de “Common Language Specification”

⁵ Más información en <http://www.mono-project.com>

⁶ Más información en <http://www.gnu.org/software/dotgnu>



que permiten la ejecución de aplicaciones .NET en sistemas operativos distintos de los de esta compañía, como GNU/Linux, Solaris, Mac OS X y BSD (OpenBSD, FreeBSD y NetBSD). Por otro lado, la plataforma hace mucho énfasis en el intercambio de información basada en el estándar actual que supone el lenguaje XML¹.

Tras el lanzamiento de .NET, en 2002 Microsoft creó una serie de bibliotecas para la plataforma que eran una correspondencia casi directa con DirectX a las que llamó **Managed² DirectX (MDX)**. MDX es, en realidad, un **enlace transparente al desarrollador entre las aplicaciones .NET y DirectX** y depende de ésta última puesto que, en último término, es quien ejecuta realmente las funciones, pero esto poco o nada debe importarle al desarrollador.

En diciembre de 2006 nació XNA³ [MSDN, 2006b]. **XNA es la evolución de MDX** hasta el punto de anunciarse el cese del soporte a la segunda desde el lanzamiento de la primera. Como evolución que es, XNA sigue dependiendo de DirectX pero su diseño abstrae aún más al desarrollador, siendo más sencillo y empleando en mayor medida los principios de la programación orientada a objetos. Otra diferencia con **DirectX** es que mientras éste **es una biblioteca, XNA es un marco de trabajo** y, como tal, ofrece una serie de tipos destinados a ser ejecutados dentro de un entorno definido. Por ejemplo, existe un tipo que representa las funciones básicas de un videojuego de modo que el usuario sólo deberá especificar el código que dibuja la pantalla cada cierto tiempo, olvidándose por completo de aspectos como llamar a ese método en cada intervalo temporal o inicializar el dispositivo gráfico. XNA lo hace por él.



Ilustración 7: Logotipo de XNA

¹ Más información en <http://www.w3.org/XML>

² Microsoft se refiere al código gestionado por el .NET Framework como *managed* para diferenciarlo del aquél que no lo es

³ Siglas de “XNA’s Not Acronymed” (es una definición recursiva)



Con el nacimiento de XNA, Microsoft puso a disposición de los usuarios de Xbox 360 una descarga de contenido, el **XNA Game Launcher**, mediante su sistema de distribución digital Xbox LIVE. El XNA Game Launcher dotaba a su máquina de una implementación del .NET Compact Framework, extendiendo la BCL sólo con las bibliotecas de XNA (el XNA Framework) dejando, de hecho, **C# como única vía de programación**. Realmente serviría cualquier otro lenguaje que sólo dependa de la BCL, como F#¹, pero no tienen soporte oficial para XNA por parte de Microsoft. El XNA Game Launcher incluía, además, funcionalidades destinadas a facilitar la conexión de la videoconsola al PC, porque era, y sigue siendo, la única vía de transferir juegos XNA a la primera. Por su parte, el PC recibió el **XNA Game Studio Express**, que reúne las bibliotecas de XNA para PC y Xbox 360, herramientas de gestión de contenido (el XNA Content Pipeline), ejemplos y tutoriales e integración con una versión gratuita del IDE² Visual Studio 2005 recortado para su uso sólo con C#, el **Visual C# 2005 Express**. Con XNA Game Studio Express se pueden desarrollar juegos basados en XNA tanto para PC como para Xbox 360, empleando la mencionada conexión para transferirlos a esta última ya sea para instalarlos o para depurarlos. La descarga y uso del XNA Game Launcher y, por tanto, el despliegue de videojuegos en Xbox 360, requiere de una suscripción al denominado **XNA Creators Club**³, a razón de 50 € por cada 4 meses o 100 por cada año⁴, a renovar en cada cumplimiento de periodo y dotando de acceso a servicios exclusivos para sus miembros.

En marzo de 2007, Microsoft lanzó una actualización de XNA 1.0 denominada **XNA 1.0 Refreshed** [MSDN, 2007a], que incluía algunas mejoras en distintos apartados, entre ellos la posibilidad de hacer paquetes de instalación que facilitaban el despliegue de videojuegos desde el PC a la Xbox 360. En diciembre de 2007, apenas un año después de su primera versión, se lanzó **XNA 2.0** [MSDN, 2007b], con mejoras como la posibilidad de desarrollar juegos en red vía Xbox LIVE. Ambas actualizaciones tuvieron repercusión también en Xbox 360 por medio de sendos nuevos contenidos. La primera de ellas se limitó a la actualización del XNA Game Launcher, pero la segunda supuso un nuevo nombre para el mismo concepto. El renombrado **XNA Game Studio Connect** ofrece desde entonces, sobre los mismos principios que el Game Launcher, el

¹ Más información en <http://research.microsoft.com/fsharp>

² Siglas de “Integrated Development Environment”

³ Más información en <http://creators.xna.com>

⁴ Precios en el momento de redactar este documento

soporte a XNA 2.0 y una mejor integración con la videoconsola. XNA 1.0 (Refreshed) y 2.0 pueden convivir tanto en PC (Game Studio 1.0 y 2.0) como en Xbox 360 (Game Launcher y Game Studio Connect). En el primer caso se debe a que así se mantiene la posibilidad de desarrollar para ambas versiones, en el segundo es debido a que el XNA Game Studio Connect no es capaz de ejecutar juegos de XNA 1.0: si detecta un juego de esa versión, se descargará automáticamente el Game Launcher. Durante la Games Developers Conference de 2008, Microsoft anunció el lanzamiento de XNA Game Studio 3.0 para finales de ese mismo año [MSDN, 2008a].

Se ha insistido constantemente en que XNA está destinado al desarrollo no profesional, pero en verdad la idea de Microsoft es que lo empleen también **profesionales independientes**¹. Por otro lado, la meta de Microsoft a largo plazo con XNA era crear “*el YouTube de los videojuegos*” [BBC, 2006], algo que confirmó en la Games Developers Conference de 2008 anunciando para finales de ese año el lanzamiento del servicio **Xbox LIVE Community Games** [MSDN, 2008b]. Dicho servicio permitirá a los miembros del XNA Creators Club poner sus creaciones a disposición de todas las Xbox 360 del mundo vía Xbox LIVE y cobrar por su descarga si así lo desean. Un sistema *meritocrático*² en el que participarán los miembros del XNA Creators Club será el medio de publicación. Aunque el proyecto entorno a XNA no es, ni mucho menos, la primera iniciativa similar que ha surgido en la historia de los videojuegos³, sus características, el estado actual de la tecnología y, claro, el impulso de su creadora, Microsoft, hacen pensar que será la primera con verdadera repercusión en el mercado.



Ilustración 8: Pasos a seguir en Xbox LIVE Community Games

¹ El juego comercial Schizoid está hecho con XNA, <http://www.torpexgames.com/games.php>

² Más información en <http://es.wikipedia.org/wiki/Meritocracia>

³ Le preceden, entre otros, el Net Yaroze de PlayStation, http://en.wikipedia.org/wiki/Net_Yaroze



La estrategia de Microsoft abarca varios frentes, incluido el universitario. Y es que la Games Developers Conference de 2008 también sirvió para desvelar el programa DreamSpark¹, por el cual los estudiantes universitarios pueden descargarse gratuitamente las herramientas de desarrollo de Microsoft y usarlas con fines académicos. El mismo programa pone a disposición de los estudiantes **subscripciones gratuitas al XNA Creators Club** de un año de duración. Desde su lanzamiento, XNA **ha sido adoptado por 400 universidades** de todo el mundo [GamesIndustry, 2008].

Otro frente que Microsoft no olvida es el **software libre**, por medio de **CodePlex** [Microsoft, 2006b], un sitio para albergar proyectos de esta naturaleza. Convertido rápidamente en un referente, alberga varios proyectos de Inteligencia Artificial (IA) y videojuegos, destacando dos de ellos. Por un lado, **SharpSteer**², una biblioteca de comportamientos de conducción automática de elementos móviles (ver “Técnicas comunes”, página 27, punto 4), compatible con XNA. Por otro, **JadEngine**³, un motor de videojuegos basado en MDX, sin intención de ser migrado a XNA [JadeEngine, 2007], con un importante bloque de IA que da soporte a búsqueda de caminos (ver “Técnicas comunes”, página 27, punto 3), máquinas de estado finitas (ver “Técnicas comunes”, página 28, punto 6), comportamientos de conducción (mejor diseñados que los de SharpSteer, desde el punto de vista de la POO⁴) y algoritmos genéticos (ver “Técnicas prometedoras”, página 31, punto 2). **No se han encontrado proyectos para XNA destinados a la planificación automática** independiente de dominio y se duda que existan para la plataforma .NET, de ahí el interés de este proyecto.

2.2 Inteligencia Artificial en videojuegos

El objetivo que persigue la **Inteligencia Artificial** (IA) es la aplicación de técnicas automáticas que permitan resolver problemas para los cuales sería preciso el razonamiento humano. La disciplina que lleva ese nombre se viene centrando históricamente en la elaboración de sistemas complejos que tienen en común un fuerte componente algorítmico y, normalmente, un elevado coste computacional. A pesar de todo, y los videojuegos no son una excepción, en la simulación del razonamiento **suelen participar también técnicas que, formalmente, pertenecen a otras áreas de la**

¹ Más información en <https://downloads.channel8.msdn.com>

² Más información en <http://www.codeplex.com/SharpSteer>

³ Más información en <http://www.jadengine.com>

⁴ Siglas de “Programación Orientada a Objetos”



Ingeniería Informática. En este apartado se mencionarán algunas de las técnicas de uso actual en videojuegos clasificadas, según [Rabin *et al.*, 2004], en comunes y prometedoras.

2.2.1 Técnicas comunes

A continuación se exponen una serie de técnicas bastante extendidas dentro de la simulación de comportamientos inteligentes en videojuegos según [Rabin *et al.*, 2004]. Para cada una de ellas se expondrá una breve descripción y, si se considera conveniente, algún ejemplo relacionado.

1. **Análisis de terreno.** Consiste en inspeccionar un terreno para determinar los puntos estratégicos, como recursos, cuellos de botella, puntos de emboscada, etc. En juegos de estrategia permite determinar dónde situar determinados elementos. En juegos de disparos, se usa para descubrir puntos de francotirador, para cubrirse o desde dónde lanzar granadas. La técnica permite que los agentes puedan así desenvolverse en entornos desconocidos, y no sólo los preprogramados.

Los **mapas de influencia** son un método de análisis de terreno que permiten ver la distribución de poder entre las fuerzas presentes en un mapa. Normalmente es una rejilla bidimensional que se superpone al mapa y en el que las unidades que caen en cada celda son combinadas en un único número que representa la influencia de todas ellas. Cada unidad tiene, además, una influencia sobre las celdas adyacentes, menor cuanto más distante es la celda. Además de servir para el análisis de terreno, puede emplearse en el cálculo de la diferencia de poder entre dos facciones, mediante la resta de las fuerzas representadas en sus mapas de influencia. Un ejemplo de uso no bélico es Sim City (Maxis, 1989), en el que se muestra la influencia de la policía, los bomberos o el tráfico para ayudar al usuario a decidir y al juego a simular el mundo.

2. **Arquitectura de subsunción.** Esta arquitectura, proveniente de la robótica, estructura el comportamiento de un agente en máquinas de estado concurrentes y separadas en capas. Las capas bajas se ocupan de acciones rudimentarias, como la evasión de obstáculos, y las altas se ocupan de tareas de mayor nivel, como el descubrimiento o la selección de objetivos. La mayor prioridad de las capas bajas asegura que sólo si se cubren las necesidades básicas se consumirán recursos en otras tareas. En robótica, esta arquitectura hace tiempo que tiende a ser sustituida



por sistemas de planificación combinados con control reactivo. Es lógico pensar que el futuro de los “robots *software*” dedicados a videojuegos sea el mismo.

3. **Búsqueda de caminos** (*pathfinding*). Consiste en la determinación del camino a seguir para ir de un punto a otro de un entorno dado. Una vez representado el entorno convenientemente, y decididos los puntos de origen y destino, un algoritmo de búsqueda devuelve una lista de puntos por los que pasar, como si de un rastro de migas de pan se tratara, una guía. El elemento móvil en cuestión emplea esos puntos como guías para encontrar su camino hasta la meta. El algoritmo más empleado en la actualidad es el A* o algún derivado optimizado para casos especiales, como el RTA* [Korf, 1990] para tiempo real o el D* [Stentz, 1994] para entornos parcialmente desconocidos. Algunos algoritmos desarrollados en robótica, como el RRT [Bruce & Veloso, 2002], son también aplicables. Esta técnica suele combinarse con la evasión de obstáculos, explicada más adelante, para poder sortear otros elementos.
4. **Comportamiento emergente**. Un comportamiento emergente es aquél que no ha sido explícitamente programado pero surge por la interacción de reglas simples de bajo nivel.

Los **comportamientos de conducción** (*steering behaviors*) son un claro ejemplo de esta técnica. Consisten en la combinación de un determinado número de reglas simples, como seguir un objeto concreto, ir a un cierto punto, mantener la distancia con la pared, etc. que dan lugar a un comportamiento complejo que se traduce en la conducción de un elemento móvil de un punto a otro de una forma muy natural.

La **evasión de obstáculos** (*obstacle avoidance*) es una especialización de los comportamientos de conducción que suele combinarse con la búsqueda de caminos para que, mientras se sigue la guía establecida por el algoritmo de búsqueda, puedan además evitarse a otros elementos móviles para no chocar contra ellos. Para ello hace uso de la predicción de trayectorias, explicada más adelante. Con esta técnica, por ejemplo, un conjunto de unidades puede atravesar un estrecho sin chocar entre ellas.

La **conducción de manadas** (*flocking*) es otra especialización de los comportamientos de conducción. Es una técnica para mover grupos de criaturas de



forma natural, como se requiere en la simulación del comportamiento de bandadas de pájaros, bancos de peces o colonias de insectos. Cada criatura sigue 3 reglas básicas de movimiento (separación, alineamiento y cohesión) que hacen emerger un comportamiento de grupo que sólo se emplea para criaturas sin destino específico.

La conducción de manadas es también un tipo de **inteligencia colectiva** (*swarm intelligence*), que es aquella que surge de la interacción de los distintos miembros de un grupo entre sí y con el entorno. Cada miembro del grupo se comporta siguiendo una serie de reglas simples que, unido al comportamiento del resto de los miembros, genera un resultado conjunto que no se produciría por separado.

La gestión de **formaciones** es una técnica que pretende imitar los movimientos de grupos militares. Es similar la conducción de manadas, pero aquí cada unidad es guiada hacia una posición de destino específica, basada en su posición en la formación.

5. **Jerarquía de comandos.** Se trata de una técnica basada en la cadena de mando militar, en la que existe una estrategia de alto nivel, basada en la gestión de unidades colectivas como los escuadrones, y otra de bajo nivel, basada en el combate individual. Suele emplearse en juegos de estrategia. También se usa cuando se pretende dotar a varios agentes de un comportamiento común coherente, como en algunos juegos de disparos.

La **asignación de tareas** (*manager task assignment*) podría considerarse un tipo de jerarquía de comandos. Esta técnica consiste en tener una lista de tareas por hacer y una entidad superior que selecciona qué agente debe ocuparse de la tarea, basándose en sus capacidades y eligiendo al mejor de ellos. Se pretende imitar la figura del entrenador (*manager*) de un equipo deportivo de forma diferente a ir agente por agente asignándole una tarea. Aquí las tareas deciden la prioridad, por lo que se suele requerir una arquitectura de comunicación entre ellos, como una pizarra (ver “Técnicas prometedoras”, página 33, punto 6).

6. **Máquinas de estado finitas** o autómatas. Es una de las técnicas más empleadas y define un conjunto finito de estados y unas transiciones entre ellos, con un único estado activo en cada momento. Normalmente cada estado suele representar un



comportamiento y permanece a la espera o escucha eventos que provoquen la transición a otro. Por ejemplo, un agente en estado “patrullar” verificará periódicamente si ve un enemigo y entonces cambiará al estado “atacar”.

Las **máquinas de estado a pila** (*stack-based state machines* o *push-down automata*) son aquellas que, al contrario de lo que sucede con las anteriores, pueden recordar estados pasados almacenándolos en una pila de la cual pueden ser retomados. Esta técnica suele emplearse cuando un agente está realizando una acción, es interrumpido y luego quiere retomar la acción en la que se encontraba. Por ejemplo, en estrategia, una unidad puede estar reparando algo y ser atacada. Muerto el enemigo, la unidad retomará la actividad de reparación.

7. **Nivel de detalle.** El nivel de detalle es una técnica muy extendida en el campo de la generación de gráficos tridimensionales en tiempo real. Consiste en emplear texturas y modelos poligonales detallados cuando sólo cuando el ojo humano puede percibir esos detalles, esto es, cuando un objeto esté cerca de la cámara. Es un método que disminuye los requerimientos computacionales sin pérdida de calidad aparente y que trasladado a la IA consiste en realizar cálculos sólo cuando el jugador puede apreciar su resultado. Por ejemplo, variar la frecuencia de actualización de un agente dependiendo de la proximidad del jugador. Otro ejemplo es calcular rutas sólo para los personajes que el jugador puede ver, empleando una aproximación de movimientos en línea recta para aquellos no visibles.
8. **Predicción de trayectorias** (*dead reckoning*). El símil humano “cálculo a ojo”, que sería su traducción literal, define bien a una técnica utilizada en la estimación de una posición futura de un elemento móvil, basándose en parámetros actuales como su posición actual, velocidad y aceleración. Se suele emplear una aproximación en línea recta y en espacios cortos de tiempo. En juegos de disparos se emplea para predecir la posición del oponente y en los deportivos para anticipar la posición de otros jugadores y pasar así efectivamente la pelota o interceptar a otro jugador. Su nivel de acierto (dependiente, por ejemplo, de la velocidad de muestreo) suele ser un parámetro configurable para adaptar la dificultad de la IA del juego.
9. **Programación ligera** (*scripting*). Esta técnica consiste en formular la lógica del juego empleando un lenguaje diferente del usado para su núcleo (comúnmente,



C/C++) y, normalmente, más sencillo de usar, con menos capacidades (más ligero) pero orientadas sólo al tipo de programación que requiere el juego en cuestión. Algunas compañías disponen de sus propios lenguajes creados desde cero, como el Unreal Script¹ pero otras prefieren emplear otros ya existentes como Python² o Lua³, empleado en videojuegos de renombre como World of Warcraft (Blizzard Entertainment, 2003) o motores punteros como el CryEngine 2⁴ (Crytek, 2007). El perfil del usuario de *scripts* es de artista o diseñador de niveles, por lo que suele ponerse mucho énfasis en su robustez [Sweeney, 2006]. Los *scripts* pueden ser compilados estáticamente, interpretados o compilados en tiempo de ejecución.

Los **sistemas dirigidos por eventos** (*trigger systems* o *event-driven systems*) son aquellos que permiten el manejo de reglas simples del tipo “si-entonces” para agregar lógica al mundo del juego. Su sencillez y robustez lo hace útil para diseñadores de niveles por lo que, normalmente, son expuestos a través de herramientas de diseño o *scripting*. Un diseñador puede poner un disparador en el suelo de una habitación para que suene algo (efecto) cuando el jugador lo pise (condición), tal y como podía hacerse, por ejemplo, con el editor de niveles del videojuego Starcraft (Blizzard Entertainment, 1998).

Se empleará ahora el videojuego F.E.A.R. (Monolith Productions, 2005), en el cual se inspiró este proyecto, para mostrar que el uso de estas técnicas no es ni mucho menos algo aislado o desconectado. La arquitectura de agentes de F.E.A.R. implementa un sistema de jerarquía de comandos mediante asignación de tareas que permite organizar la dirección de individuos y grupos [Orkin, 2006]. También hace uso de un sistema simplificado de máquinas de estados, así como el análisis de terreno [Orkin, 2005]. El movimiento de sus agentes está dirigido por caminos calculados mediante búsqueda por A*, complementado con la evasión de obstáculos (recuérdese que el camino es sólo una guía). Como juego de disparos que es, es prácticamente obligatorio el empleo de predicción de trayectorias. Por otro lado, las herramientas de diseño emplean *scripting* para definir, por ejemplo, el comportamiento asociado a cada agente. Con todo, los pilares de su sistema de IA son una arquitectura de pizarra y un

¹ Más información en <http://unreal.epicgames.com/UnrealScript.htm>

² Más información en <http://www.python.org>

³ Más información en <http://www.lua.org>

⁴ Información extraída de <http://www.lua.org/uses.html>



planificador automático, curiosamente ambos considerados por [Rabin *et al.*, 2004] como técnicas prometedoras (ver “Técnicas prometedoras”, página 33, punto 6 y página 35, punto 11).

2.2.2 Técnicas prometedoras

Las siguientes líneas introducirán un conjunto de técnicas que, según [Rabin *et al.*, 2004], son de uso poco o nada frecuente en la simulación de comportamientos inteligentes en videojuegos pero que, sin embargo y bien aplicadas, arrojan resultados netamente superiores en cuanto a calidad de la simulación se refiere que aquellas que están más difundidas. Cada una de las técnicas será brevemente descrita y ejemplificada, esto último sólo si se considerase apropiado.

1. **Aleatoriedad filtrada.** Consiste en filtrar los resultados de un generador de números aleatorios de modo que sean eliminadas las secuencias que puedan parecer no aleatorias. Si bien esto sería algo ilegal en juegos automáticos de azar, debe tenerse en cuenta que se está aplicando a un campo bien distinto en el que debe prevalecer el entretenimiento por encima de todo y dicho entretenimiento puede verse afectado si, por ejemplo, un personaje que ejecuta secuencias animadas aleatorias repite varias veces seguidas la misma, restando naturalidad, o si en 10 lanzamientos de una moneda dan todos el mismo resultado, restando credibilidad.
2. **Algoritmos genéticos.** Un algoritmo genético es una técnica de búsqueda aleatoria basada en principios evolutivos. El algoritmo representa cada estado del espacio de búsqueda como un individuo, éste a su vez como un conjunto de cromosomas, que a su vez se descompone en un conjunto de genes. La estructura genética del individuo codifica el estado en el espacio de búsqueda y dicho espacio es la combinación de todos los posibles valores que puede tomar esa estructura. El algoritmo parte de una población de individuos que va evolucionando mediante técnicas que imitan a la naturaleza y su aleatoriedad, como la selección natural, la mutación o el cruzamiento. La búsqueda finaliza cuando se consigue un individuo que satisface una serie de condiciones preestablecidas que componen la meta [Holland, 1975]. Lo aleatorio de esta técnica hace que consiga un mejor rendimiento que otras cuando se trata de buscar soluciones próximas a la óptima (subóptimas), pero cada ejecución del algoritmo puede arrojar un resultado diferente. A pesar de ello, se trata de una técnica computacionalmente costosa, por lo que no suele emplearse en tiempo real



sino, por ejemplo, en casos de optimización como pueda ser la asignación de los mejores parámetros para un personaje del juego o la codificación de un programa que cumpla una serie de restricciones (lo que se conoce como **programación genética** [Koza, 1990]).

3. **Aprendizaje de situaciones de debilidad** (*weakness modification learning*). El objetivo de esta técnica es evitar que el juego pierda repetidamente del mismo modo ante el jugador. La idea básica consiste en grabar una secuencia del juego que preceda al fallo, de modo que cuando sea reconocida en el futuro éste pueda modificar su comportamiento. Ello no quiere decir que vaya a mejorar su rendimiento en victorias, sólo que no fallará del mismo modo. La ventaja de esto es que sólo requiere grabar una única secuencia por error. Por ejemplo, en un juego de fútbol se puede grabar el punto en el que se encontraba la pelota, su vector de movimiento y los jugadores que estaban alrededor.
4. **Aprendizaje por refuerzo**. Se trata de una técnica que permite aprender funciones de decisión a base de prueba y error de modo que cuando se consigue un resultado considerado positivo se incentiva el aprendizaje de la regla aplicada y se hace lo contrario cuando se falla. Es particularmente útil cuando los efectos de las acciones en el mundo del juego son desconocidos, pero requieren algo que aporte ese incentivo o castigo, ese refuerzo.

Las **redes de neuronas** son un tipo de sistema cuyo objetivo es aprender complejas funciones no lineales que relacionan un conjunto de variables de entradas con una serie de variables de salida. Su nombre se debe a que internamente consisten en un conjunto de elementos estructuralmente idénticos conectados entre sí de forma análoga a las neuronas del sistema nervioso. La función que las redes pueden representar está controlada por el peso que tiene cada una de las conexiones entre las neuronas. Esos pesos son modificados mediante técnicas de entrenamiento basadas en mostrar a la red relaciones de entradas y salidas de cuya relación con el resultado que ella arroje se deriva la modificación a aplicar sobre cada peso. Se trata de una técnica poco utilizada en tiempo real debido al coste computacional del entrenamiento. El videojuego Collin McRae Rally 2.0 (Codemasters, 2000) empleó esta técnica para la conducción de vehículos. En Back & White (Lionhead Studios, 2001) se empleó para el reconocimiento de gestos y el aprendizaje de reglas simples



con salidas tipo “sí/no”, como decidir si la criatura cuidada por el jugador tenía o no hambre.

5. El **aprendizaje de árboles de decisión** es una técnica consistente en la elaboración de un árbol de decisión a partir de unos datos dados. Un árbol de decisión es una serie de sentencias “si-entonces” anidadas en forma de árbol que relacionan una serie de valores de entrada con una salida. Su uso más común es la predicción. Por ejemplo, se puede tomar como entrada la vida y la munición de un agente y como salida si sobrevivirá o no a un combate. Lo que persigue esta técnica es que esas estructuras en árbol se generen sobre la marcha a partir de ejemplos, lo cual hace muy apto su uso en tiempo real. Esta técnica también se empleó en el videojuego Black & White (Lionhead Studios, 2001).
6. **Arquitectura de pizarra.** Esta arquitectura está diseñada para resolver un problema entre varios agentes, escribiendo información en un espacio compartido por todos ellos, llamado pizarra. Un uso de esta arquitectura es que los agentes, basándose en la información de la pizarra, propongan soluciones parciales al problema, a cada una de las cuales se le asigna un valor de relevancia de modo que se va aplicando cada vez la más relevante de las soluciones parciales propuestas hasta resolver el problema completo. Por ejemplo, puede emplearse en la decisión de una estrategia de ataque conjunta de modo que cada unidad proponga el rol que podría cumplir. Otro uso de esta arquitectura es realizar cualquier tipo de coordinación empleando la pizarra como punto de comunicación global. Por ejemplo, los agentes pueden indicar en la pizarra qué están haciendo para que otros no lo repitan de modo que así se puede evitar que, por poner un caso, todos ataquen al mismo objetivo.
7. **Lógica difusa** (*fuzzy logic*). Se trata de una extensión de la lógica clásica en la cual los hechos no son totalmente ciertos o falsos sino parcialmente. Por ejemplo, en la lógica clásica un animal puede estar o no hambriento mientras que en la lógica difusa puede estar hambriento con una certeza de un 10% (poco hambriento), de un 90% (muy hambriento) o cualquier otro valor porcentual entre 0 (nada) y 100 (todo). La lógica difusa emplea reglas de combinación de hechos que manejan esa incertidumbre. Por ejemplo, en la lógica clásica una condición del tipo “feliz = con dinero \wedge sin hambre” tiene sólo 2 valores (sí/no) posibles resultantes de 4 combinaciones distintas de los valores (sí/no) de “con dinero” y “sin hambre”. En



lógica difusa el valor debe ser un número real entre 0 y 100, un grado de certeza, por lo que las reglas de combinación clásicas no sirven. Este tipo de lógica está más próxima a la realidad de los videojuegos actuales, puesto que en pocas ocasiones se tiene una certidumbre plena.

8. **Modelado del jugador.** Esta técnica consiste en modelar un perfil del comportamiento del jugador que el videojuego pueda emplear para predecir su comportamiento y así adaptar sus características (como el grado de dificultad) a ese perfil. Durante el juego, el perfil es continuamente refinado. Por ejemplo, en un juego de disparos se puede observar que el jugador tiene mala puntería con una determinada arma y puede emplearse para explotar esa debilidad u ofrecerle ventajas que la compensen, como una mayor exposición de los enemigos al campo abierto. Esta técnica se empleó en el videojuego *Face Cry* (Crytek, 2004). Una variante más potente consiste en modelar un perfil que sea capaz de imitar al jugador como si de él se tratase, con una sofisticación tal que pueda jugar por él. La empresa AiLive¹ comercializa una herramienta de desarrollo destinada a este tipo de modelado, cuyo máximo interés es la facilidad de desarrollo y su relación resultados/coste.
9. **Redes bayesianas.** Como la lógica difusa, las redes bayesianas son una técnica de razonamiento con incertidumbre. Una red bayesiana es un conjunto de nodos conectados de modo que éstos representan variables relacionadas con estados, características o eventos del mundo del juego. Las conexiones entre los nodos representan relaciones causales entre ellos. Aplicando inferencia probabilística sobre el grafo se puede inferir el valor de otras variables. Esta técnica permite al juego tener creencias acerca de un jugador obtenidas a partir de la información de la que dispone. Por ejemplo, en juegos de estrategia en tiempo real se puede emplear para estimar si el jugador habrá construido determinados tipos de unidades, basándose en aquellas que ya se sabe que ha construido. Su uso obliga al jugador a utilizar estrategias que intenten confundir al videojuego creando incertidumbre acerca de los datos que éste puede observar, como si el oponente fuera humano.

¹ Más información en <http://www.ailive.net/liveCombat.html>



10. **Sistemas de producción**, sistemas basados en reglas o sistemas expertos. Los sistemas expertos pretenden modelar el conocimiento de un experto expresándolo en forma de reglas. Este tipo de sistemas consisten en un conjunto de reglas y hechos manejados por un motor de inferencia que determina qué regla debe ser empleada en cada momento, resolviendo los conflictos que se den cuando varias reglas puedan ser aplicadas (se disparen) al mismo tiempo. En cierto modo, estos sistemas pueden verse como la evolución de los dirigidos por eventos. La comunidad académica creó un agente (o *bot*) para el videojuego Quake II (id Software, 1997) basado en 800 reglas diferentes determinadas manualmente pero el aprendizaje automático (algoritmos genéticos, aprendizaje por refuerzo, etc.) aplicado a la generación automática de este tipo de sistemas también puede dar lugar a modelos de jugador competitivos.
11. **Terreno inteligente**. El objetivo de esta técnica es otorgar inteligencia a objetos inanimados de modo que el agente pueda preguntarle qué hace y cómo se usa, no necesitando así haber sido programado explícitamente para interactuar con ese objeto. Esta técnica se basa en la teoría de la facilitación (*affordance theory*) que dice que el diseño de un objeto indica un cierto tipo de interacción con él de modo que, por ejemplo, una puerta sin pomo sólo puede abrirse empujándola por el lado que no tiene bisagra. El videojuego Los Sims (Maxis, 2000) emplea esta técnica para desplazar buena parte de la inteligencia a los objetos de modo que los agentes pueden interactuar con nuevos objetos creados por medio de expansiones oficiales o por usuarios anónimos. Las expansiones oficiales de Los Sims le convirtieron en el videojuego más vendido de la historia, rentabilizando con creces este sistema. Un ejemplo de uso es incluir en un objeto hamburguesa información que indica que satisface la hambruna y que para usarlo hay que cogerlo con las dos manos y llevárselo a la boca.
12. **Planificación**. El propósito de la planificación es buscar una secuencia de acciones que permitan al juego cambiar la configuración actual del mundo para conseguir otra determinada que, comúnmente, le es más beneficiosa. Las acciones tienen una serie de precondiciones que deben cumplirse para poder ser aplicadas y una serie de efectos que tienen lugar al hacerlo. Las precondiciones pueden bien estar en la configuración de la que se parte bien haber aparecido tras la aplicación de una



acción previa en la secuencia. Una planificación efectiva depende de la elección del algoritmo de búsqueda, la representación del mundo del juego y el conjunto de acciones disponibles. Esta es la técnica en la que se centra este proyecto.

El videojuego F.E.A.R. (Monolith Productions, 2005), que sirvió de inspiración a este proyecto, servirá ahora como ejemplo de uso de este tipo de técnicas en conjunción con otras más extendidas que ya se han visto. La IA de este videojuego gira alrededor de un sistema de planificación ligero, diseñado para hacer posible su uso en tiempo real, para lo cual se empleó una representación sencilla del mundo [Orkin, 2004], muy dependiente del funcionamiento concreto del videojuego. El sistema de planificación se utilizó sólo para modelar la inteligencia individual de cada soldado, consiguiendo un comportamiento emergente a ese nivel. Un sistema de agrupación de individuos en escuadrones permitió un comportamiento emergente de grupo basado en el individual y, por tanto y en última instancia, en el funcionamiento del sistema de planificación. Dicho sistema de planificación no sólo dependía del videojuego en la forma de representar el mundo sino en la profunda integración con él a través de una arquitectura de pizarra [Orkin, 2005]. El sistema de IA de F.E.A.R. fue, a la postre, uno de sus máximos valedores para la consecución de la estupenda crítica que cosechó.

2.3 Planificación automática

La meta principal de este trabajo es el desarrollo de un sistema de planificación automática pero, ciertamente, no es objetivo de este documento explicar la planificación automática en toda su vasta extensión, sino exponer la idea básica con el fin de que pueda servir de punto de partida para la recopilación de cuanta información al respecto se considere necesaria. Es por ello que en este apartado se preferirá la vía intuitiva y sencilla a la hora de introducir los conceptos fundamentales, optando por algo más de formalismo en la primera definición del concepto principal.

2.3.1 Noción básica

La planificación es la parte de la actuación que conlleva razonamiento. Es un proceso de deliberación abstracto y explícito que elige y organiza acciones mediante la anticipación de sus resultados esperados. Esta deliberación pretende alcanzar de la mejor manera posible unos objetivos preestablecidos. La planificación automática es un



área de la Inteligencia Artificial que estudia este proceso de deliberación computacionalmente [Ghallab *et al.*, 2004].

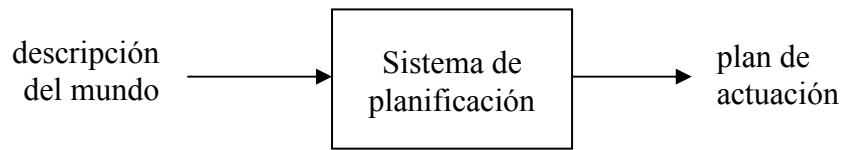


Ilustración 9: Diagrama intuitivo de un proceso de planificación

El proceso de planificación recae en unos sistemas especialmente diseñados para ello (comúnmente denominados **planificadores**) que reciben como entrada una descripción del mundo sobre el que actuar y dan como resultado un **plan** de actuación (o plan, a secas) a ejecutar sobre el mismo. A continuación se explicarán cada uno de estos elementos empleando para ello el sencillo ejemplo práctico que se enuncia a continuación:

Arancha es la encargada de que en su casa no falte pan a la hora de la comida, pero ella no tiene dinero para comprarlo. Tanto su padre como su madre pueden dárselo. Su padre se encuentra en la oficina, mientras que su madre se encuentra en casa, como ella. ¿Cuál es la mejor manera de que Arancha compre el pan?

El simplismo del ejemplo evidencia que no requiere de un sistema como el desarrollado en este proyecto para ser resuelto. Cualquier humano podría resolverlo sin problemas. Así, es muy probable que Arancha, tras evaluar distintas formas de conseguir su **objetivo**, distintos planes, se debatiera entre los siguientes:



Plan 1	Plan 2
<ol style="list-style-type: none">1. Pedir dinero a su madre.2. Ir a la panadería.3. Comprar el pan.4. Volver a casa.	<ol style="list-style-type: none">1. Ir a la oficina.2. Pedir dinero a su padre.3. Ir a la panadería.4. Comprar el pan.5. Volver a casa.

Tabla 1: Planes de ejecución intuitivos

Siendo ambos planes igual de efectivos, puesto que con cualquiera de ellos Arancha consigue comprar pan, seguramente se decantaría por el primero, puesto que su **coste** es menor desde el punto de vista del número de acciones necesarias: requiere 4 frente a las 5 del segundo. Con ello Arancha habría resuelto el problema en su vertiente estratégica. Ahora sólo tendría que resolver el problema en su vertiente táctica, es decir, llevando a cabo las acciones del plan que ha elaborado.

El hecho de que Arancha pueda resolver un problema como este con tan poco esfuerzo y tanta naturalidad oculta la complejidad real del proceso de razonamiento que ha llevado a cabo. En primer lugar, Arancha habrá identificado una serie de **objetos** que intervienen en el problema, como son sus padres, ella misma, su casa o la oficina. También habrá identificado una serie de modelos de acción, u **operadores**, como ir a un lugar, pedir dinero a alguien o comprar el pan. Esos operadores le han servido como plantilla para obtener una serie de **acciones**, como pedirle dinero a su madre o ir a la oficina (se dice que las acciones son *instancias*¹ de los operadores). Además, le han permitido descartar otras acciones sin sentido, como pedir dinero a la oficina. Ella sabe que la oficina es un lugar, no una persona. Y es que lugar y persona son los **tipos** de objeto que se manejan en este problema y los operadores exigen que los objetos implicados, de haber alguno, tengan un cierto tipo. Por otro lado, Arancha ha detectado una serie de modelos de **hechos**, o **predicados**, cuya certeza o no debe verificar en algún momento dado, como estar en un lugar o tener dinero. Al igual que sucede con los operadores, en esos predicados pueden o no estar implicados objetos y éstos deben ser de un cierto tipo. Por ejemplo, sabe que no se puede estar en su padre pero sí en la panadería. Su padre es una persona, no un lugar. Con todo esto, ha pensado cuál era la

¹ Deformación lingüística proveniente del inglés “instances”



situación actual, el **estado inicial**, del problema (ella está en casa, no tiene dinero, no tiene pan, etc.) y a qué situación quiere llegar (ella está en su casa y tiene pan), qué debe cumplir su **estado meta**. Ya se ha dicho que para ella es mejor, tiene menos coste, el plan que conlleve menos acciones, por lo que ya sólo le queda razonar cuál es el mejor de los planes, buscar qué conjunto de acciones es el mejor, **planificar** en definitiva. Pero para elaborar los planes necesita organizar de algún modo sus acciones. Y es que aún queda decir que percibió una serie de condiciones que deben darse de forma previa a la aplicación de los operadores que identificó y, por tanto, a las acciones que éstos describen. Por ejemplo, para comprar el pan hay que estar en la panadería y tener dinero. También se percató de que esas **precondiciones** pueden bien estar satisfechas por hechos ciertos en el estado inicial o por otros que son consecuencia de los **efectos** de aplicar los operadores y, por tanto, las acciones que definen. Por ejemplo, podría haber tenido inicialmente dinero o, como es el caso, conseguirlo como efecto de haberlo pedido.

Recapitulando, que Arancha identificó una serie de tipos, predicados y operadores, estos últimos con sus precondiciones y efectos que podrían intervenir en cualquier problema similar, independientemente de si, por ejemplo, las personas implicadas son sus padres o sus abuelos. Esos elementos sirven, dicho de otro modo, como plantilla para describir cualquier problema del mismo tipo y a esa plantilla, a esa definición, se le denomina **dominio**. Los objetos que identificó, así como todos los elementos que se derivan de ellos, como los hechos, las acciones y los estados inicial y meta, componen la definición del **problema**. Dominio y problema forman la descripción del mundo. Dada esta descripción del mundo, el planificador se ocupa de realizar de forma automática el proceso de razonamiento que necesitó Arancha para determinar qué plan, qué conjunto de acciones organizadas de qué manera, sería el mejor para ejecutarlo y lograr su objetivo. La diferencia entre ambos es que el planificador puede servirse de la potencia de cálculo de las máquinas para dar con la solución a problemas mucho más complejos de los que la mente humana puede abarcar, y en mucho menos tiempo.

Sabiendo que a los sistemas descritos se les prefiere llamar planificadores y que a los planes de actuación se les denomina comúnmente planes, a secas, el diagrama inicial queda como sigue:



Ilustración 10: Diagrama intuitivo, de mayor detalle, de un proceso de planificación

En este proyecto, tanto la descripción del dominio y el problema como la ejecución del plan le son ajenos al planificador. Dicho de otro modo, que aquí se abarca la construcción del planificador y la definición de un formato de descripción de las entradas (dominio y problema) y salidas (plan) del mismo que les permitan a otros, ya sean humanos (Arancha, por ejemplo) o sistemas automáticos (como un videojuego), interactuar con él para que les ayude a solventar situaciones que involucren la resolución de problemas de planificación.

2.3.2 Planificación clásica

Aunque con salvedades, la definición de planificación en la que mejor encaja el sistema objeto de este trabajo es la que [Ghallab *et al.*, 2004] denomina clásica. La **planificación clásica** es aquella que trata con mundos con un número finito de estados, estáticos y completamente observables y maneja metas restringidas, tiempo implícito y planes secuenciales. Para ayudar a entender esta definición, a continuación se aclaran cada uno de estos conceptos.

- Se dice que el número de estados del **mundo** es finito cuando no existen acciones que construyan nuevos objetos en el mundo ni se manejan variables numéricas asociadas al estado. Por otro lado, el que sea estático implica que se mantiene en el mismo estado hasta que se le aplica alguna acción. Finalmente, si es completamente observable significa que se tiene un conocimiento completo de su estado.
- Las **metas** se consideran restringidas si consisten sólo en la especificación de un conjunto de estados meta de modo que el objetivo sea alcanzar cualquiera de ellos. Se consideran metas extendidas aquellas que imponen otro tipo de restricciones, como que el plan haga pasar o no por determinados estados o que tenga un cierto número de acciones.
- El **tiempo** se considera implícito cuando las acciones no tienen duración sino que son instantáneas, y el tiempo no es tenido en cuenta para poder manejar, por ejemplo, restricciones en la secuencia de acciones.



- El **plan** es secuencial cuando consiste en una lista ordenada de acciones. Otros tipos de planes permiten conjuntos de acciones parcialmente ordenadas, secuencias con saltos condicionales dependiendo del estado, autómatas que indican qué acción aplicar en función a la secuencia de estados por la que se ha pasado, etc.

Como ya se ha dicho, esta definición es la que mejor describe el sistema sin que éste acabe de ajustarse por completo a ella. Por ejemplo, el sistema puede manejar opcionalmente variables numéricas en el estado, dependiendo del problema. Por otro lado, que sea o no estático el mundo es algo que también depende del problema, aunque en muchas ocasiones puede realizarse una simplificación que asuma dicha estaticidad. Con el tiempo sucede algo similar.

CAPÍTULO 3

OBJETIVOS

Si te caes siete veces, levántate ocho



3 Objetivos

El objetivo principal de este proyecto es la **creación de un sistema de planificación automática para su uso internacional en videojuegos desarrollados para la videoconsola Xbox 360 mediante la tecnología XNA, licenciado como *software* libre**. Este objetivo pretende maximizar la aportación de la Universidad a la comunidad mundial de desarrolladores de ese tipo de videojuegos mediante algo para cuya construcción se requieren los conocimientos técnicos avanzados que posee un Ingeniero en Informática especializado en Inteligencia Artificial.

Del objetivo principal se deducen todos los secundarios¹, a saber:

1. **El idioma del producto resultante debe ser el inglés.** No tiene sentido crear un sistema para la comunidad mundial de desarrolladores si éste se hace en un idioma desconocido para la mayoría, como el español. El producto incluye tanto el código fuente como la documentación, exceptuando esta memoria por tratarse de material exigido como requisito para la consecución de un título oficial expedido por una universidad española.
2. **El sistema de planificación debe ser funcional.** Se pretende crear algo utilizable por la comunidad de desarrolladores, lo cual no se consigue sin un sistema plenamente funcional. Por tanto, el resultado de este proyecto ha de ser la implementación del sistema de planificación.
3. **El producto resultante debe facilitar su continuación por terceros.** Se busca fomentar al máximo que el proyecto pueda ser continuado por cualquiera con los conocimientos necesarios dentro de las posibilidades que ofrece el *software* libre. Para ello, tanto la documentación como el código generado deben estar escritos para ser leídos por terceros.
4. **El producto resultante debe facilitar su uso por terceros.** La comunidad de desarrolladores para la que se crea el producto no tiene, por norma general, conocimientos técnicos avanzados en Inteligencia Artificial. Por tanto, debe facilitarse el uso del resultado basándose en la abstracción de aquellos conceptos menos relevantes de cara a su utilización.

¹ Los números no pretenden establecer ningún orden entre objetivos, sino servir de identificadores



5. **El sistema de planificación debe ser genérico.** Se pretende dotar a la comunidad de desarrolladores de una herramienta susceptible de ser utilizada en cualquier tipo de videojuego. Sólo siendo genérica es posible cumplir esta premisa.
6. **El sistema de planificación debe ser eficiente.** Se busca dar a la comunidad de desarrolladores una herramienta que les sea útil en la consecución de sus metas y la eficiencia es un factor muy importante dentro de los objetivos de un videojuego. Sólo un producto eficiente será utilizado por la comunidad.
7. **El sistema de planificación debe ser flexible.** De cara a potenciar la utilidad de la herramienta por parte de la comunidad de desarrolladores, la flexibilidad es una característica tan importante como la eficiencia. El sistema resultante debe conjugar ambas de modo que la primera perjudique lo menos posible a la segunda.
8. **El sistema de planificación debe ser multiplataforma.** La tecnología XNA de Xbox 360 se sustenta sobre la base del .NET Compact Framework y el .NET Framework de PC es un superconjunto de éste. Siendo un sistema genérico, debe permanecer desacoplado de la lógica del videojuego, por lo que no tiene sentido que haga un uso directo de ningún componente de XNA fuera de los provistos por el .NET Compact Framework y, por tanto, de los del .NET Framework. Un diseño que lo contemple desde el inicio debería hacer trivial la consecución de un sistema compatible, al menos, tanto en PC como en Xbox 360.
9. **El producto resultante debe seguir los estándares de .NET.** Sin perjuicio de seguir cualquier otro estándar de desarrollo, el producto resultante debe seguir los existentes para la plataforma .NET, con objeto de minimizar la curva de aprendizaje de cara a su utilización o continuación. Serán de especial relevancia los estándares de diseño, de calidad del código final y de intercambio de datos.

CAPÍTULO 4

DESARROLLO

*No se puede desatar un nudo
sin saber cómo está hecho*



4 Desarrollo

En este apartado se expondrá toda aquella información que se considera de relevancia en relación a la construcción del sistema objetivo. Se comenzará explicando la metodología de desarrollo empleada para pasar, posteriormente, por las distintas fases que componen el proceso de construcción. Las representaciones gráficas, cuando se estimen oportunas, se realizarán empleando el lenguaje UML¹ o algún otro tipo de notación comúnmente extendida, según el caso. Por su parte, las referencias al sistema construido se harán por su nombre: Aran.

4.1 Metodología

La construcción de Aran se considera perteneciente al área de I+D+i², primando la búsqueda de resultados sobre los recursos empleados, especialmente los temporales. Por otro lado, la envergadura del proyecto se considera pequeña, siendo llevada a cabo por una única persona con asesoramiento de terceros. Ambas razones hacen desestimar la adopción de una metodología de desarrollo estricta y, normalmente, orientada a la conjugación de la consecución de objetivos con el menor número de recursos, como suelen ser las existentes. En su lugar se ha preferido seguir una metodología propia **adaptada a las necesidades del proyecto** y basada en las ideas generalizadas en esta área.

La metodología empleada **se centra en el análisis, el diseño, la implementación y la calidad**, obviando fases que se consideran fuera de las necesidades del proyecto, como pudieran ser el estudio de viabilidad, la planificación, el presupuesto, la implantación, la aceptación, el mantenimiento o la seguridad, entre otras. No es que estas fases no existan o no sean necesarias, sino que su peso en este proyecto no se considera lo suficientemente relevante como para ser plasmado en una documentación como la que supone esta memoria.

La metodología aplicada centra sus esfuerzos en las áreas mencionadas **con el fin de cumplir los objetivos** propuestos, especialmente los relacionados con la consecución de un producto funcional y un desarrollo que pueda ser continuado por terceros. Al respecto de este último punto, se considera de especial interés la

¹ Siglas de “Unified Modeling Language”

² Investigación, Desarrollo e innovación



diferenciación entre el diseño y la implementación y, dentro del diseño, entre conceptual y detallado. Esta separación maximiza las posibilidades de continuidad del proyecto, bien sobre sus resultados en la tecnología empleada, bien sobre la utilización de los conceptos en los que se basa, sin que lo primero interfiera en lo segundo.

La aplicación de la metodología hace de las siguientes herramientas los tres pilares fundamentales sobre los que sustentarla, a saber:

1. **Cuaderno de bitácora.** Se trata de un diario en el cual se han ido plasmando informalmente todos aquellos datos que se han considerado de relevancia para el desarrollo del proyecto. En él se expresan conceptos, posibilidades, análisis, resultados, opiniones, reuniones, referencias y un largo etcétera de información no estructurada.
2. **Lista de tareas por hacer.** Consiste en una lista de tareas identificadas y acompañadas de su descripción, ordenadas por prioridad de ejecución. La lista va siendo modificada mediante la adición de nuevas tareas descubiertas durante el proceso de desarrollo, la eliminación de las cumplidas o la modificación de las características de las existentes, como su prioridad.
3. **Repositorio de ficheros.** Se trata de un sistema de gestión de ficheros que permite administrar una base de datos para almacenar la evolución temporal de los mismos, con la posibilidad de incluir descripciones adscritas a los cambios realizados sobre cada uno de ellos y recuperar versiones actuales o pasadas. Este tipo de sistemas se les conoce comúnmente como gestores para el control de versiones y suelen ser empleados para la administración concurrente de ficheros fuente. A pesar de que este desarrollo es llevado a cabo por una única persona, el sistema de control de versiones se ha considerado útil para el almacenamiento de información histórica recuperable.

4.2 Análisis

En este apartado se realizará un análisis no exhaustivo del problema a resolver, plasmado en la descripción de los casos de uso.



4.2.1 Actores

La forma en la que se presentan los sistemas de planificación automática dependen en buena medida del uso que se le va a dar y Aran no es una excepción. Es más, podría decirse que su dependencia es mayor de lo habitual, pues varios tipos de usuario pueden hacer uso de él mientras que lo común suele ser un único tipo. La construcción del sistema deberá lidiar, por tanto, con una problemática que obliga a buscar una solución lo suficientemente flexible como para cubrir los deseos de unos y otros tipos de usuario pero, y esto es, si cabe, más importante, sin que éstos entren en conflicto. Así, por ejemplo, la facilidad de utilización que pueda requerir un usuario no deberá afectar a la eficiencia que exija otro. A continuación se exponen los distintos tipos de usuario que se estima van a hacer uso de Aran.

En primer lugar, se encuentran los desarrolladores que, con probabilidad alta, desconocen las técnicas sobre las que se sustenta este producto. Ello hace que la forma final del mismo deba proveer a este tipo de usuario de un mecanismo de abstracción de dichas técnicas de modo que pueda emplear conceptos más sencillos de entender a la hora de utilizar Aran. Para este tipo de usuario prima la funcionalidad sobre la eficiencia, entre otras cosas porque le es difícil conseguir esto último por falta de conocimiento. Aún así, querrá moldear el sistema a sus necesidades dentro de sus posibilidades, siendo la adaptación más común aquella que afecta al manejo del lenguaje de comunicación con el planificador. A este rol, el más común dada la orientación del presente proyecto, se le denominará **desarrollador básico**.

En segundo lugar, se puede identificar otro tipo de desarrollador con conocimientos técnicos suficientes como para entender Aran a bajo nivel. Este tipo de usuario tenderá a la explotación máxima de las posibilidades del producto y requerirá poder adaptarlo a sus necesidades para la consecución de este objetivo. Querrá, en definitiva, extender la funcionalidad básica de Aran o hacerse con aquellas partes del control de la misma que le permitan aumentar la eficiencia, siempre por medio de un uso específico de los interiores del sistema. Para este tipo de usuario prima la eficiencia por encima de la funcionalidad, puesto que es capaz de crear esta última de forma que se adapte más a sus necesidades. Sin embargo, es un usuario práctico que sólo rehace aquellas partes que puede mejorar. En este sentido, no abordará el cambio de las bases de un sistema que, siendo genéricas, sean a su vez eficientes. Por otro lado, el hecho de



que su máxima sea la eficiencia no le impide rehusar la misma cuando no le es necesaria, realizando en ese caso las mismas labores de un usuario básico. A esta clase de usuario se le denominará **desarrollador avanzado**.

Los dos tipos de roles ya presentados tienen en común el ser desarrolladores pero existe otro tipo de usuario cuya prioridad no es la eficiencia del sistema sino la capacidad de utilización del mismo para la prueba de diferentes modelos conceptuales, diferentes representaciones, de un mismo entorno, de un mismo dominio. En este sentido no le interesa siquiera, como sí sucede con el desarrollador básico, la adaptación del lenguaje de comunicación con el planificador. Le basta el existente. Tampoco le preocupará la integración eficiente de Aran en otro sistema mayor. A este tipo de usuario se le denominará **investigador**. Es obvio que la investigación no tiene por qué limitarse a este campo de actuación. Si aquí se identifica al investigador con estas labores es porque, en caso de ocuparse de tareas ya comentadas, se le considera en representación del rol de desarrollador. Dicho de otro modo, un desarrollador puede desempeñar labores de investigación.

Hasta ahora todos los roles identificados se corresponden con usuarios que pretenden resolver problemas de planificación a distinto nivel. Sin embargo, un planificador no es más que un algoritmo de búsqueda especializado en un determinado tipo de representación. Si ese algoritmo de búsqueda no hace un uso específico de dicha representación es lógico pensar en su reutilización con otras representaciones que no se ajustan a las restricciones requeridas por un planificador. Surge así un cuarto tipo de usuario que pretende hacer uso de los algoritmos de búsqueda que, habiendo sido desarrollados para su uso como parte de Aran, son lo suficientemente genéricos como para ser empleados en otros problemas no relacionados con la planificación. Se identificará este tipo de usuario como **buscador**. Cualquiera de los otros tipos ya mencionados puede llevar a cabo labores relacionadas con este perfil.

4.2.2 Casos de uso

Sobre la base de las descripciones realizadas, se puede elaborar el siguiente diagrama de casos de uso.

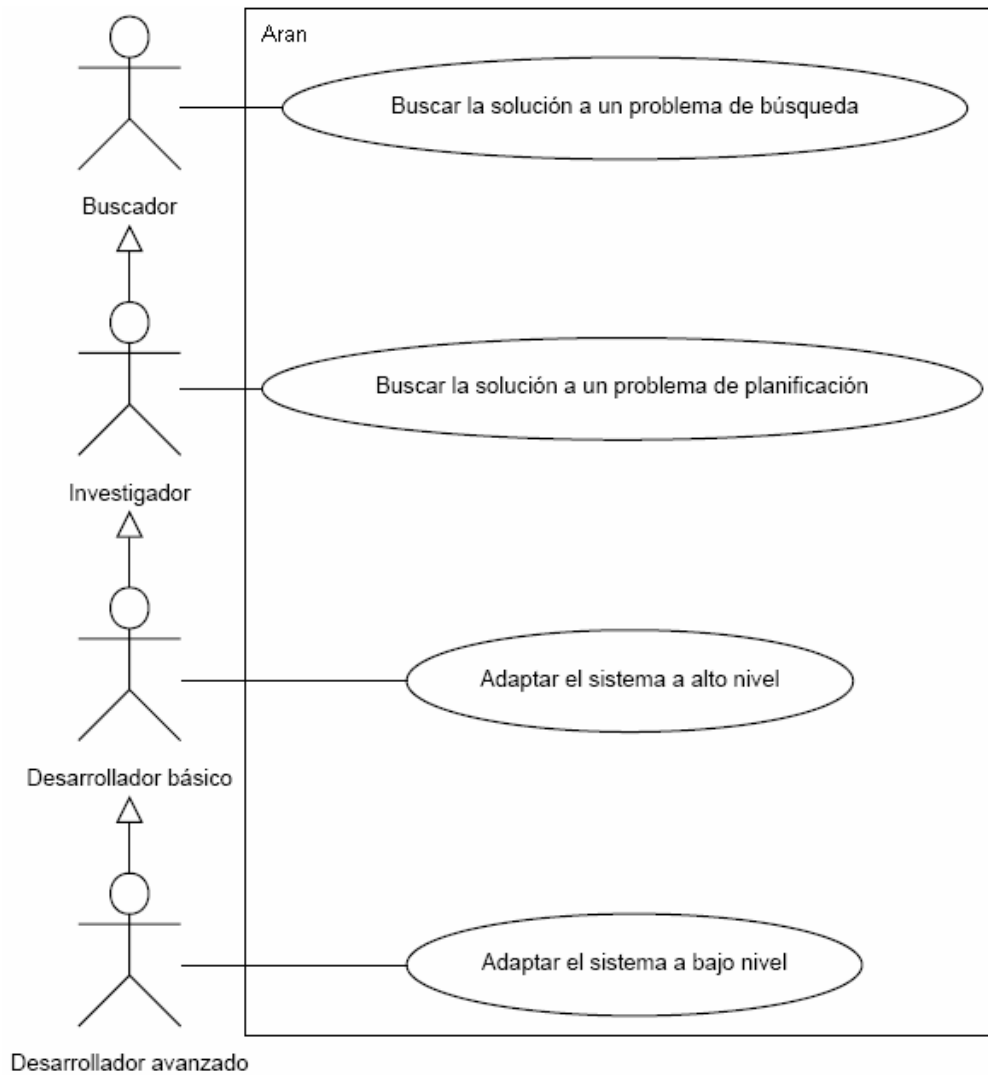


Ilustración 11: Diagrama de casos de uso

El diagrama de la Ilustración 11 refleja cuatro actores que se corresponden con los cuatro perfiles de usuario identificados así como los casos de uso relacionados con cada uno. A continuación se describen informalmente estos últimos.

- **Buscar la solución a un problema de búsqueda.** El usuario selecciona un algoritmo de búsqueda de entre los provistos por Aran, siempre que no dependan de su núcleo. A continuación, proporciona al algoritmo todos los elementos que requiere para gestionar el conocimiento acerca del problema. Finalmente, ejecuta el algoritmo y obtiene un resultado. El resultado puede ser que se ha resuelto o no el problema y, en caso de resolución, cuál es la solución.
- **Buscar la solución a un problema de planificación.** El usuario selecciona un determinado sistema de representación de entre los admitidos por el traductor de



Aran. Seguidamente, expresa un problema empleando el método permitido por el sistema de representación. A continuación ejecuta el traductor, obteniendo los elementos del núcleo de Aran que el algoritmo de búsqueda requiere para gestionar el conocimiento sobre el problema. Traducido el problema, el usuario selecciona un algoritmo de búsqueda de entre los provistos por Aran. A continuación, ejecuta el algoritmo y obtiene un resultado. El resultado puede ser que se ha resuelto o no el problema y, en caso de resolución, cuál es la solución, es decir, el plan. Finalmente, el usuario emplea algunos de los elementos obtenidos de la traducción para interpretar el plan.

- **Adaptar el sistema a alto nivel.** El usuario construye una nueva funcionalidad de alto nivel. A continuación, la relaciona con Aran. Finalmente, emplea el conjunto en su beneficio. Una posible adaptación de alto nivel es la construcción de un sistema de representación compatible con el traductor de Aran.
- **Adaptar el sistema a bajo nivel.** El usuario construye una nueva funcionalidad de bajo nivel. A continuación, la relaciona con Aran. Finalmente, emplea el conjunto en su beneficio. Una posible adaptación de bajo nivel es la construcción de un traductor. Otra posible adaptación de bajo nivel es la adición de nuevas funciones al núcleo, como otro selector de acciones.

4.3 Diseño conceptual

En este apartado se recogen las decisiones tomadas en relación al diseño de Aran, expresadas desde un punto de vista conceptual con el fin de maximizar su reutilización en proyectos similares.

4.3.1 Arquitectura del sistema

A fin de dar solución a los requerimientos que cada tipo de usuario puede tener con respecto a la utilización de Aran, se ha decidido subdividirlo en varios subsistemas interrelacionados entre sí de modo que se minimicen las dependencias entre ellos. Este diseño **favorece el desacoplamiento** entre componentes de modo que puedan ser utilizados con independencia del resto o dependiendo del menor número de otros componentes.

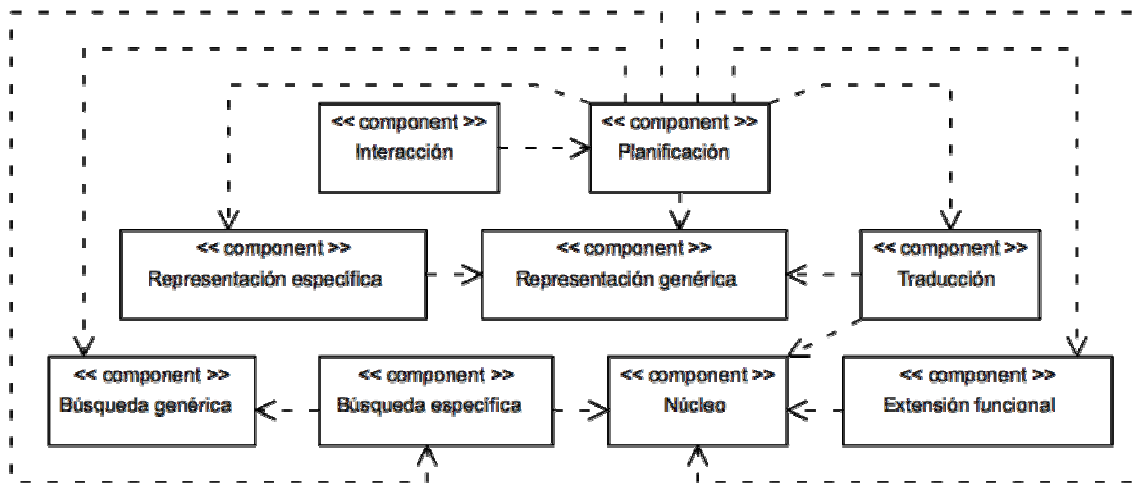


Ilustración 12: Diagrama de componentes del sistema

La Ilustración 12 recoge las relaciones de dependencia existentes entre los distintos componentes. A continuación se expone brevemente el papel de cada uno de ellos en el conjunto. En los apartados siguientes se explicarán con mayor detalle.

- **Subsistema núcleo.** Es el encargado de la definición de las estructuras representativas de los elementos esenciales, es decir, los estados y las acciones. Define un marco de trabajo para el uso eficiente de estructuras estáticas que gestionan ambos elementos al tiempo que define las interfaces necesarias para la adición de nuevas capacidades mediante funciones. Como base que es, no tiene ninguna dependencia con otros subsistemas. En lo sucesivo se hará referencia a este subsistema simplemente como “el núcleo”.
- **Subsistema de extensión funcional.** Se ocupa de la definición de las funciones de extensión de las capacidades del núcleo. Su labor se centra en la adición de funciones de manejo numérico, si bien podría contener cualquier tipo de función que cumpla las interfaces definidas en el núcleo. Puesto que necesita conocer dichas interfaces, depende del núcleo.
- **Subsistema de búsqueda genérica.** Es el encargado de la definición de la infraestructura necesaria para la creación y utilización de algoritmos de búsqueda genéricos. Dicha infraestructura contiene los componentes comunes a todo algoritmo, así como la definición de algunos de ellos. Se centra en la búsqueda hacia delante, si bien podría contener algoritmos de cualquier otro tipo. Ser genérico le permite no depender de ningún otro subsistema.



- **Subsistema de búsqueda específica.** Se trata de una extensión del subsistema de búsqueda genérica que, empleando los elementos definidos por aquél, hace también uso de los pertenecientes al núcleo con el fin de definir algoritmos de búsqueda específicos. De lo explicado se deduce su dependencia tanto con el núcleo como con el subsistema de búsqueda genérica.
- **Subsistema de representación genérica.** Se ocupa de indicar las interfaces que debe cumplir un sistema de representación que permita la definición de un mundo sobre el que realizar la planificación, dividida en dominio y problema. Dicho de otro modo, indica cuál debe ser el resultado de la representación, pero no cómo se ha de conseguir. Ser genérico le hace ser independiente del resto de subsistemas.
- **Subsistema de representación específica.** Es el encargado de la definición de un sistema de representación que cumpla las interfaces definidas por el subsistema de representación genérica. En otras palabras, define cómo se consigue una representación como la indicada por dicho subsistema. Puesto que requiere conocer las interfaces que debe cumplir, depende del subsistema de representación genérica.
- **Subsistema de traducción.** Su misión es la de traducir los elementos devueltos por el subsistema de representación genérica en elementos del núcleo. En definitiva, hacer de puente entre el método admitido por el subsistema de representación específica y las estructuras de bajo nivel del núcleo. Lo descrito permite entender su dependencia tanto con el subsistema de representación genérica como con el núcleo.
- **Subsistema de planificación.** Es el encargado de coordinar al resto de subsistemas presentados hasta ahora para hacer posible el proceso de planificación completo. Para ello toma una definición de alto nivel de un problema a resolver realizada en el modo admitido por el subsistema de representación específica y hace uso del subsistema de traducción para obtener las estructuras necesarias para el empleo del núcleo. Con ayuda del subsistema de búsqueda genérica o específica obtiene o no un plan que resuelve el problema, representado mediante elementos definidos en el núcleo. En caso de resolución, utiliza nuevamente el subsistema de traducción para obtener, a partir del plan de bajo nivel, elementos de alto nivel pertenecientes al subsistema de representación específica. Dichos elementos conforman el plan de



alto nivel. De lo dicho se puede deducir que este subsistema depende de todos los demás salvo el de interacción.

- **Subsistema de interacción.** Es el ocupado de transmitir las órdenes del usuario al subsistema de planificación, así como de hacerle llegar los mensajes de éste al usuario. Es, en definitiva, la interfaz de usuario del subsistema de planificación. Depende, por tanto, de dicho subsistema.

La Ilustración 13 servirá para aclarar las explicaciones dadas, pues en ella pueden observarse, a alto nivel, las interacciones existentes entre los distintos componentes del sistema.

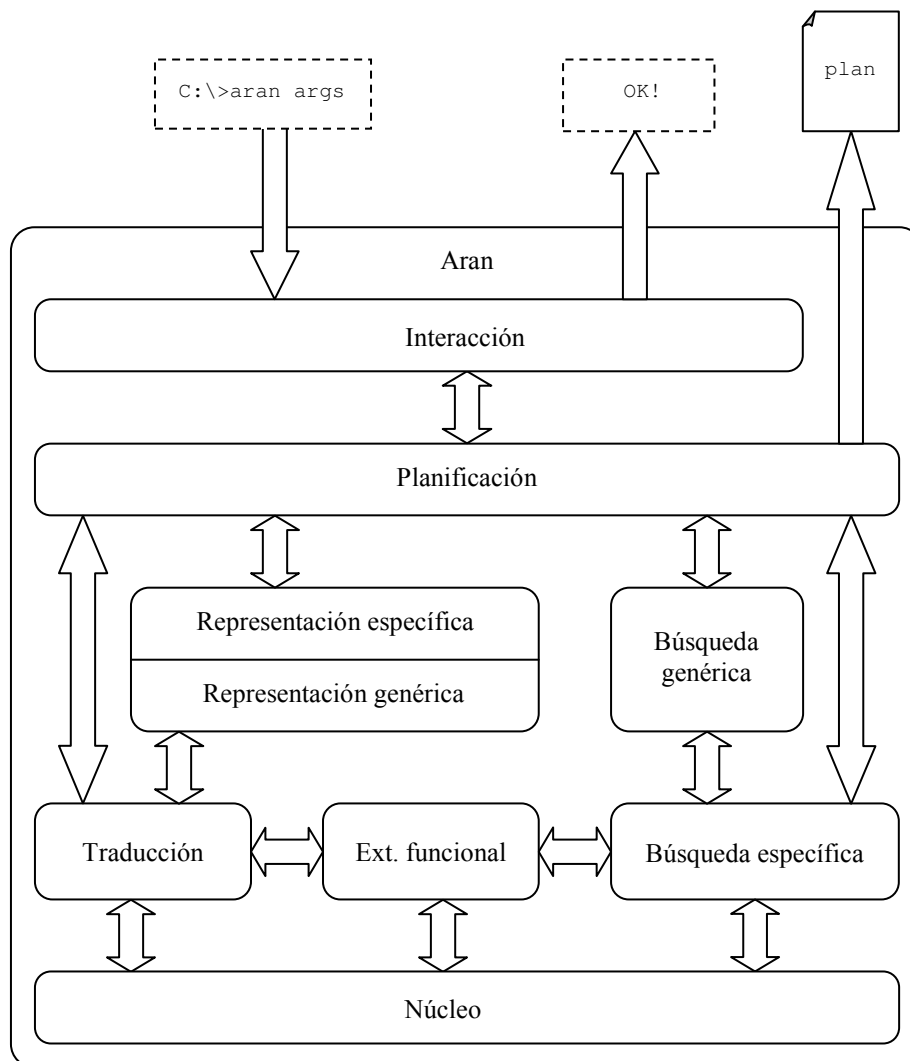


Ilustración 13: Diagrama de interacciones entre los componentes del sistema

Por otra parte, la Ilustración 14 muestra el algoritmo general del funcionamiento del sistema a alto nivel, desde que recibe las entradas hasta que genera las salidas.

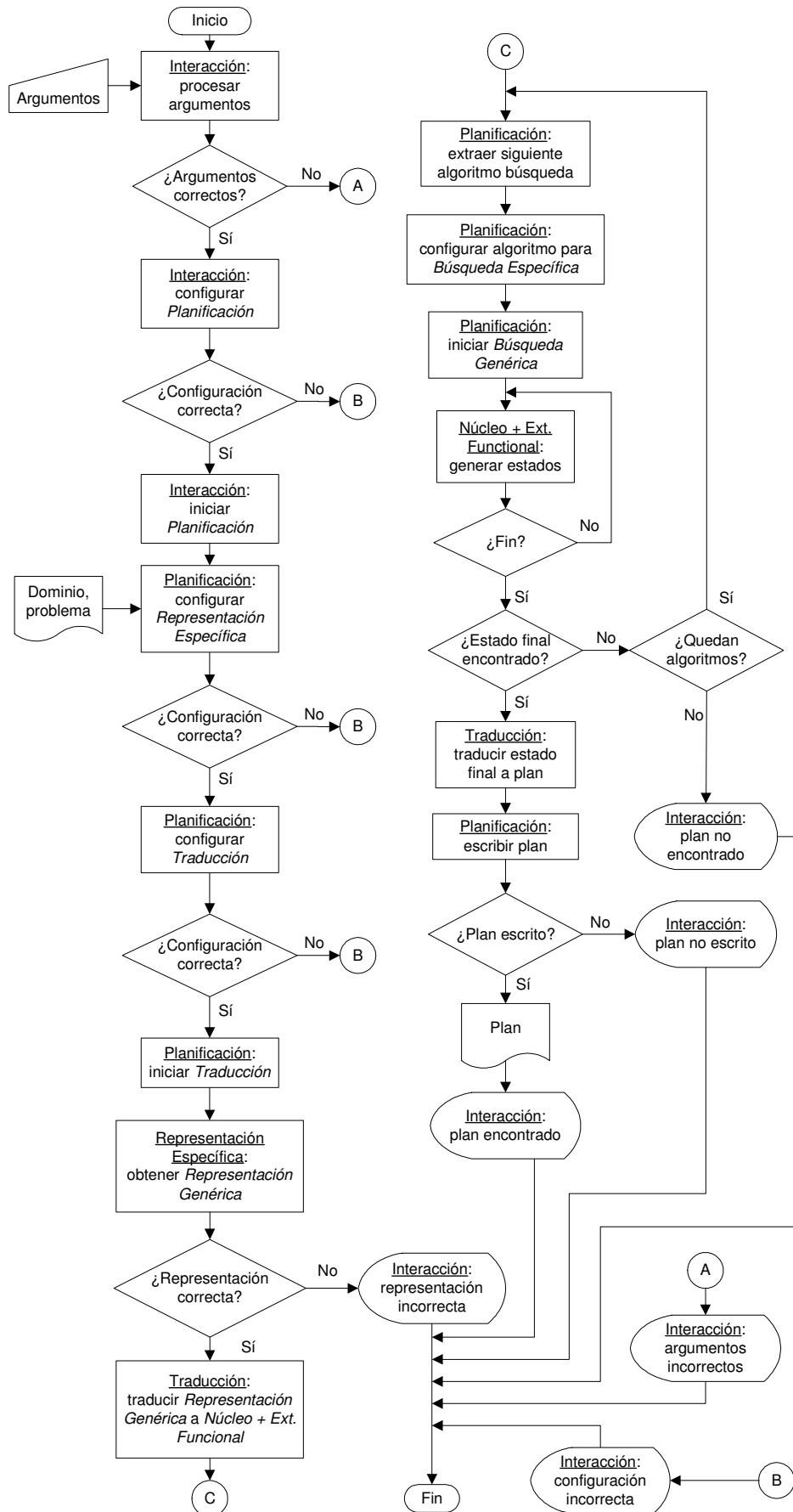


Ilustración 14: Diagrama de flujo del sistema



Volviendo de nuevo la estructuración en componentes, el diseño empleado busca, como ya se ha indicado, el mínimo acoplamiento entre éstos con el fin de favorecer la flexibilidad del sistema. Al tiempo, pretende conseguir la máxima cohesión entre los elementos pertenecientes a cada uno de los componentes para favorecer la eficiencia. Será, de este modo, **el usuario quien decida en qué forma pretende hacer uso del sistema y, por tanto, cuál es su prioridad: flexibilidad o eficiencia**. En este sentido, los componentes descritos pueden clasificarse en tres niveles de abstracción: alto, medio y bajo.

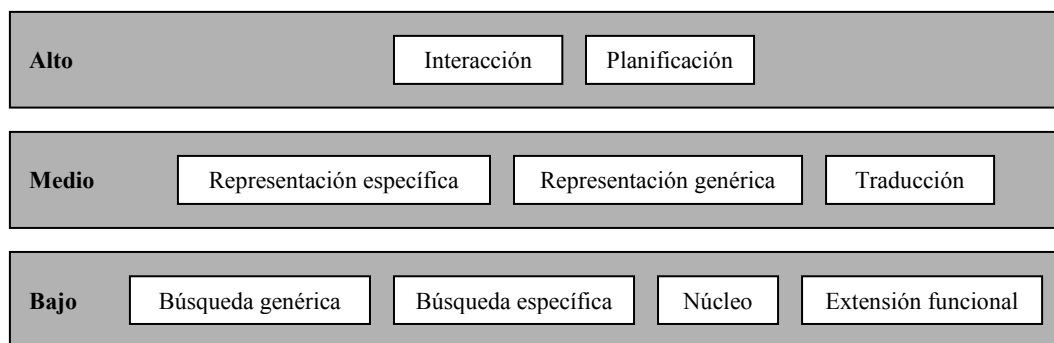


Ilustración 15: Niveles de abstracción

Dicha clasificación, representada en la Ilustración 15 y favorecida por la minimización del acoplamiento, busca cumplir los requerimientos de cada tipo de usuario. El nivel **bajo** es aquél en el que deberían trabajar los perfiles **buscador** y **desarrollador avanzado**. El nivel **medio** debería ser el natural del **desarrollador básico** mientras que el nivel **alto** debería ser en el que moverse como **investigador**, salvo cuando se quiera hacer uso de los algoritmos de búsqueda genéricos.

No es difícil percatarse de que el gran canalizador del desacoplamiento en este diseño es el subsistema de planificación, pues las dependencias del sistema en su conjunto han sido derivadas a éste en concreto. Por su parte, la funcionalidad ha seguido el camino inverso, residiendo en su mayor parte en el resto de componentes. Con estas decisiones, el diseño pretende hacer del **subsistema de planificación un elemento altamente reemplazable** en función de las necesidades del usuario. Su elevado número de dependencias hace que su adaptación pierda enteros frente a otras soluciones. Esto, que podría considerarse algo en contra es, en verdad, un elemento a favor dado que la utilización de Aran pasará, normalmente, por su integración dentro de otros proyectos más grandes. Esta integración afectará con alta probabilidad al subsistema mencionado y no a la funcionalidad que coordina. Este escenario se antoja altamente probable en el



desarrollo de videojuegos, de ahí su impacto en el diseño. En este sentido, **los niveles de abstracción bajo y medio constituyen un marco de trabajo** reutilizable en distintos ámbitos mientras que el nivel alto no lo es. Y es que, aunque no se ha mencionado hasta ahora, es obvio que el subsistema de interacción es otro elemento con alta probabilidad de ser sustituido cara a la integración. Su dependencia del subsistema de planificación le traslada su inflexibilidad, mayor aún si se tiene en cuenta que la interfaz que represente difícilmente se ajustará a todos los escenarios.

Los distintos subsistemas serán tratados en los siguientes apartados siguiendo un orden distinto al expuesto para facilitar su entendimiento.

4.3.2 Subsistema de representación genérica

El primero de los problemas al que debe enfrentarse cualquier usuario de un sistema de planificación es la transmisión de su conocimiento a dicho sistema. Una vez transmitido, el sistema empleará el conocimiento para llevar a cabo el proceso de razonamiento en busca de una solución, un plan.

El método por el cual el usuario transmita su conocimiento ha de serle indiferente al sistema puesto que, al fin y al cabo, a él sólo le interesan una serie de conceptos bastante abstractos relacionados entre sí. El cometido del subsistema de representación genérica es, precisamente, ese. **Proveer al sistema de un mecanismo de representación del conocimiento lo suficientemente abstracto como para permitir su implementación por parte de terceros de forma diversa.**

La pertenencia de la plataforma objetivo a la POO¹ hace pensar que, quizá, los conceptos que debería recoger el sistema de representación genérica del conocimiento deberían estar inspirados también en la POO. Aunque es éste un argumento de peso, se considera que una decisión así habría facilitado el aprendizaje inicial al usuario pero aquello que aprendiera le habría sido de escasa utilidad en un mundo, el de la planificación automática, en el que la mayoría de los sistemas emplean **representaciones clásicas**. Ejemplo de ello es la especificación de PDDL² que, siendo un lenguaje de reciente creación (1998), se inspira en este tipo de representación y, no hay que olvidarlo, pretende aglutinar los conceptos manejados por los distintos

¹ Siglas de “Programación Orientada a Objetos”

² Siglas inglesas de “Lenguaje de Definición de Dominios de Planificación”



planificadores desarrollados a nivel mundial, pues fue creado para su uso en las distintas ediciones de la IPC¹. Así pues, **se considera de mayor utilidad** no huir de este tipo de representación intentando conseguir un equilibrio entre la dificultad de desarrollo del sistema y su potencia descriptiva en el tipo de entorno al que va destinado: los videojuegos. De este modo, se ha considerado a bien limitar el sistema a los conceptos relacionados con la planificación más sencilla que recoge **PDDL**, la de tipo **STRIPS** (requisito :strips), yendo algo más allá mediante el **tipado** (requisito :typing) y el **manejo de números** (requisito :fluents). El tipado permite una mayor eficiencia del planificador, al reducir el número de acciones a generar, mientras que el soporte numérico se antoja especialmente interesante en videojuegos dado lo habitual del manejo de conceptos como munición, moneda, distancia, velocidad, nivel de daño, nivel de vida o tiempo. Este último hace pensar en si sería interesante admitir restricciones temporales a la hora de construir la secuencia de acciones. En este proyecto no se ha considerado necesario teniendo en cuenta los precedentes puesto que, como ya se expuso, la Inteligencia Artificial empleada hoy día en videojuegos es aún bastante sencilla, por lo que los conceptos manejados por el sistema construido se consideran suficientes para juegos actuales, más teniendo en cuenta que éste no es un proyecto destinado al uso profesional. Por otro lado, añadir capacidades temporales a un planificador como éste es suficientemente costoso como para no tenerlo en consideración en un proyecto de esta envergadura. Dicho esto, se entiende que, para su uso en videojuegos, es de interés poder representar la siguiente información²:

1. Una **lista de tipos** de objeto. Un tipo clasifica un objeto de modo que luego otros conceptos pueden emplear dicha clasificación para restringir el uso de los objetos a los de una determinada clase. Todo objeto ha de tener un tipo y cada tipo está representado por dos parámetros:
 - 1.1. Un **identificador** de tipo, unívoco en cualquier ámbito. Siempre que quiera hacerse una referencia a un tipo, se hará a través de su identificador.
 - 1.2. Una **relación de parentesco** con otro tipo. La relación de parentesco es opcional y permite definir jerarquías de tipos, de modo que un tipo hijo equivalga a uno padre a la hora de llevar a efecto las restricciones mencionadas.

¹ Siglas de “Internacional Planning Competition”

² El “Anexo B: Ejemplos de utilización del lenguaje” puede facilitar el entendimiento de lo que se expone a continuación, pues el lenguaje XML sigue de forma bastante fiel esta estructura.

El concepto de tipo es básico en POO y aquí tiene idéntico cometido. Aunque existen sistemas que no emplean tipos, como STRIPS¹, su uso está ampliamente extendido ya que permite al sistema de planificación **reducir las posibilidades combinatorias** a la hora de sintetizar elementos durante el proceso de razonamiento. Aunque conlleva un tratamiento de la información más complejo, la reducción combinatoria se traduce, en términos generales, en una mayor eficiencia y ésta es uno de los objetivos de este proyecto. Por otro lado, una representación con tipos es equivalente a una sin ellos si sólo se define un único tipo común a todo objeto. No puede decirse lo mismo del caso inverso. Por tanto, el empleo de tipos dota de mayor flexibilidad al sistema, y ese es otro de los objetivos del proyecto.

2. Una **lista de predicados**. Un predicado define la estructura que debe presentar un tipo de hechos de modo que todo hecho ha de ajustarse a la plantilla expuesta por algún predicado. Su interés es doble. Por un lado, ofrece al usuario un mecanismo de reducción de errores al impedir que defina hechos que no se ajusten a una plantilla preestablecida. Por otro, y más importante, dota al sistema de planificación de la información necesaria para la definición automática de hechos en función de los objetos de los que disponga en el ámbito de definición en el que se encuentre. La estructura que se ha decidido emplear para la representación de un predicado no difiere de la clásica:

- 2.1. Un **identificador** de predicado, unívoco en cualquier ámbito. Siempre que quiera hacerse una referencia a un predicado, se hará a través de su identificador.

- 2.2. Una **lista de parámetros**. Un parámetro define las características que un objeto debe cumplir para poder ser empleado como argumento de un hecho definido por el predicado. En este sentido, basta con que haga referencia al **tipo** del objeto.

La identidad del predicado la define únicamente su identificador, no considerándose la lista de parámetros a este respecto. Así, **se descarta la sobrecarga de predicados**, algo común en la POO. El porqué de este descarte se debe a que, siendo algo inusual en el mundo de la planificación automática, añadiría una complejidad al tratamiento de la información que difícilmente se vería compensada por la utilización que el usuario le daría. Por otro lado, se trata de una decisión

¹ Siglas de “Stanford Research Institute Problem Solver”



reconsiderable en un futuro, dada la facilidad de hacer compatible hacia atrás una medida como esta.

3. Una lista de **funciones**. Las funciones tienen un cometido similar al de los predicados, con la diferencia de que éstas son plantillas de definición de variables. El beneficio de su uso es idéntico al de los predicados, así como su estructura:

- 3.1. Un **identificador** de función, unívoco en cualquier ámbito. Siempre que quiera hacerse una referencia a una función, se hará a través de su identificador.

- 3.2. Una **lista de parámetros**.

En las funciones **no se admite sobrecarga** por idénticas razones a las expuestas en el caso de los predicados.

La gestión numérica es difícil, prueba de ello es que no todos los planificadores existentes la realizan, pero se ha decidido su consideración en la representación del conocimiento debido a que, de otro modo, se dificultaría el **manejo de recursos** ampliamente extendidos en el mundo de los videojuegos, como los ya mencionados **munición, moneda, distancia, velocidad, nivel de daño, nivel de vida o tiempo transcurrido**.

4. Una lista de **operadores**. Siguiendo con las plantillas, los operadores lo son de las acciones. Cumplen una función similar a los predicados y funciones, permitiendo al sistema de planificación la definición automática de acciones. La diferencia es que, en este caso, su uso permite al usuario no tener que explicitar nunca una acción, ya que todas son definidas automáticamente como parte del proceso de razonamiento. Esto simplifica la utilización del sistema, siendo éste otro de los objetivos del proyecto, al tiempo que favorece, como todos los conceptos ya mencionados, la reutilización del conocimiento. La representación de un operador consiste en los siguientes elementos:

- 4.1. Un **identificador** de operador, unívoco en cualquier ámbito. Siempre que quiera hacerse una referencia a un operador, se hará a través de su identificador.

- 4.2. Una **lista de parámetros**. Dentro de un operador, los parámetros son considerados a todos los efectos como objetos. Como tales, han de tener, además del **tipo** de objeto, un **identificador** unívoco en el ámbito del operador. Siempre que quiera hacerse una referencia a un parámetro, se hará a través de su identificador.

- 4.3. Una **lista de hechos**. Es común que un hecho sea utilizado en más de una ocasión dentro de un operador. En un modelo en el cual es el subsistema de



traducción el encargado de detectar esta reutilización, dicho subsistema ha de comparar todos y cada uno de los hechos con respecto al resto. Por otro lado, la identificación de un hecho requiere la verificación de su firma, compuesta tanto por el identificador del predicado cuya plantilla cumple como los identificadores de los objetos empleados como argumentos de tal plantilla. Esto se traduce en una pérdida de eficiencia que esta lista de hechos pretende reducir. Así, el subsistema de traducción sólo permitirá el uso de los hechos declarados en la lista de modo que sólo tendrá que comprobar entre sí los hechos de la misma, siendo siempre el número de comprobaciones menor o igual al necesario con el otro método. Identificando cada uno de los hechos de la lista se permite una reutilización eficiente basada en la referencia a los hechos por este identificador en lugar de por su firma. Así pues, cada hecho de la lista debe tener la siguiente estructura:

- 4.3.1. Un **identificador** de hecho, unívoco en el ámbito del operador. Siempre que quiera hacerse una referencia a un hecho, se hará a través de su identificador.
- 4.3.2. Una **relación de cumplimiento de plantilla** (*instanciación*¹) con un predicado.
- 4.3.3. Una **lista de argumentos**. Cada argumento es una relación con un objeto del ámbito de declaración del hecho. En este caso, un parámetro del operador, puesto que son los únicos objetos presentes en ese ámbito. El número y tipo de los argumentos debe coincidir con los del predicado.
- 4.4. Una **lista de variables** (*fluents* en PDDL). El propósito de de esta lista es muy similar a lo explicado con los hechos, así como la estructura de sus elementos:
 - 4.4.1. Un **identificador** de variable. Siempre que quiera hacerse una referencia a una variable, se hará a través de su identificador.
 - 4.4.2. Una **relación de cumplimiento de plantilla** (*instanciación*) con una función.
 - 4.4.3. Una **lista de argumentos**. El número y tipo de los argumentos debe coincidir con los de la función.
- 4.5. Una **lista de precondiciones**. Cada precondición es la referencia a un hecho, de entre los declarados, que debe ser cierto para poder aplicar el operador. Todos

¹ Deformación lingüística proveniente del inglés “instantiation”



los hechos, sin excepción, deben ser ciertos. Dicho de otro modo, no se admite la utilización de hechos negados (es decir, comprobar que un hecho no es cierto) ni conectivas diferentes de la conjunción (como la disyunción). Se entiende que para el propósito de este proyecto, un diseño sin estas últimas opciones es suficiente, pues introducirlas habría requerido el uso de recursos que se prefieren dedicar a otras funcionalidades consideradas de mayor interés. Por otro lado, las restricciones impuestas a las precondiciones son comunes en muchos sistemas de planificación actuales (el de F.E.A.R. [Orkin, 2006] es uno de ellos) que no por ello dejan de ser de utilidad.

4.6. Una **lista de precondiciones funcionales**. Cada precondición funcional es una operación cuyo resultado es lógico (sí/no). Tal y como sucede con las precondiciones no funcionales, la unión entre sus resultados es siempre mediante la conjunción. La información que se requiere de cada operación es la que sigue:

4.6.1. El identificador de operación, o **símbolo**. Para el propósito de este proyecto se consideran suficientes las operaciones lógicas de conjunción (AND) y disyunción (OR), las comparativas $>$, \leq , $=$, \geq y $<$, las aritméticas $+$, $-$, $*$, $/$ y las de asignación $=$, $+=$, $-=$, $*=$ y $/=$. Las de asignación del tipo de $+=$ se consideran de utilidad porque permiten unir dos operaciones en una, simplificando la representación. Como precondiciones funcionales, sólo tienen sentido las operaciones comparativas y lógicas. Obsérvese que estas últimas añaden, de hecho, la posibilidad de concatenar condiciones empleando una opción distinta a la conjunción. Cabría preguntarse por qué aquí sí se admite esto y no es así en la lista de precondiciones. La respuesta es que el enfoque del sistema en cuanto a aplicación eficiente de elementos se centra en los hechos y no en las variables, por lo que en el tratamiento de éstas puede añadirse este tipo de funcionalidad sin perjuicio de su rentabilidad en recursos de desarrollo, algo que no sucede con los hechos. Que sólo se permita la conjunción como conectiva en la lista se debe a que simplifica el modelo de representación, siendo además la más típica.

4.6.2. Una **lista de operandos**. Dos operandos se consideran suficientemente genéricos como para poder ser empleados en cualquier tipo de operación. Igualmente genérico habría sido un enfoque vectorial en el que un único operador pudiera aplicarse a un número indefinido de operandos pero, una



vez más, las complicaciones que añade no se consideran rentables para el propósito de este proyecto. Entendiendo que en una operación del tipo $A + B$, o $A += B$, A es el primer operando y B el segundo, los operandos pueden ser de tres tipos:

- Una referencia a una **variable**. Tiene sentido como primer o segundo operando en operaciones comparativas, aritméticas y de asignación.
- El valor de una **constante**. Tiene sentido como segundo operador en operaciones comparativas, aritméticas y de asignación. Las constantes deben ser números de 32 bits en coma flotante de precisión simple¹. Se considera que tal representación es suficiente para el propósito de este proyecto descartándose, al tiempo, la representación entera. Esta se considera de menor utilidad en videojuegos y no aglutina a los números reales, mientras que la coma flotante sí hace lo propio con los enteros, considerándose su menor precisión como algo de escasa repercusión en los resultados derivados de la utilización del sistema.
- El resultado de una **operación**. Tiene sentido como primer o segundo operador en operaciones lógicas si la operación es también lógica o comparativa, en comparativas si la operación es aritmética y en aritméticas si la operación es también aritmética. Tiene sentido como segundo operador en operaciones de asignación si la operación es aritmética.

4.7. Una **lista de eliminaciones**. De idéntica estructura que la lista de precondiciones, refleja los hechos que serán eliminados del estado actual como primer paso cuando se aplique el operador.

4.8. Una **lista de adiciones**. Incluye referencias a los hechos que serán añadidos al estado actual como segundo paso cuando se aplique el operador. Su estructura es idéntica a la de la lista de eliminaciones.

4.9. Una **lista de efectos funcionales**. Se trata de una lista de operaciones a ejecutar sobre el estado actual como tercer paso cuando se aplique el operador. Su estructura es idéntica a la de la lista de precondiciones funcionales, con la salvedad de que todas las operaciones que contenga han de ser de asignación puesto que son las únicas que tienen sentido como efecto.

¹ Estándar de representación numérica IEEE 754



En aras de la flexibilidad, se considera que un sistema de planificación orientado a videojuegos debe permitir un **coste potencialmente diferente por acción**. Una vía intermedia de solucionar este problema es otorgar a cada operador un coste fijo para todas las acciones que se deriven de él. Esa es la política, por ejemplo, del planificador de F.E.A.R. [Orkin, 2006]. La opción por la que se decanta el diseño de Aran es aún más flexible, puesto que implica los parámetros del operador en el cálculo del coste. Con esta solución se puede hacer que, por ejemplo, cueste más ir de un sitio a otro en función a la distancia que los separa y/o la velocidad del móvil o que la aplicación de una ley cueste más dependiendo del apoyo de la población. En definitiva, que esta vía añade un mayor dinamismo al sistema y favorece su reutilización. Una forma de aplicar esto sería incluir en cada operador una sección de coste, de forma que pudiera ser utilizada por el planificador para el cálculo del coste total del plan. Sin embargo se prefiere no hacer explícita esta sección, nuevamente, por razones de flexibilidad que serán explicadas cuando se hable de la métrica.

5. Una **lista de objetos**. La lista define los objetos que pueden ser utilizados para confeccionar un estado. Cada objeto ha de tener:
 - 5.1. Un **identificador** de objeto, unívoco en el ámbito del problema. Puesto que los únicos objetos de esta lista no interfieren con los de un operador (sus parámetros), se permite que sus identificadores puedan solaparse. Siempre que quiera hacerse una referencia a un objeto, se hará a través de su identificador.
 - 5.2. Una **relación de cumplimiento de plantilla** (*instanciación*) con un tipo.
6. Una **lista de hechos**. De idéntica estructura y misión que la definida dentro de un operador. Los identificadores deben ser unívocos en el ámbito del problema, lo que supone que pueden solaparse con los de los operadores por el mismo motivo que pueden hacerlo los de los objetos.
7. Una **lista de variables**. Nuevamente, de igual estructura y misión que la de los operadores, con identificadores de variable unívocos en el ámbito del problema.
8. Una **lista de hechos iniciales**. Hace referencia a los hechos de la lista que son ciertos en el estado inicial del problema. Basándose en la hipótesis del mundo cerrado, todos los hechos no incluidos en esta lista serán considerados falsos. La estructura de la lista es la misma que la de las adiciones de un operador.
9. Una **lista de inicializaciones funcionales**. Se trata de una lista de operaciones de asignación que tiene la misma estructura que la de efectos funcionales de un



operador. Su misión es la asignación de valores iniciales a las variables que, de otro modo, valdrán 0.

10. Una **lista de hechos meta**. Hace referencia a los hechos de la lista que deben ser ciertos en el estado meta del problema. La estructura de la lista es la misma que la de las precondiciones de un operador, incluidas las restricciones a la negación y la disyunción.
11. Una **lista de metas funcionales**. Se trata de una lista de operaciones de idéntica estructura que la de precondiciones funcionales de un operador. En este caso, todas las operaciones de la lista deben devolver un resultado cierto en el estado meta. Obsérvese que, dada la capacidad de anidación de operaciones, ello no significa que todas las operaciones anidadas deban dar ese resultado, sino sólo las que forman directamente la lista. Esto permite la elaboración de metas funcionales complejas, como lo permiten los operadores con sus precondiciones funcionales.
12. Una **métrica**. La métrica es la forma en la que se obtiene la calidad de un plan. Se compone de los siguientes elementos:
 - 12.1. Un **tipo** de métrica. La métrica puede ser de **minimización o maximización**, dependiendo de si considera mejor un plan que minimice o maximice el valor del objetivo.
 - 12.2. Un **objetivo**. Se trata de un **operando** cuyo valor se emplea en el cálculo de la calidad de un plan. Dicho valor ha de depender del estado que permite alcanzar el plan, por lo que carece aquí de sentido el empleo de constantes. Asimismo, sólo tienen sentido las operaciones aritméticas. Por tanto, el objetivo será un valor bien obtenido directamente de una variable o bien resultante de una operación aritmética.

La **riqueza expresiva** que permite la métrica hace innecesaria la definición expresa de una sección de coste en los operadores. Así, los operadores pueden modificar las variables que estimen oportunas, siendo el objetivo de la métrica el que defina cómo combinarlas. Obsérvese que la flexibilidad de esta solución permite definir una métrica consistente en minimizar la longitud del plan, pero también muchas otras. Esta flexibilidad es crucial para diversificar el tipo de videojuegos en los que poder usar este sistema. Por otro lado, el que sea la métrica quien decida cómo combinar las variables hace posible que, sin necesidad de modificar los operadores, pueda definirse un objetivo para cada problema, aumentando así las posibilidades de



reutilización. A pesar de estas ventajas, y teniendo en cuenta que la métrica clásica (minimización de la longitud del plan) es la más común, se ha decidido considerar la definición de una métrica como **opcional**, tomándose la clásica por omisión.

De los elementos comentados, pertenecen a la representación del **dominio** los cuatro primeros, esto es, tipos, predicados, funciones y operadores. El resto pertenecen a la representación del **problema**. Esta separación tiene una utilidad funcional de cara a la reutilización, pero en nada afecta al subsistema de representación pues a éste le interesa la información en su conjunto, independientemente de su origen o agrupación. Dicho de otro modo, se trata de una división lógica. Es por ello que se ha evitado hacer esta distinción durante las explicaciones.

4.3.3 Subsistema de representación específica

Ya se ha comentado que al sistema de planificación le es indiferente el método por el cual el usuario le transmite su conocimiento siempre y cuando éste le proporcione los elementos conceptuales que le son necesarios, de cuya definición se ocupa el subsistema de representación genérica. Sin embargo, es inevitable emplear alguna forma concreta de representación que el usuario pueda utilizar para construir esos conceptos. Ese es el cometido del subsistema de representación específica. **Proveer al sistema de un mecanismo concreto de representación del conocimiento que permita modelar los conceptos necesarios para llevar a cabo el proceso de razonamiento.**

El método de representación más extendido en planificación automática es el uso de un **lenguaje de texto**. No parece, sin embargo, que un sistema cuya interfaz de comunicación dependa del procesamiento de texto sea el escenario más eficiente posible y, desde luego, menos en un mundo, como el desarrollo de videojuegos, en el que la eficiencia suele ser un requisito poco menos que indispensable. Por otro lado, es extremadamente complicado construir un sistema que sea, a la par, eficiente y genérico. Si se busca lo primero a buen seguro se perderá parte de lo segundo, y viceversa. Por tanto, en videojuegos el escenario más común debería ser aquél en el que el subsistema de representación específica estuviera diseñado de forma específica para el juego en cuestión, integrándose con él al máximo pero, al tiempo, perdiendo toda posibilidad de ser genérico. Un paso más en esa búsqueda de la eficiencia en el uso de la planificación en videojuegos sería, incluso, prescindir del subsistema de traducción, pero eso es algo



que no se va a tratar por ahora. Así pues, fijando de nuevo la atención en qué subsistema de representación ha de ser el elegido parece que, a pesar de todo, la representación textual es la mejor posicionada. Recordando de nuevo su ineficiente capacidad de integración, hay que decir a su favor que en aquellos juegos en los que la eficiencia no sea una necesidad sí es posible emplear este tipo de representación. Por otro lado, permite cumplir otro de los objetivos del proyecto, que es tener algo funcional, que no tiene que ver con lo eficiente. Algo más a tener en cuenta a favor de la representación textual es que permite cumplir las necesidades del tipo de usuario que se identificó en la fase de análisis como investigador, más si se tiene en cuenta que en el campo de la investigación la representación textual es prácticamente un requisito. Por último, las definiciones del subsistema de representación genérica se ajustan sobremedida a los elementos de un lenguaje de texto estándar ya que, como se comentó, se basa en ideas comúnmente extendidas y tiene como principal fuente de inspiración el lenguaje textual PDDL¹. No significa esto, sin embargo, que el texto sea la única vía posible para representar los conceptos necesarios por el planificador, como se verá al final de este apartado.

Si el subsistema de representación genérica se ha basado en PDDL, cabe pensar que el de representación específica habría de ser un analizador sintáctico (*parser*) de tal lenguaje o un subconjunto del mismo. Al fin y al cabo, todos los elementos de representación tienen una traducción directa a PDDL. Sin embargo, se ha decidido que la **base sintáctica del lenguaje** desarrollado sea XML, el estándar de representación textual del momento y, se espera, del futuro. Se considera que, estando éste fuertemente presente en la plataforma de destino, .NET, favorecerá el aprendizaje de los potenciales usuarios. XML goza, además, de amplia aceptación entre la comunidad de desarrolladores, con independencia de la plataforma, lo que facilita la integración de Aran con otros sistemas existentes o futuros. Un ejemplo claro de beneficio en este sentido es la existencia de bibliotecas de tratamiento de este formato en múltiples lenguajes de programación, así como editores con ayudas para agilizar su escritura manual. Y es que el lenguaje es tan expresivo como lento de escribirse, al requerir de una serie de construcciones obligatorias que otros lenguajes no necesitan. Por otro lado, las mismas construcciones sintácticas que lo hacen lento de escribir lo hacen, en muchas

¹ Siglas de “Planning Domain Definition Language”



ocasiones, lento de leer, en parte por la tendencia al rápido crecimiento de los ficheros escritos en este lenguaje, no siempre proporcional a la cantidad de datos contenidos. En cualquier caso, y dada la capacidad del sistema construido para admitir otras representaciones, XML se considera una opción suficientemente buena como para ser la base del lenguaje a crear.

A continuación se comentarán algunas decisiones tomadas en relación al lenguaje construido para la representación de conocimiento en Aran, cuya gramática puede consultarse en el “Anexo A: Gramática del lenguaje”, página 217. Al respecto, sólo decir que, aunque lo común hoy día es que la gramática de un lenguaje basado en XML se defina mediante un esquema XML, se ha decidido emplear en esta memoria la notación EBNF¹ como mecanismo para simplificar su entendimiento y legibilidad. Ello ha requerido exponer una gramática más rígida de lo que en verdad es, no considerando los distintos elementos sintácticos propios de XML como tales, sino como literales. Debe entenderse que tal simplificación, que hace más inflexible el lenguaje, no tiene por qué ser seguida por un analizador sintáctico a rajatabla. De hecho, el desarrollado no lo hace. Sería una contrariedad que, habiendo tomado determinadas medidas que facilitan el uso del lenguaje se fuera, sin embargo, inflexible en algo tan sencillo de llevar a cabo.

Algo que no es una simplificación de la representación de la gramática en notación EBNF es la **no restricción del uso de los diferentes tipos de operaciones en las listas funcionales**. Por ejemplo, la gramática no recoge el que en una operación aritmética no pueda formar parte de una lista de precondiciones funcionales. Esta decisión se debe a una simplificación, sí, pero de cara a la reutilización. Y es que, en verdad, este tipo de restricciones pertenecen a la representación genérica, por lo que se considera más apropiado derivar su comprobación al subsistema de traducción. Esto deriva, además, en una simplificación del proceso de implementación de un nuevo subsistema de representación específica, algo que, como se ha comentado, se antoja probable de cara a un uso práctico del sistema.

Con el **número de operandos** que debe recibir cada operación se hace algo similar a lo explicado en el párrafo anterior, siendo el subsistema de traducción el encargado de comprobar que han de ser exactamente dos. Aquí, sin embargo, esta restricción sí se ha incluido en el esquema XML (no puede hacerse lo propio en EBNF),

¹ Siglas de “Extended Backus-Naur Form”



puesto que el coste de tal inclusión es ínfimo. Por otro lado, ese ínfimo coste permite tener una definición más rica de cara a la utilización de la gramática en herramientas de ayuda a la escritura que se basan en esquemas XML. Cabe decir que el esquema, aunque no sea recogido en esta memoria, sí ha sido desarrollado, entre otras cosas por simplificar la implementación de este subsistema, como se verá en el correspondiente apartado. **Otras restricciones en el número elementos** presentes de un determinado tipo también han sido incluidas en el esquema. En este caso porque, ahora sí, pertenecen al ámbito específico de representación y no al genérico. Claros ejemplos de esto son los elementos representados como opcionales en EBNF.

Hablando ahora de la filosofía general que ha inspirado el desarrollo de la gramática, ha de decirse que ésta ha sido definida con un claro objetivo en mente. Ante todo el lenguaje ha de ser **funcional y fácilmente implementable**, intentando evitar en la medida de lo posible que esto repercuta en su facilidad de uso pero sin que esto sea el principal referente de diseño. Dicho esto, podrá observarse que el lenguaje refleja fielmente la estructura conceptual requerida por la representación genérica. Por ejemplo, en él también es necesario declarar una lista de hechos o variables que, posteriormente, han de ser referenciados. Esta correspondencia directa está pensada para simplificar al máximo la traducción entre lo reconocido por el analizador sintáctico (*parser*) del lenguaje y la estructura de datos que define el subsistema de representación genérica lo que conlleva, claro, un menor esfuerzo de desarrollo. Si bien esta filosofía no repercute negativamente en determinadas construcciones, como el mencionado uso de hechos y variables, sí lo hace en otros, como la declaración de operandos. Las **listas de hechos y variables** evitan al usuario reescribir las tediosas construcciones sintácticas que representan un hecho cada vez que éste es reutilizado. Pero la facilidad de reconocimiento de los **operandos** que se buscaba con la gramática diseñada tiene el efecto de requerir un número quizá excesivo de anidamientos a poco compleja que sea la función a expresar. Este punto es, quizá, uno de los que peor nota podría llevarse en una evaluación de *usabilidad* del lenguaje y, sin embargo, se ha mantenido a conciencia en pro de ese objetivo principal marcado al comienzo: la consecución de funcionalidad a bajo coste. Se entiende, de nuevo, que la sustitución de este subsistema será algo común en el uso en videojuegos y, en cualquier caso, el sistema está preparado para recibir otros analizadores sintácticos. De ahí que no se le haya dado mayor atención a la resolución de un problema que no deja de estar presente. Se estudió como posibilidad el



empleo de una sintaxis parecida a la de MathML¹ pero su reconocimiento conllevaba dificultades que no se estimaron asumibles en relación al propósito del proyecto. Por último, la escritura y procesamiento de este tipo de estructuras tediosas no es impedimento para un sistema automático, mientras que la escritura y lectura por parte de humanos puede facilitarse empleando herramientas de desarrollo para XML.

Otro elemento que podría tildarse de tedioso en el lenguaje diseñado es el nombre de las **etiquetas** utilizadas, pues **ninguna** de ellas **emplea abreviaciones**. Es, en efecto, otro elemento que complica su escritura y utilización, pero que se ha mantenido así porque se ha preferido la legibilidad frente a la rapidez de escritura, apoyándose en lo ya expuesto acerca de que el uso de herramientas de edición de XML facilita estas tareas. Así, por ejemplo, se ha empleado la etiqueta “`predicate`” frente a su posible abreviación “`pred`” o, más claro aún, “**functionalPreconditions**” frente a “`funcp`”. Esto, por otro lado, es algo común en los lenguajes que se basan en XML y, de hecho, uno de los principales motivos de que los ficheros que emplean este formato tiendan a un rápido crecimiento y deban ser comprimidos. No será, por lo general, el caso de los ficheros de Aran, puesto que, comúnmente, el conocimiento acerca de un problema de planificación suele conllevar la representación de pocos conceptos y, además, los más habituales no requieren el uso de etiquetas de nombre largo. Quizá, de nuevo, la peor parte se la lleva la representación de **operaciones**, pero ya se ha dicho que su formato obedece a la facilidad de procesamiento que permite. A este respecto hay que añadir algo que no se comentó anteriormente, y es que las expresiones funcionales más habituales en un problema de planificación suelen ser las más sencillas, como el incremento o decremento del valor de una variable.

En otro orden de cosas, los nombres de etiquetas expuestos en el párrafo anterior ejemplifican la notación **lowerCamelCase**² que se ha empleado para las mismas. Este tipo de notación permite que la mayoría de las etiquetas se escriban completamente en minúsculas. Se ha hecho así para favorecer su escritura manual puesto que reduce las posibilidades de error, siendo XML un lenguaje sensible a mayúsculas. Esto puede resultar contradictorio dado que se han tomado otras decisiones que dificultan tal escritura. Pero, a diferencia de aquellas, esta decisión no implicaba esfuerzos adicionales inasumibles, por lo que, aquí sí, se ha preferido optar por el lado del usuario.

¹ Más información en <http://www.w3.org/Math>

² Más información en <http://es.wikipedia.org/wiki/CamelCase>



Volviendo a lo tedioso que puede parecer el lenguaje cara a su procesamiento manual, ya sea para ser escrito o leído, puede resultar llamativo el hecho de que la definición de **predicados y funciones** exija que sus **parámetros tengan** no sólo tipo sino **nombre**, cuando en la representación genérica no se exige identificador de parámetro (y los nombres actúan como tales en el lenguaje). Esta decisión es semántica. Una ayuda que se obliga al escritor a brindar para facilitar la labor del lector, puesto que dando nombre a los parámetros este último podrá comprender mejor el significado de cada uno. Esto tiene como consecuencia el que, en ocasiones, el nombre del parámetro sea el mismo que el del tipo, pero es algo que se asume en pro de esa labor semántica que desempeñarán en muchos otros casos.

Pasando ahora a las cadenas de caracteres y, por ende, a los **nombres** que el usuario puede dar a los distintos elementos, ha de decirse que se ha determinado no poner trabas a las mismas en ningún sentido, al contrario de lo que sucede en otros lenguajes como PDDL. Siendo texto libre, los identificadores pueden emplear cualquier carácter permitido en el tipo `string`¹ del esquema XML, lo que incluye, entre otros, espacios en blanco y tildes (si la codificación de la cabecera XML lo permite). La **única restricción** impuesta es **que no sean la cadena vacía**. Al respecto del **formato numérico**, no se ha impuesto tampoco ningún tipo de restricción que no venga ya determinada por el tipo `float`² del esquema XML.

Un sencillo apunte respecto de los **símbolos de las operaciones**. Sólo decir que son los mismos que los mostrados cuando se habló de ellos en la explicación del subsistema de representación genérica, con la salvedad de que `>` se sustituye por `gt` y `<` por `lt`. Esta sustitución es necesaria debido a que el símbolo `<` no está permitido en XML y debe emplearse su sustituto `<`. Para simplificar su uso se ha optado por eliminar el *ampersand*. Al tiempo, se ha optado por una sustitución similar con el símbolo `>`, por coherencia, ya que no era realmente necesaria

Se ha mencionado ya en varias ocasiones lo tedioso que puede resultar el procesamiento del lenguaje por parte de humanos. Las siguientes medidas pretenden suavizarlo. Por un lado, la mayoría de las **etiquetas anidadas son opcionales**, siempre y cuando no afecte a la coherencia del conjunto. Además, para estas etiquetas

¹ Más información en <http://www.w3.org/TR/xmlschema-2/#string>

² Más información en <http://www.w3.org/TR/xmlschema-2/#float>



opcionales se permite que el **orden de aparición pueda ser cualquiera**. Así, por ejemplo, dentro de un operador puede escribirse la sección de adiciones antes que la de hechos. Por otro lado, un tipo puede declararse antes que su tipo padre. Ambas medidas se sustentan en funcionalidad que ha sido desplazada al subsistema de traducción pero que, en cualquier caso, no tendría por qué haberse permitido en el subsistema de representación específica.

Un elemento importante en el diseño del lenguaje es la consideración en el mismo de la **representación de los planes**. En ocasiones los planificadores actuales emplean una representación del plan que poco o nada tiene que ver con la utilizada en la definición del dominio y el problema y que, incluso, tiene una vía de salida diferente (por ejemplo, entrada en ficheros y salida por pantalla). No será esa la vía que se siga en Aran, pues se considera de especial importancia mantener la coherencia del sistema en este sentido con el fin de facilitar la integración con otros. Es por ello que la gramática creada contempla tanto el formato de los ficheros de entrada (contenedores de dominios y problemas) como de los de salida (contenedores de planes).

El último, pero no menos importante, apunte con respecto al lenguaje diseñado corresponde al cometido de la **etiqueta raíz**, “*definition*”. Dicha etiqueta sirve de contenedor común a todas las definiciones (dominio, problema y plan) permitiendo que convivan distintas definiciones en un mismo fichero. Esto no tiene sentido en el caso del plan, pero sí cuando se habla de unir dominio y problema, algo que no habría sido posible si las etiquetas “*domain*” y “*problem*” hubieran sido diseñadas como cabeza de fichero. Aunque, en principio, el uso más común será el de separar ambas definiciones no se ha querido cerrar la puerta a una posibilidad que se antoja útil en ciertos casos y que no conlleva ningún esfuerzo adicional en el desarrollo.

Antes de acabar este apartado se retomará la afirmación inicial acerca de que la división de la representación en genérica y específica hace posible el **uso de sistemas que no requieran de un lenguaje de texto**. Aunque en este proyecto no se ha desarrollado ninguno por los motivos expuestos al comienzo, el mejor ejemplo es cualquiera que emplee un modelo actualizable cargado permanentemente en memoria. Con algo de este tipo, la carga del modelo puede hacerse por métodos diversos no necesariamente derivados de una lenta lectura de fichero como, por ejemplo, la codificación directa utilizando el lenguaje de programación usado en el desarrollo del



videojuego. Luego, también vía código, ese modelo podría ser modificado mediante métodos de acceso directo a sus estructuras de datos, siempre más rápidos que el procesamiento de texto. Finalmente, esos datos habrían de ser traducidos a la estructura requerida por el subsistema de representación genérica, algo que un buen diseño evitaría o dejaría en un esfuerzo mínimo. Ese modelo en memoria deriva en una representación actualizada con respecto al estado del videojuego mucho más eficiente en tiempo que cualquier tratamiento textual. Eso sin tener en cuenta que un diseño bien integrado podría hacer que dicho modelo fuera, directamente, el del mundo que el videojuego ha siempre de representar.

Retomando finalmente el lenguaje XML del que aquí se ha hablado, el “Anexo B: Ejemplos de utilización del lenguaje” contiene la definición de un dominio y un problema haciendo uso del mismo.

4.3.4 Subsistema núcleo

El núcleo representa la **base sobre la que se sustentan todos los demás conceptos** de un sistema informático. En la construcción de un sistema de planificación automática con búsqueda basada en estados, esto se traduce en la **definición de dos entidades, acción y estado**, así como de aquellos elementos que permitan una gestión básica de las mismas. En este sentido, se ha considerado básica la **selección de acciones y la planificación relajada** de las mismas. En este apartado se explican estos elementos a nivel conceptual, cuyo desarrollo se ha basado en el trabajo previo de terceros en el campo de la I+D+i. Por un lado, la siempre presente solución aplicada por Jeff Orkin en F.E.A.R. Por otro, las ideas en las que se basa el planificador desarrollado por Tomás de la Rosa para SAYPHI¹, en el seno del grupo PLG² de la Universidad Carlos III de Madrid. Por último, las explicaciones de Jörg Hoffmann y Malte Helmert en relación al desarrollo, respectivamente, de los planificadores Fast Forward (más conocido como FF) y Fast Downward.

4.3.4.1 Representación de estados

A nadie se le escapa que la representación que mejor encaja con un estado compuesto de hechos que pueden o no ser verdaderos y que, de hecho, sólo incluye a

¹ Siglas de “Sistema de Aprendizaje Y Planificación Heurística Informativa”

² Siglas de “Planning and Learning Group”



estos últimos es la de un conjunto. Por otro lado, el elevado número de operaciones relacionadas con el manejo de esos conjuntos desaconseja el uso de estructuras dinámicas. Por último, el número de estados que puede tener el mundo sobre el que planificar es finito y, por tanto, decantarse por su representación con **una estructura de datos estática** depende casi en exclusiva de cuánto espacio se esté dispuesto a perder a favor de la eficiencia porque, comúnmente, este tipo de estructuras tienen un mayor consumo en memoria debido la política de reserva a priori. En el desarrollo de videojuegos es habitual el empleo de estas de estructuras para aumentar la eficiencia en tiempo, que suele ser una de las prioritarias. Por otro lado, la vía estática goza de gran aceptación en planificación automática, habiéndose desarrollado distintas técnicas de codificación eficiente en espacio. Basándose en ello, se ha decidido emplear estructuras estáticas de representación en pro de la eficiencia, algo que se considera esencial a tan bajo nivel. **La flexibilidad habrán de aportarla las capas superiores**, enmendando como les sea posible las posibles carencias del núcleo en este sentido. Así, el subsistema de traducción deberá hacer posible la traducción de estructuras dinámicas de representación a las estáticas aquí requeridas tomando, por hacer un sencillo símil, una foto del estado de las mismas. Para ello el sistema presupone que el mundo es estático y permanece inalterado mientras él lleva a cabo el proceso de planificación. Esta presunción puede chocar con el dinamismo que caracteriza a los videojuegos, pero asume que una buena integración del núcleo con el videojuego debería minimizar las diferencias conceptuales entre esta solución y la realidad. Un buen ejemplo de esta asunción es el trabajo de integración llevado a cabo en un videojuego en tiempo real como F.E.A.R. [Orkin, 2006], inspirador de este proyecto. Y es que conseguir una eficiencia similar requerirá en último extremo trabajar directamente con el núcleo, para lo cual éste ha de prepararse deshaciéndose en lo posible de todo lastre a dicha eficiencia temporal, como lo son las estructuras dinámicas.

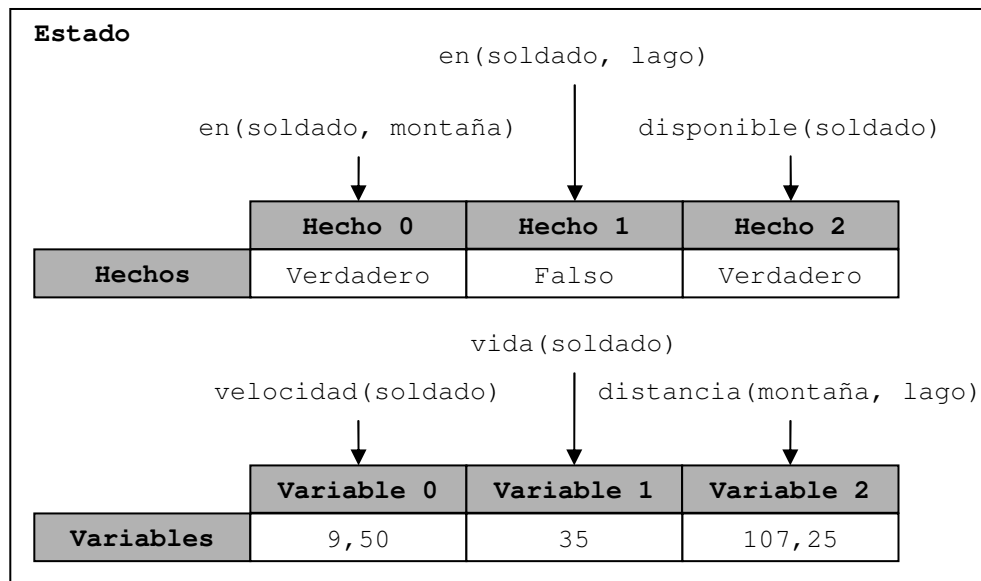


Ilustración 16: Representación conceptual de un estado

Siguiendo los principios ya comentados, los **estados** en Aran están representados por **dos listas estáticas de valores, unos booleanos y otros numéricos reales**. Los primeros representan la certeza de los hechos y los segundos los valores de las variables. La identificación de unos y otros se hará por posición, derivando la gestión de la correspondencia a entidades externas al núcleo, pues a éste le es indiferente. Dicha entidad se encargará de determinar qué hecho corresponde a qué posición en la lista de hechos y qué variable corresponde a qué posición en la lista de variables. Esta representación es la empleada por algunos planificadores, como el de SAYPHI y es la elegida porque permite un bajo consumo de memoria a la par que una sencilla gestión de los cambios en el estado. En otros planificadores, como el de F.E.A.R., emplean otros tipos de datos, como pueden ser los enumerados¹. Así evitan la explosión de variables booleanas que puede darse en casos como el del posicionamiento, en el que un elemento sólo puede estar en una determinada localización al mismo tiempo de entre las N posibles. La representación elegida derivará en N variables booleanas de las que sólo una tendrá valor verdadero al mismo tiempo. Ese otro tipo de representación emplea una única variable numérica que puede tomar cualquiera de esos N valores. Se estudió la posibilidad de implementar el algoritmo propuesto en [Edelkamp & Helmert, 1999] para la conversión de este tipo de hechos en variables enumeradas, pero requiere de un análisis del conocimiento aportado por el usuario que se estimó lo suficientemente complejo como para no ser llevado a cabo, dado que la representación de los estados es

¹ Números enteros que sólo pueden tomar unos ciertos valores predeterminados



sólo una pequeña parte de todo el proyecto y no se consideró rentable invertir el esfuerzo necesario. A esta decisión ayudó el hecho de que una implementación eficiente de las listas estáticas booleanas reduce el impacto de la representación elegida en el consumo de memoria. Volviendo a dicha representación, quizá choque ver que son representados los hechos falsos. Ello es así porque ha de preverse la posibilidad de que el hecho cambie a verdadero en un momento dado y, siguiendo la política de reserva a priori de memoria, ha de hacersele un hueco en el estado para contemplar ese caso. En [Edelkamp & Helmert, 1999] proponen algunas optimizaciones en este sentido que sí se siguieron, aunque derivando esas tareas al subsistema de traducción.

Comentada la representación de hechos y variables, a continuación se lista la **estructura completa de un estado**:

1. La lista de certeza de **hechos**. La longitud de la lista es idéntica para todo estado de un mismo problema.
2. La lista de valores de **variables**. Al igual que sucede con los hechos, la longitud de la lista es idéntica para todo estado de un mismo problema.
3. Una referencia al **estado padre**. El estado padre es aquél del que otro deriva. El resultado de la planificación será el estado final, a partir del cual podrá obtenerse el camino, por medio de estos enlaces, hasta el estado inicial.
4. Una referencia a la **acción aplicada**. Se refiere a aquella que se aplicó al estado padre para obtener el actual. Recorriendo la sucesión de estados desde la meta hasta el inicial puede obtenerse, mediante estos enlaces, la lista invertida de las acciones aplicadas. Invertiendo dicha lista se obtienen las acciones a aplicar a partir del estado inicial para alcanzar la meta.
5. La **longitud del plan** hasta este estado. En efecto, la representación de un estado sirve al núcleo, al mismo tiempo, de nodo en la cadena de acciones que representa el plan. Guardar la longitud del plan hasta un cierto nodo permite, durante la inversión de la lista de acciones, reservar memoria a priori en una estructura estática que almacene el plan ordenado. Además, permite utilizar dicha longitud de plan cuando la heurística empleada sea esa, evitando seguir los enlaces hasta el estado inicial



para calcularla. Téngase en cuenta que en esos casos la obtención de esta longitud será una operación muy frecuente y ha de evitarse que sea, además, costosa.

Quizá la estructura de un estado no sea todo lo pura que pudiera, en tanto en cuanto todo aquello que no haga referencia a los valores que identifican al estado propiamente dicho puede sacarse a estructuras auxiliares. Si se han decidido incluir como parte del estado es haciendo honor a la cohesión que los elementos de cada subsistema han de presentar en pro del mejor funcionamiento posible de sus interiores. Teniendo los estados referencias directas a otros elementos se favorece la eficiencia del conjunto.

Un último apunte acerca de los estados es que éstos necesitarán ser comparados para determinar su **igualdad** o no con otros durante el proceso de búsqueda de la solución. En la definición de estado que emplea Aran, es posible que sólo algunas variables sean relevantes para distinguir dos estados mientras que otras simplemente cumplan una función de almacenamiento de valor, no aportando información a tal comparación y, de hecho, haciendo que ésta devuelva resultados no deseados si son tenidas en cuenta, pudiendo derivar en bucles infinitos durante la búsqueda. Así pues, el proceso de comparación requerirá determinar previamente qué **variables** del estado son **relevantes** y cuáles no, pero eso, una vez más, no es tarea del núcleo.

4.3.4.2 Representación de acciones

El núcleo sólo trabaja con acciones, no existiendo estructura alguna que represente explícitamente un operador. Derivado del diseño empleado en SAYPHI, se ha dispuesto que cada acción contenga **tres listas estáticas, una para las precondiciones, otra para las adiciones y otra para las eliminaciones, que guardan valores numéricos enteros que hacen referencia a la posición de un hecho en el estado**. De este modo, si, por ejemplo, se quiere representar que el hecho 2, el 7 y el 25 son precondiciones de una acción, ésta guardará dichos números en la lista de precondiciones. Comprobar que un estado cumple las precondiciones se traduce en tomar uno a uno los números contenidos en la lista de precondiciones e ir verificando que, efectivamente, el hecho identificado por ese número en el estado es cierto. En el ejemplo mencionado, supondría comprobar que los valores de las posiciones 2, 7 y 25 de la lista de hechos en el estado son verdaderos. Lo mismo para las adiciones y



eliminaciones, pero en este caso en lugar de una verificación se produce una modificación del valor del hecho, pasando de falso a verdadero si es añadido y al contrario si es eliminado. Este tipo de representación permite que, al contrario de lo que sucede con todos los estados, **cada acción pueda tener listas de longitudes diferentes**, evitando así la reserva indiscriminada de memoria que lo contrario supondría. Ahora bien, **acciones derivadas del mismo operador tendrán idéntica estructura**, que no valores, por lo que las listas de éstas sí tendrán idénticas longitudes.

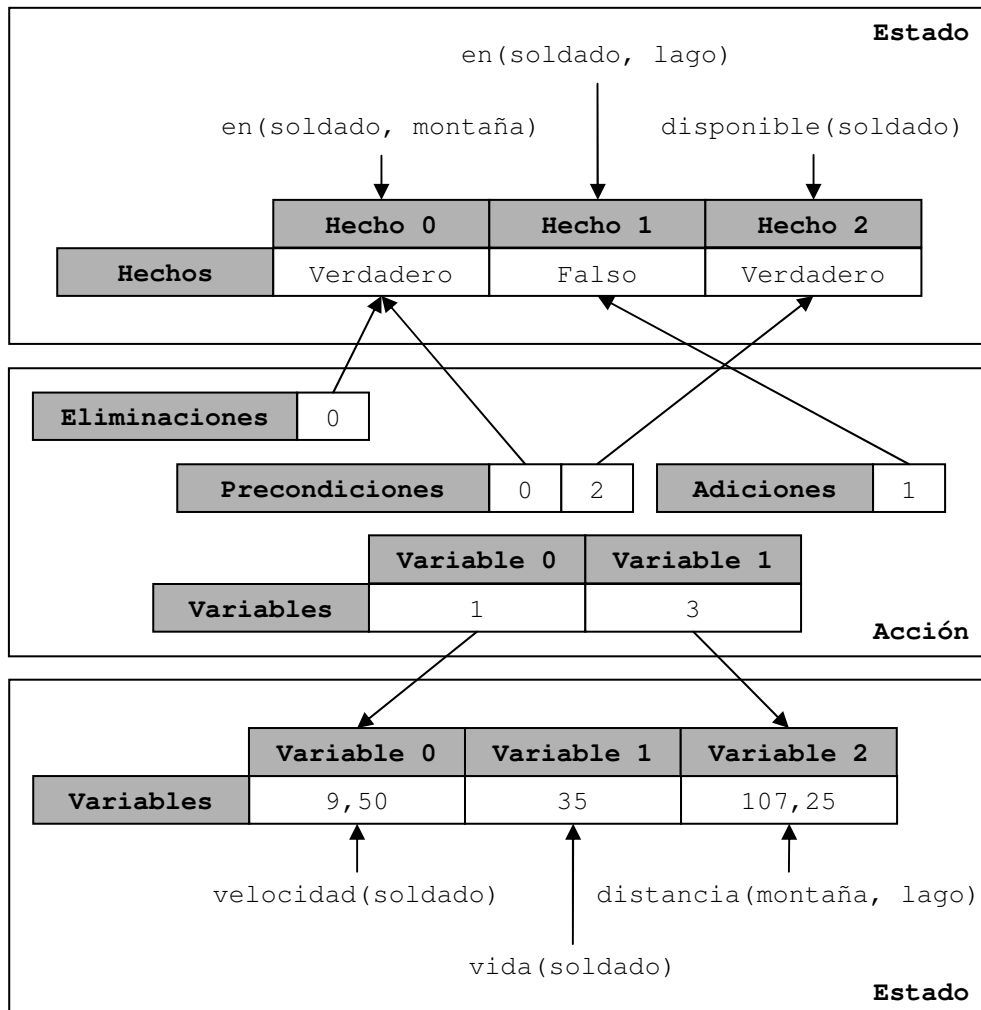


Ilustración 17: Representación conceptual de una acción

Las **variables que emplea una acción son representadas de idéntica forma** a como se hace con precondiciones, eliminaciones y adiciones, con la diferencia de que, obviamente, los valores de la lista estática (porque aquí sólo hace falta una) **apuntan a posiciones de la lista de variables del estado**, no a la de hechos. En este caso el cometido de tales punteros es diferente, pues **sirven de redirección de variables de cara al uso de las operaciones**, apoyándose en lo dicho anteriormente acerca de



igualdad de estructura entre acciones del mismo operador. Si se observa, las operaciones que contiene un operador son parte de su estructura y, por tanto, algo que debe ser compartido por todas sus acciones. Esas operaciones manejarán siempre un número determinado de variables, idéntico para toda acción del mismo operador. Así pues, es posible reutilizar la estructura de dichas operaciones haciendo que busquen las variables a las que afectan en alguna otra estructura, esta vez sí, propia de cada acción, pues cada acción hará uso de unas ciertas variables, potencialmente diferentes a las de sus “hermanas”, por mucho que comparta con ellas el número total de variables implicadas. Imagínese, por ejemplo, que las operaciones de un operador emplean siempre 2 variables. Ahora piénsese en 2 acciones diferentes de ese mismo operador. Una de ellas empleará las variables 4 y 10 de la lista estática del estado, mientras que la otra utilizará las variables 3 y 7. Bien, pues esos números son los que habrán de almacenarse en la lista estática de variables de la acción, para hacer corresponder, en el primer caso, la variable 0 de la acción con la 4 del estado y la 1 de la acción con la 10 del estado. En el segundo caso la variable 0 de la acción se corresponderá con la 3 del estado y la 1 de la acción con la 7 del estado. Como puede verse, y ya se ha dicho, el núcleo trabaja sólo con identificadores numéricos, no importándole en absoluto si la variable 3 del estado es, por decir algo, “resistencia(muro)” o “munición(pistola)”. De esa otra correspondencia se ocupará el subsistema de traducción permitiendo al núcleo aumentar su eficiencia.

El principal problema con el que ha de lidiarse al respecto del modelado de las **operaciones** es que éstas pueden ser **muchas y muy diversas**. Podría establecerse algún modelo, también estático, que recogiese los elementos comunes de tal diversidad dando como resultado, seguramente, algo similar a lo definido en el subsistema de representación genérica. Un modelo como ese permitiría al núcleo conocer el tipo de sus operaciones y los operandos de éstas. Ahora bien, algo que le es necesario al subsistema de traducción no parece serlo para el núcleo si éste no trabaja nunca con esa información, como es el caso. Y es que el uso principal para el que están pensadas las operaciones en este sistema es la modificación de valores que tomarán parte en la evaluación de la métrica, permitiendo así una flexibilidad en su definición imposible de otro modo. Si se emplea, además, en el cálculo de precondiciones funcionales, por ejemplo, es sólo porque, en definitiva, la inclusión de esta funcionalidad es casi directa. No quiere decir ello que sea funcionalidad principal y, por tanto, **no se estima**



oportuno establecer mecanismos de manejo de metainformación acerca de la estructura de dichas operaciones pues, en esencia, sería sólo necesaria en el caso de que quisiera dársele un tratamiento pormenorizado de cara a un mejor funcionamiento de la funcionalidad que proporcionan. Esta decisión trae una serie de consecuencias que merman la funcionalidad de esas precondiciones funcionales, como se indicará más adelante. Esta merma es conocida y asumida, recordando una vez más que el propósito de este proyecto es la consecución de un sistema de planificación centrado principalmente en el manejo de hechos y la gestión de la métrica.

Expuestas las razones por las cuales no se elaborará un modelo pormenorizado sino minimalista de **representación de operaciones**, a continuación se identifican los elementos comunes a todas ellas. Algo que al subsistema de representación genérica no le interesaba pero que aquí es de vital importancia es la ejecución de las operaciones, y sobre la base de ello se elaborarán distintas interfaces dependiendo del tipo de operación a representar. Así, en función del resultado que devuelven, éstas se han clasificado en:

- **Condición.** Su ejecución **no modifica** el estado y su resultado es un **valor booleano**. El núcleo empleará este tipo de operaciones para representar las precondiciones funcionales. Requiere conocer el resultado de la evaluación de la condición para poder así acumularlo al del resto de precondiciones. Que se les denomine condición y no precondición se debe a que el primer término es más genérico, previendo posibles usos de estas condiciones fuera de la representación de las acciones.
- **Efecto.** Su ejecución **modifica** el estado y no devuelve **ningún valor**. El núcleo empleará este tipo de operaciones para representar los efectos funcionales. Él no trata directamente con el resultado de tales efectos, limitándose a aplicarlos, de ahí que no se le devuelva ningún resultado.
- **Función.** Su ejecución **no modifica** el estado y su resultado es un **valor numérico**. El núcleo no empleará directamente este tipo de operaciones, pero servirán para representar aquellas que devuelven resultados numéricos y que, eventualmente, podrán formar parte de alguna condición o efecto compuesto.

Las operaciones **han de aplicarse siempre sobre un estado** y, por tanto, dependen de la estructura del mismo. Por otro lado, aquellas que deban hacer uso de



variables dependerán también de la estructura de la acción que las contiene. Sin embargo, y con el fin de compartir la estructura de las operaciones entre acciones, éstas no pueden apuntar directamente a ningún estado o acción, por lo que habrán de recibir ambos cada vez que sean invocadas. A este tipo de operaciones se les ha denominado **dependientes de acción**. Y es que, aprovechando la estructura ya definida para las operaciones, puede emplearse una variación de las mismas para utilizar condiciones, efectos y funciones fuera de la estructura de una acción, no dependiendo, por tanto, de ésta. Este tipo de operaciones **independientes de acción** no requieren, en consecuencia, recibir acción alguna como parte de sus parámetros de invocación, conservando, eso sí, el estado al que deben aplicarse. Este último tipo de operaciones son útiles en la definición de las inicializaciones y metas funcionales, pues aquí no es innecesario emplear ningún tipo de redirección de variables. Se manejan variables, sí, pero las operaciones no han de ser reutilizadas en la estructura de varias acciones, por lo que pueden permitirse hacer una referencia directa y, por tanto, más eficiente, a las variables del estado.

	Dependiente de acción	Independiente de acción
Condición	Entrada: estado, acción Salida: sí/no	Entrada: estado Salida: sí/no
Efecto	Entrada: estado, acción Salida: nada	Entrada: estado Salida: nada
Función	Entrada: estado, acción Salida: número	Entrada: estado Salida: número

Tabla 2: Tipos de representación de operaciones

La libertad que ofrece la definición que el núcleo hace de una operación abre, adrede, muchas posibilidades de utilización. En este proyecto sólo se acometerá el desarrollo de operaciones numéricas derivadas, además, al subsistema de extensión funcional. Sin embargo, bien podrían desarrollarse lo que [Orkin, 2004] denomina **precondiciones y efectos contextuales**. Se trata en ambos casos de piezas de código que pueden ser ejecutadas durante la ejecución del planificador y que le ayudan a tomar decisiones dependiendo de qué esté sucediendo en el mundo mientras él planifica. No supone esto un abandono de la presunción de estaticidad del mundo, no al menos completa. La condición que ponen en F.E.A.R. al uso de este tipo de elementos



contextuales es que deben manejar conceptos que no conciernen al planificador. Por ejemplo, una precondition contextual sería verificar que existe espacio suficiente en la habitación para ejecutar la animación asociada a una acción.

Por utilidad, el núcleo desarrolla una única función, independiente de acción. El valor que devuelve es el guardado por el estado indicando la longitud del plan. Así pues, ésta podría decirse que es la **función de obtención de longitud de plan**.

Explicados los conceptos fundamentales, a continuación se expone la **estructura completa de una acción**:

1. **Identificador** de acción. Un número identificará unívocamente cada acción. La misión de este número es similar a la que tiene la posición que ocupa un hecho en un estado. Nuevamente será misión de una entidad externa buscar una forma de asociar este identificador numérico a la representación específica de la acción. Por ejemplo, deberá saber que la acción 37 se corresponde con “`encender(candil, mechero)`”.
2. La lista de referencias a hechos que son **precondiciones**. Como ya se ha dicho, la longitud de la lista es idéntica para todas las acciones del mismo operador.
3. Una referencia a una **precondición** funcional. Se trata de una condición dependiente de acción. Es una sola y no varias porque se estima más potente la utilización de condiciones compuestas cuando haya de verificarse más de una al mismo tiempo. La condición referenciada es la misma para todas las acciones del mismo operador.
4. La lista de referencias a hechos que son **eliminaciones**. Nuevamente, la longitud de la lista es idéntica para todas las acciones del mismo operador.
5. La lista de referencias a hechos que son **adiciones**. Otra vez, la longitud de la lista es idéntica para todas las acciones del mismo operador.
6. La lista de redirecciones de **variables** de acción a variables de estado. Como el resto, la longitud de la lista es idéntica para todas las acciones del mismo operador.



7. Una referencia a un **efecto** funcional. Se trata de un efecto dependiente de acción. Por el mismo motivo que la precondition, es uno solo y no varios. Igualmente, el efecto referenciado es el mismo para todas las acciones del mismo operador.

Pueden, además, identificarse dos tipos de **operaciones relacionadas con el manejo de acciones** por parte del núcleo:

- **Comprobar si la acción es aplicable a un estado.** Esta operación consiste en verificar que se cumplen, en este orden, las preconditiones que hacen referencia a la certeza de hechos y la precondition funcional. Dada la variedad y potencial complejidad que puede tener una precondition funcional, ésta será verificada sólo si todos los hechos requeridos como ciertos lo son. Con ello se persigue reducir la ejecución de operaciones en un intento por mejorar la eficiencia.
- **Aplicar la acción a un estado.** Puesto que la búsqueda sobre un espacio de estados conlleva la posible existencia de múltiples estados al mismo tiempo, es preciso que la aplicación de una acción derive en la creación de un nuevo estado modificado, y no en la modificación del existente. Así, el primer paso será copiar el estado padre para crear el nuevo. El nuevo estado guardará la referencia al padre del que proviene. Seguidamente, se cambiará el valor de los hechos referenciados en la lista de eliminaciones a falso. A continuación se modificará el valor de los hechos referenciados por la lista de adiciones a verdadero. Se da, por tanto, preferencia a las adiciones frente a las eliminaciones en caso de hipotético conflicto, si bien la entidad externa encargada de la creación de la acción debería evitar este tipo de situaciones. Finalmente, se aplicará el efecto funcional, que modificará el estado dependiendo de la función que encapsule.

4.3.4.3 Selección de acciones

El proceso de selección de acciones consiste en determinar **qué acciones de entre todas las existentes son aplicables a un estado dado**. Este proceso podría ser tan simple como recorrer todas las acciones ejecutando para cada una de ellas la operación de comprobación de aplicabilidad explicada con anterioridad. Pero esa solución sería tan simple como ineficiente. Se precisa, sin embargo, de un método que tome en consideración todas las acciones, sin salvedad. No puede obviarse ninguna de las existentes porque, en ese caso, no se tendría la certeza de estar seleccionando todas las



acciones aplicables al estado. La solución empleada, pasa por aprovechar ese concepto de totalidad para dividir el proceso en dos operaciones, una global a toda operación y otra concreta a cada una de ellas. Así, en un primer lugar se seleccionarán todas aquellas acciones para las cuales el estado tenga como verdaderos todos los hechos que requieren. En un segundo lugar, se recorrerán todas esas acciones (una lista normalmente mucho menor que la original) verificando para cada una de ellas si se cumple su precondition funcional. El segundo paso no requiere, por simple, comentario alguno. A continuación se explicará cómo se lleva a cabo el primero de ellos, el cual se basa en la forma en la que lo hace SAYPHI.

Optimizar el primer paso del proceso de selección pasa por crear una estructura de datos que contemple, al mismo tiempo, todas las condiciones de todas las acciones. Se trata de disponer esos datos de forma que se pueda aplicar sobre ellos operaciones que podrían calificarse de vectoriales. El beneficio de este tipo de operaciones, que trabajan a la vez con un buen número de datos, es bien conocido, más aún cuando existe la posibilidad de ejecutar varias en paralelo. Ha de recordarse que una máquina como la Xbox 360 dispone de 6 *threads hardware* (aunque, en verdad, XNA sólo permite hacer uso de 4 de ellos).

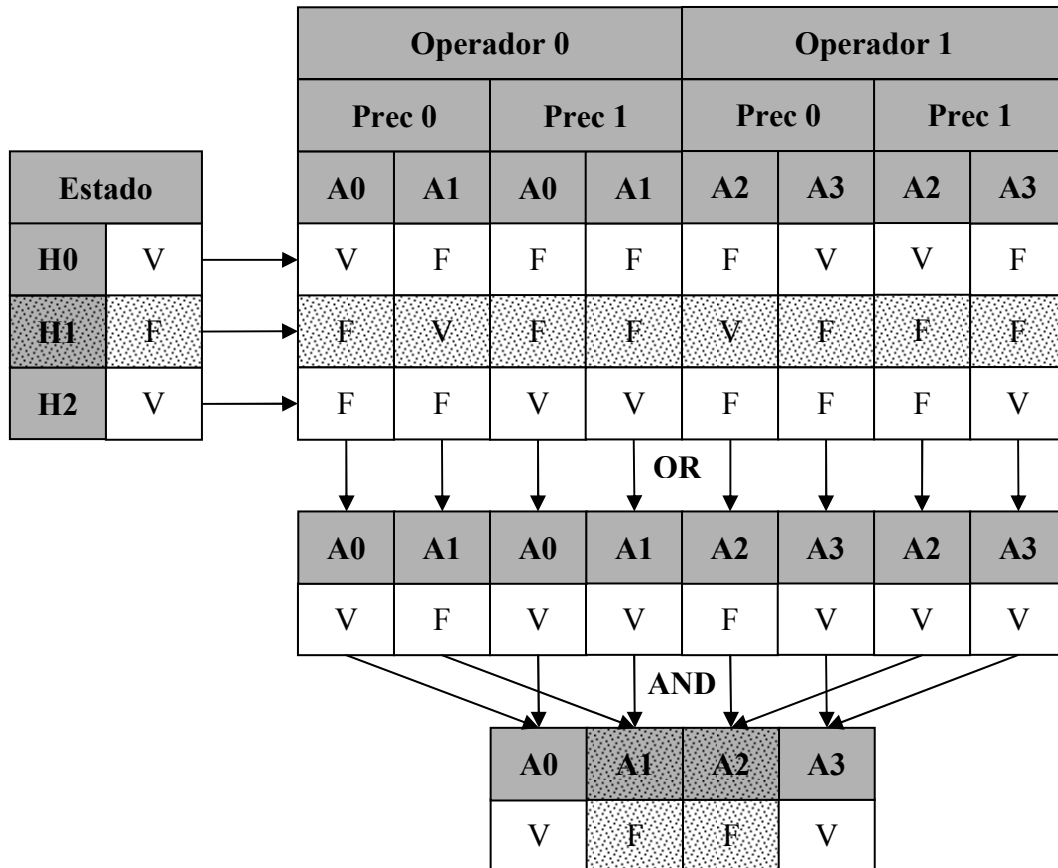


Ilustración 18: Primer paso de la selección de acciones

La Ilustración 18 muestra un modelo conceptual de la estructura de datos, en forma de tabla. Dicha tabla tiene tantas filas como hechos puede albergar el estado. En el ejemplo se dispone de un estado con 3 posibles hechos, H0, H1 y H2. En cuanto a las columnas, tiene tantas como operadores. Cada columna de operador se subdivide, a su vez, en tantas columnas como precondiciones tiene ese operador. Cada columna de precondición vuelve a subdividirse en tantas columnas como acciones se derivan de ese operador. En el ejemplo se tienen 2 operadores, cada uno de ellos con 2 precondiciones y 2 acciones. Esto es, obviamente, una simplificación, puesto que cada operador puede tener un número distinto de precondiciones y, claro, de acciones. En última instancia, las columnas representan la activación de las acciones cuando un hecho del estado es verdadero. De ahí que a esta estructura se le haya decidido llamar **tabla de activación de acciones**. Las acciones, como se ha comentado en su momento, disponen de un identificador unívoco, de ahí que las del ejemplo se llamen A0, A1, A2 y A3. Las precondiciones sólo requieren ser reconocidas a nivel de operador, de ahí que sus identificadores sólo sean unívocos a ese nivel. En definitiva, el valor de cada celda de la tabla será verdadero (V) o falso (F), dependiendo de si la acción en cuestión tiene ese



hecho como precondition o no. Por ejemplo, el hecho H0 es la primera precondition de las acciones A0 y A3 y la segunda precondition de A2. Nótese el sinsentido de que dos hechos sean la misma precondition de una acción. Es por ello que para una misma columna no puede haber más de un valor verdadero. Por ejemplo, en la columna correspondiente a la primera precondition de A0 sólo la celda correspondiente a la fila de H0 es verdadera. Por último, se habrá observado que la tabla sólo pretende recoger las activaciones positivas, es decir, qué acciones se activan con los hechos verdaderos. En este sentido, conviene recordar que en el sistema de planificación construido sólo se permite el uso de precondiciones positivas.

La Ilustración 18 también recoge **cómo se utiliza la tabla de activación de acciones** para decidir qué acciones se activan con los hechos verdaderos que presenta el estado. Así, en un primer paso, las **filas que no se correspondan con hechos verdaderos en el estado son anuladas**. Estas filas no serán tomadas en cuenta en próximos pasos. En el ejemplo se anulan la fila correspondiente a H1, puesto que tal hecho es falso. El segundo paso es **aplanar la tabla** de modo que sólo quede una fila en la que cada celda tenga valor verdadero si en la columna de la que proviene había algún valor verdadero, esto es, si la precondition a la que se corresponde la celda es cierta para la acción a la que se refiere. Volviendo una vez más al ejemplo, la primera celda por la izquierda, correspondiente a la primera precondition de A0, tiene valor verdadero porque la fila de H0 de la columna de la que proviene tenía valor verdadero. Sin embargo, la celda correspondiente a la precondition 0 de A1 tiene valor falso, puesto que la fila que tenía valor verdadero era la de H1, y ésta fue anulada previamente. En definitiva, se trata de hacer una **operación OR con los valores de la columna**. La fila resultante permite conocer para cada acción qué precondiciones se activan en ese estado. En el ejemplo, para el estado dado se activan todas las precondiciones salvo las primeras de A1 y A2. Sabiendo que el sistema construido requiere que todas las precondiciones de una acción se cumplan para poder aplicarla, eso significa que A1 y A2 no podrán ser aplicadas. Generalizando, el tercer paso consiste en **reducir la fila** de modo que sólo exista una columna por acción. Dicha columna, celda realmente, tendrá valor verdadero si todas las columnas de la fila de la que provienen tenían también valor verdadero. En otras palabras, ha de realizarse una **operación AND con los valores de las columnas del mismo nombre**, depositando el resultado en la única existente en la nueva fila. Si se observa, el resultado es **una lista de activación**, que indica qué



acciones ven cumplidas todas sus precondiciones en ese estado y, por tanto, son candidatas a ser aplicadas, siempre que se cumpla su precondición funcional. En el ejemplo, las acciones activas son A0 y A3.

4.3.4.4 Planificación relajada

La meta del sistema de planificación construido es devolver planes en los cuales se cumplan todas y cada una de las restricciones impuestas por el usuario. Básicamente, dichas restricciones son la búsqueda de un estado objetivo, la minimización o maximización de una métrica y la correcta ordenación de las acciones dentro del plan. Para esto último, el planificador ha de tener siempre muy presentes las precondiciones y los efectos de las acciones, de modo que el plan asegure que en el momento de ejecutar una acción se cumplen sus precondiciones, bien porque eran ciertas en el estado inicial o bien porque el efecto de alguna acción anterior las hizo ciertas, y siempre que ninguna acción anterior las haya hecho falsas después. La planificación relajada **consiste en suavizar, relajar, algunas de esas restricciones con el fin de hacer menos costoso el proceso de planificación a costa, claro, de que el plan resultante no cumpla las restricciones que no se tuvieron en cuenta, que se relajaron**. Este tipo de planificación ha sido empleado en diversos planificadores para poder calcular una heurística independiente del dominio. En Aran se ha decidido emplear esta misma técnica, utilizando el cálculo de planes relajados que emplea el planificador FF [Hoffmann & Nebel, 2001]. No es objetivo de esta memoria explicar en profundidad este proceso, pues puede encontrarse información detallada en la bibliografía. Sólo se dará, por tanto, una noción intuitiva del funcionamiento de la planificación relajada de FF a fin de que puedan entenderse próximos apartados que se basan en ello. Decir, por último, que su inclusión en el núcleo se debe a que **se entiende que el uso de un planificador relajado puede ir más allá del cálculo de la heurística**. Si bien en este proyecto no se emplea para nada diferente, se estima que podría hacerse en un futuro. Un posible uso podría ser emplearla como estimador de planes, sin que ello implique necesariamente su participación en proceso de búsqueda alguno.

El proceso de planificación relajada parte de la definición de acciones relajadas, en las que no se tiene en cuenta parte de la definición original de la acción o se tiene en cuenta de un modo aproximado. FF es un planificador sin manejo numérico que, por tanto, no permite el uso de precondiciones ni efectos funcionales. Es por ello que en



ningún momento menciona la relajación de tales elementos. Sí hace lo propio con los efectos de eliminación, los cuales ignora por completo. Así, durante la planificación relajada sólo puede aumentar el número de hechos verdaderos en el estado y, por tanto, nunca una acción se verá afectada porque otra elimine los hechos de los que ella depende. Se tiene, por tanto, que una acción relajada es una simplificación de la original en la que sólo se tiene en cuenta la lista de precondiciones y la de adiciones.

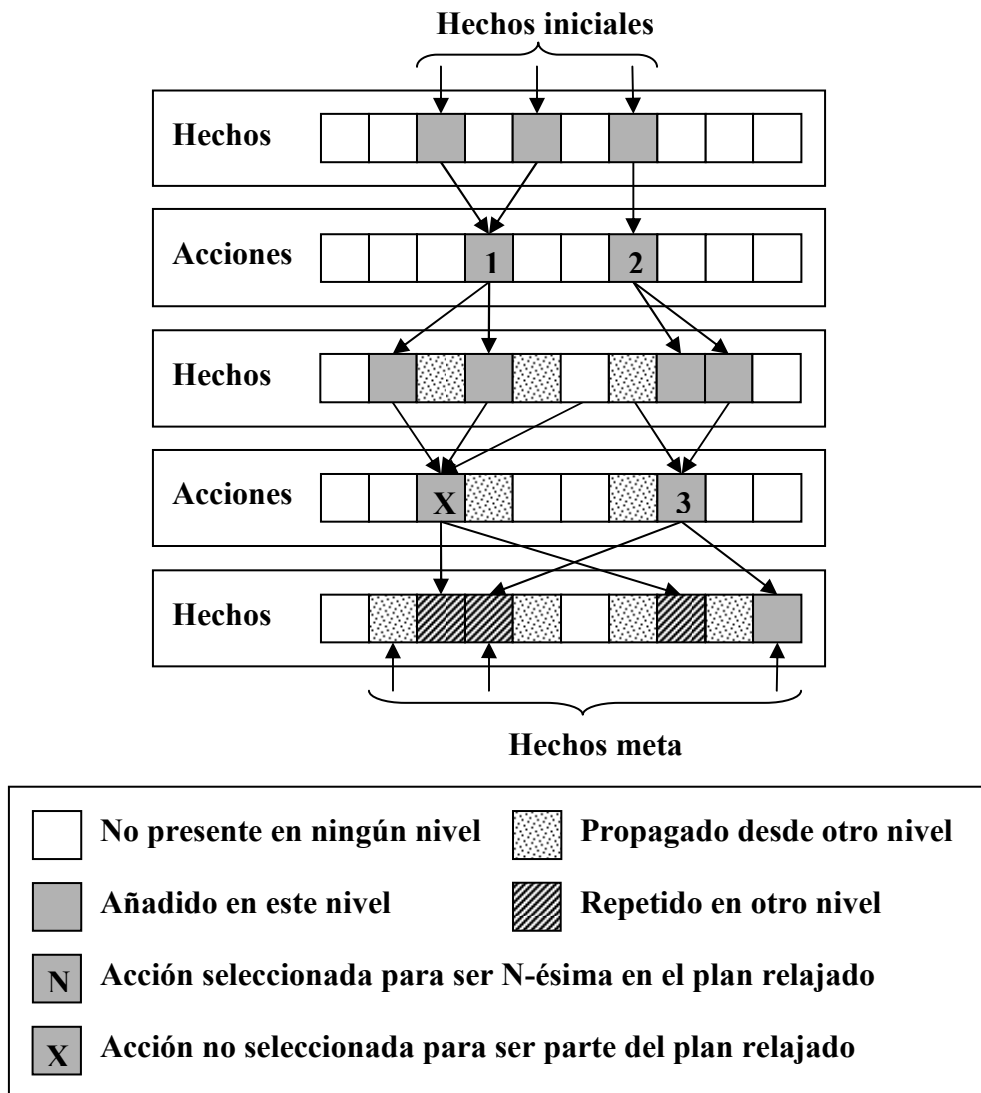


Ilustración 19: Grafo de planificación relajada

Empleando siempre acciones relajadas, FF ejecuta un proceso iterativo de **construcción de un grafo de planificación** que intercala niveles de hechos y niveles de acciones de modo que los hechos de un nivel N enlazan con acciones del nivel $N + 1$ y las acciones del nivel $N + 1$ enlazan con hechos del nivel $N + 2$. Las acciones presentes en el nivel $N + 1$ son aquellas aplicables a partir de los hechos presentes en el nivel N o



anteriores. Los hechos presentes en el nivel $N + 2$ son los añadidos por acciones del nivel $N + 1$. El nivel inicial de hechos está compuesto por los presentes en el estado inicial. El último nivel de hechos es aquel en el cual están contenidos los que, junto con los ya aparecidos en niveles anteriores, conforman la meta. Resumiendo, se trata de ir aplicando una acción tras otra partiendo del estado inicial, de modo que existan cada vez más hechos verdaderos en el estado. En el momento en que un subconjunto de esos hechos verdaderos sean los presentes en el estado meta, la construcción del grafo concluye.

Con el grafo de planificación construido, el segundo paso es **reconstruir el plan relajado**, es decir, conseguir una lista de acciones relajadas que, aplicadas unas tras otras, permitan llegar al estado relajado (aquél que sólo puede crecer en número de hechos verdaderos) que contenga los hechos del estado meta. Para ello, se siguen hacia atrás los enlaces que hicieron posible que un hecho meta fuera cierto, seleccionando las acciones a las que llevan dichos enlaces. Si un hecho fue añadido por más de una acción, se prefiere aquella de menor dificultad. La dificultad de una acción es la suma del número de nivel en el que aparecieron cada uno de los hechos que requiere como precondition. Cuanto más cercanos, pues, esos hechos al nivel inicial, menos dificultad tendrá esa acción. Las acciones de un mismo nivel son incluidas en el plan en el mismo orden en el que son seleccionadas.

4.3.5 Subsistema de extensión funcional

El núcleo de Aran sólo define las interfaces que han de presentar las operaciones funcionales, pero no sus interiores. La razón de ello es delegar en terceros componentes la extensión del núcleo por esa vía, la de la definición de los interiores de las operaciones. El subsistema de extensión funcional es uno de esos componentes, el único desarrollado en este proyecto pero no el único que puede desarrollarse. Con él se pretende **dar solución a las necesidades funcionales relacionadas con la representación a bajo nivel de las operaciones definidas en el subsistema de representación genérica**. Si se recuerda, dicho subsistema definía las operaciones AND, OR, $>$, \leq , $=$, \geq , $<$, $+$, $-$, $*$, $/$, $=$, $+=$, $-=$, $*=$ y $/=$. Por otro lado, definía tres tipos de operandos: constante, variable y resultado. No se hablará aquí del cometido de cada operación, puesto que son de sobra conocidas. Las próximas líneas se centrarán en



exponer la representación elegida para cada tipo de operando, así como del método empleado para obtener operaciones compuestas y las consecuencias que conlleva.

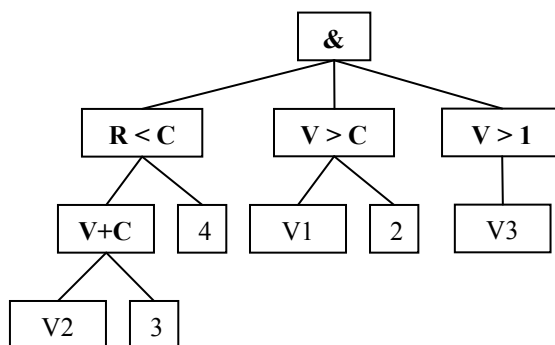


Ilustración 20: Representación conceptual de una operación de bajo nivel

A pesar de que el subsistema de representación genérica emplea una fórmula de composición de operaciones en forma de árbol binario, las operaciones a nivel de núcleo podrían emplear otra bien distinta, pues lo ligero de las interfaces del núcleo así lo permiten. Por ejemplo, podrían definirse operaciones en cuyo interior realizasen todo tipo de cálculos antes de devolver un resultado o provocar un cambio en el estado. Dichos cálculos podrían hacer uso de cualquier secuencia de instrucciones permitidas por la plataforma de destino, auténticos programas embebidos, dando lugar a un bloque compacto y rápido en ejecución. Pero tal representación requiere en .NET de capacidades de **metaprogramación**¹ de las que **no se disponen en XNA**², por lo que se descarta desde un principio, no siendo siquiera evaluada su viabilidad a nivel conceptual. Se empleará, por tanto y de nuevo, una **representación basada en árboles de expresión**, normalmente binarios. Sólo se apartarán de esta representación las operaciones con un número indefinido de operandos, como puedan ser AND, OR y una lista de efectos. El resto tendrán siempre dos operandos salvo, como se verá, cuando uno de ellos sea una constante común, como 0 ó 1, en cuyo caso su valor será implícito a la operación, por lo que no será representado en su estructura interna. Algo también implícito y no representado en esa estructura será el tipo de operación. Ya se comentó que el diseño no pretende hacer posible el uso de metainformación para ningún cometido, por lo que **el tipo de operación formará parte de su código de ejecución, nunca de su estructura**. En definitiva, lo único a representar en una operación será el **número y tipo de sus operandos**, pudiendo ser:

¹ Los tipos del espacio de nombres System.Reflection.Emit lo permiten

² La BCL del .NET Compact Framework no dispone del espacio de nombres System.Reflection.Emit



- Un **único** operando.
- Una **lista** de operandos del mismo tipo, de longitud indefinida.
- **Dos** operandos, cada uno de un tipo potencialmente distinto.

La única observación al respecto de la representación de los operandos que sean el **resultado** de la ejecución de una operación es que se trata de **referencias a las operaciones**. En tiempo de ejecución se realizará la operación y se empleará su resultado para lo que requiera el tipo de operación en el que participa como operando.

En cuanto a las **constantes**, su representación es aún más simple si cabe. La operación se limitará a **guardar internamente el valor de la constante**. Puesto que dicho valor puede ser, en principio, cualquiera, la estructura de la operación ha de guardar espacio para el almacenamiento del valor referido. Ahora bien, existen **valores bien conocidos** que son de uso frecuente como constantes, como son el 0 y el 1. Para evitar en esos casos emplear memoria innecesariamente, se ahorrará el espacio de la estructura de la operación destinado al almacenamiento del valor, haciéndolo **implícito al tipo de operación**. Por ejemplo, la función “variable + constante” tendría que guardar espacio para el valor de la constante, pero su derivada “variable + 1” no lo necesitaría.

La representación de los operandos que sean variables es también sencilla, si bien requiere algo más de explicación dado que, entre otras cosas, se introducirá aquí un derivado de ese tipo de operando del que aún no se ha hablado: la **variable global**. Este tipo de variables se utilizarán para representar aquellas definidas en el ámbito del problema que, por sus características, no tiene sentido representar en el estado. Y es que, **una vez inicializadas, no son modificadas** por operador alguno y, por tanto, por ninguna acción. Representar en el estado una variable que no va a ser nunca modificada conllevaría una reserva inútil de memoria que puede ser ahorrada. Así, este tipo de variables, en lugar de estar presentes en la lista del estado, lo están en una lista global. El empleo de una **variable sólo requiere del almacenamiento de su posición** en el estado, puesto que el estado es parte de la entrada de la operación. Sin embargo, el uso de una variable **global** requiere por parte de la operación la guarda, **además** de la posición, de una **referencia a la lista de variables globales**. En ambos casos, dependiendo de la dependencia de acción o no que tenga la operación, la referencia a la



lista de variables será **indirecta o directa**. Si existe dependencia de acción, la posición almacenada será realmente la de la lista de variables de la acción. En la posición referenciada se encontrará la posición de la variable dentro de la lista de variables del estado (o la lista global si la variable es global). Si no existe dependencia de acción, la posición almacenada hará referencia directa a la posición de la variable dentro de la lista de variables del estado (o la lista global si la variable es global).

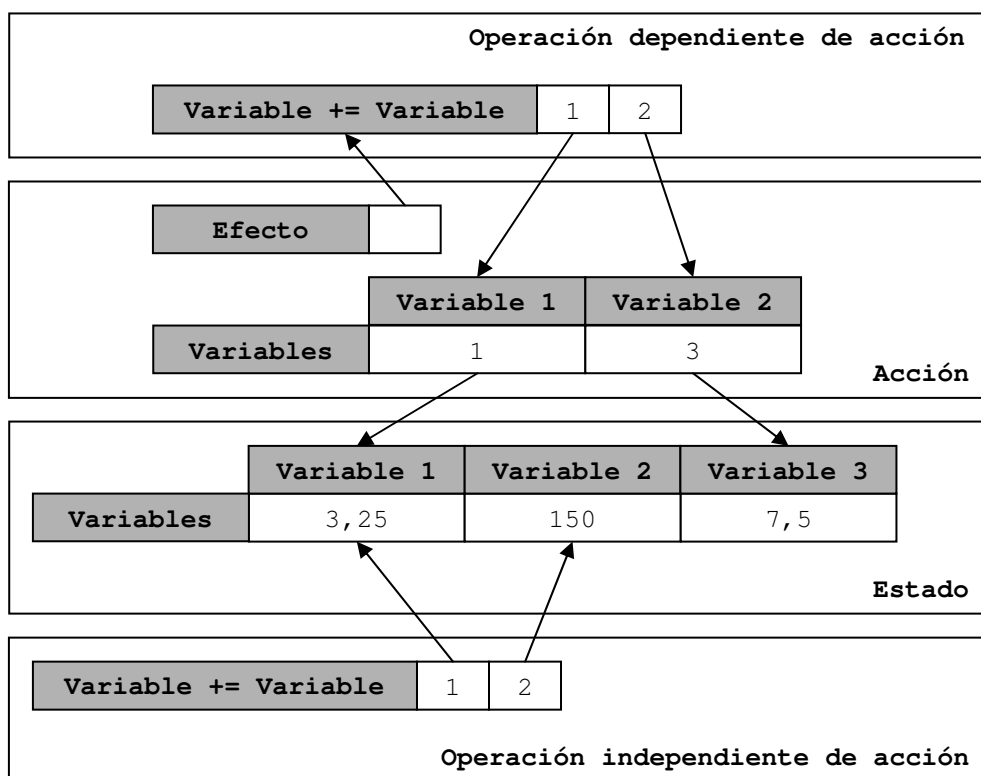


Ilustración 21: Tipos de referencia a variable, directa e indirecta

Por último, destacar que un modelo de representación como este favorece la eficiencia del conjunto, pero **requiere el desarrollo de un número considerable de operaciones**. Habrán de construirse tantas operaciones distintas como combinaciones de operador y tipos de operandos existan, eliminando aquellas que no tengan sentido, como la función “variable + 0” o el efecto “constante = variable”. A cambio de tal explosión de operaciones se obtiene un beneficio en ejecución derivado de la innecesidad de distinción del código a ejecutar en función del tipo de operación y operando, pues es implícito a la operación en sí misma. Dicho con un ejemplo, tener una función del tipo “<operando1> <operador> <operando2>” exigiría hacer algún tipo de selección en tiempo de ejecución del código a ejecutar en función del tipo de “<operando1>”, “<operador>” y “operando2”. Con una representación del tipo



“variable + variable” se sabe de antemano los tipos de la operación (+) y los operandos (variable), por lo que el sistema ha de limitarse a ejecutar un código seleccionado en tiempo de traducción, sin perder tiempo en la selección de ningún código en tiempo de ejecución. Una consecuencia derivada de esto es que la implementación de cada operación requerirá de un número de líneas de código mínima. En cierto modo, el diseño **opta por muchas operaciones con poco código en lugar de pocas operaciones con mucho código**. De cómo realizar la implementación de un número tan grande de operaciones se hablará en el apartado correspondiente.

4.3.6 Subsistema de búsqueda genérica

Planificar es utilizar un algoritmo de búsqueda que, partiendo de la descripción de un problema, intente encontrar un plan que lo resuelva. Pero los algoritmos de búsqueda no sólo se utilizan en planificación automática y, aunque se usen en planificación, no tienen por qué estar ligados a la representación de un problema de ese tipo. Los algoritmos de búsqueda que pueden ser utilizados para resolver cualquier problema de búsqueda gracias a su independencia de la representación de conocimiento se han denominado aquí genéricos. El subsistema de búsqueda genérica **se ocupa de definir algunos algoritmos genéricos, así como los elementos comunes a todos que permiten maximizar la independencia del proceso de búsqueda respecto al algoritmo empleado**. El principal objetivo de este subsistema es que Aran disponga de varios algoritmos de búsqueda que puedan ser seleccionados según las necesidades del usuario. Pero aislando en este subsistema sólo los elementos correspondientes a la búsqueda genérica presente en Aran, sin vincularlos directamente con ningún otro elemento concreto del sistema, se consigue un componente **100% reutilizable** por terceros en proyectos que requieran de este tipo de algoritmos. Y los primeros beneficiados de esto son, precisamente, los videojuegos, pues no son pocos aquellos que requieren llevar a cabo búsquedas de caminos (*pathfinding*).

Para conseguir un marco de trabajo que gire en torno a la utilización de algoritmos de búsqueda, es preciso definir esos elementos comunes a todos ellos que permiten ver al algoritmo como una caja negra que recibe una serie de entradas y da una serie de salidas, siempre del mismo formato. Es, precisamente, ese formato común el artífice de la facilidad de cambio de algoritmo. Dichos elementos son los descritos a continuación.



- Un **estado inicial**. Todos los algoritmos de búsqueda implementados en este proyecto utilizan encadenamiento hacia delante, por lo que precisan de un estado inicial del que partir.
- Una función de **coste**. Puesto que Aran pretende resolver el problema de planificación atendiendo a la minimización o maximización del coste del problema resultante, se requiere que todo algoritmo sea capaz de calcular el coste de un estado dado. Así, la función de coste recibe como entrada el estado y devuelve como salida cuánto (un número) cuesta alcanzarlo desde el estado inicial. Por convención, todos los algoritmos buscarán la **minimización** del coste, pues es lo más común. Nótese que para conseguir que lo maximicen sin necesitar modificar los algoritmos basta con que la función de coste devuelva sus resultados negados.
- Una función **heurística**. En Aran se apuesta principalmente por el uso de algoritmos heurísticos. Para ello es necesario que los que sean de ese tipo reciban una función que tome como entrada un estado y devuelva como salida el coste estimado de alcanzar un estado final desde el dado.
- Una función de validación de **meta**. Un estado final o meta es aquél que cumpla unas determinadas condiciones impuestas en el problema. La función de validación recibe como entrada un estado y devuelve si es o no meta. Con ella se permite abstraer los distintos tipos de condiciones que pueden formularse.
- Un **estado final**. El resultado de una búsqueda en Aran es un estado que cumpla las condiciones de meta, un estado final, y no el camino que va desde el inicial a éste. Esta solución es más genérica puesto que un potencial usuario del algoritmo puede no querer buscar tal camino sino, simplemente, el estado final en sí mismo. Si el usuario quiere como solución el camino mencionado, habrá de obtenerlo él a partir del estado final.
- Una función de **generación de sucesores**. La progresión de los algoritmos de búsqueda se basa en la generación de nuevos estados a partir de los existentes. Esos nuevos estados son sus sucesores. La función de generación de sucesores toma como entrada un estado y devuelve como salida todos sus sucesores. Si el usuario del algoritmo no busca como solución un estado final sino el camino desde el inicial hasta ése, deberá ser esta función la encargada de asegurarse de que cada sucesor



devuelto tiene algún tipo de relación con su padre que pueda ser recuperada posteriormente.

Puesto que este subsistema es genérico, podría incluir realmente algoritmos de búsqueda que no cumplan esta interfaz, pero no han sido desarrollados por estar fuera del propósito de este proyecto. Así, los algoritmos en los que se centra este subsistema son los siguientes:

- **A*** (A estrella) Se trata de uno de los algoritmos más utilizados en la búsqueda de caminos en videojuegos y es el empleado, por ejemplo, por el planificador de F.E.A.R. [Orkin, 2006]. Se trata de un algoritmo completo que devuelve la solución óptima si la heurística empleada es admisible¹, pero con un alto consumo de memoria. Es un derivado del algoritmo Best First (mejor el primero).

```
la lista abierta sólo contiene el estado inicial
la lista cerrada está vacía
mientras queden estados en la lista abierta
    el estado actual es el mejor de los de la lista abierta
    si el estado actual es la meta, devolverlo como solución
    mover el estado actual a la lista cerrada
    para cada sucesor del estado actual
        si el sucesor no está en la lista cerrada
            si el sucesor está en la lista abierta
                si su coste es menor que el de la lista abierta
                    el nuevo padre del de la lista es el estado actual
                    asigna el coste del sucesor al de la lista abierta
                si no, meter el sucesor en la lista abierta
    si se alcanza esta línea, entonces no se ha encontrado solución
```

Ilustración 22: Pseudocódigo del algoritmo Best First

En A* el coste se calcula como la suma de lo devuelto por la función de coste más lo devuelto por la función heurística. Si el valor de la función heurística está ponderado por un valor w (*weight*), el algoritmo se denomina **Weighted A*** y ofrece soluciones en menor tiempo a costa de reducir también su calidad. Si la función heurística devuelve siempre cero, entonces el coste es lo devuelto por la función de coste únicamente y se corresponde con el algoritmo de **Dijkstra**, que es óptimo pero puede llegar a consumir excesivos recursos en memoria y tiempo si el problema es suficientemente complejo.

¹ Es decir, no sobrevalora el coste de alcanzar la meta desde el estado dado



Para hacer estos algoritmos genéricos es preciso abstraer la forma en la que asigna el padre de un estado. Así, se hace necesaria una función de **asignación de padre** que reciba dos estados, el padre y el hijo, y establezca del modo que crea oportuno su relación de parentesco. Con algo así se permite que sea el usuario quien defina de qué forma quiere almacenar las relaciones padre-hijo, si bien comúnmente se guardarán en el estado. Nótese que en ningún momento el algoritmo requiere recuperar dicha relación de parentesco, sino sólo establecerla. Es por ello que no es necesaria su contrapartida. Recuérdesse que el resultado del algoritmo se ha definido como el estado final, y no el camino que va desde el inicial a éste.

- **Hill Climbing** (escalada). Se trata de un algoritmo que si bien no es ni óptimo¹ ni completo², consume pocos recursos de memoria, pues sólo ha de mantener un camino hacia la meta. Es, por tanto, un algoritmo útil en caso de que el consumo de memoria sea un factor importante, aunque ello implique que quizá no se encuentre solución.

```
el estado actual es el estado inicial
hasta que se encuentre o no solución
    si existen sucesores del estado actual
        si algún sucesor es la meta, devolverlo como solución
        ahora estado actual es el sucesor de menor heurística
    si no, no se ha encontrado solución
```

Ilustración 23: Pseudocódigo del algoritmo Hill Climbing

- **Beam Search** (búsqueda en haz). De nuevo un algoritmo que reduce el consumo de memoria frente a otros, menor cuanto menor es el parámetro del que depende (la anchura del haz, b). Otra vez, la contrapartida de la reducción en consumo de memoria es que la completud y optimalidad se ven comprometidas.

```
la lista abierta sólo contiene el estado inicial
mientras que la lista abierta tenga estados
    sustituir todos los estados de la lista abierta por sus sucesores
    si algún sucesor es la meta, devolverlo como solución
    quedarse con los  $b$  estados de menor coste de la lista abierta
    si se alcanza esta línea, entonces no se ha encontrado solución
```

Ilustración 24: Pseudocódigo del algoritmo Beam Search

El subsistema de búsqueda genérica contiene los algoritmos descritos, así como la definición de los elementos que forman parte de su interfaz, pero ningún desarrollo

¹ Es decir, devuelve siempre la mejor solución

² Es decir, encuentra siempre una solución (si existe alguna)



concreto de tales elementos, salvo la **heurística nula**, que devuelve siempre cero, pues no depende ésta de ningún sistema de representación del conocimiento en concreto.

4.3.7 Subsistema de búsqueda específica

El de búsqueda específica puede considerarse una extensión del subsistema de búsqueda genérica. Así, basándose en las definiciones que este último hace sobre el formato que deben cumplir los elementos comunes a los algoritmos de búsqueda hacia delante que contiene, desarrolla componentes que cumplen dicho formato dependiendo expresamente de otros elementos del sistema de planificación construido. Dicho de otro modo, **incluye los componentes necesarios para utilizar el subsistema de búsqueda genérica con el fin de resolver problemas de planificación automática representados mediante los elementos del núcleo**. Incluye también un algoritmo de búsqueda dependiente de la representación de esos elementos del núcleo y, por tanto, no genérico, sólo utilizable por este sistema.

A continuación se exponen los elementos desarrollados por este subsistema en cumplimiento de lo especificado por el de búsqueda genérica:

- Función de **coste**. Aran permite definir una métrica que ha de trasladarse de algún modo al proceso de búsqueda. La función de coste aquí incluida cumple esa misión sirviendo de **adaptador de funciones independientes de acción**. La métrica, por tanto, habrá de traducirse a una función de ese tipo y esta función de coste se limitará a llamar a aquella y devolver al subsistema de búsqueda genérica el valor que ella devuelva.
- Función **heurística**. Aquí se ha utilizado una solución basada en la del planificador FF [Hoffmann & Nebel, 2001]. Éste calcula en primer lugar un plan relajado que toma como estado inicial el recibido por la función heurística. En segundo lugar, devuelve como valor de la heurística la longitud de dicho plan, pues es esa la única métrica que utiliza este planificador. En Aran la métrica es configurable y, como ya se ha dicho, se ha de traducir en una función independiente de acción. Puesto que dicha función devuelve el coste de un estado, el valor devuelto por la heurística resultará de **restar el coste del estado final del plan relajado al del estado inicial** que, recuérdese, es el recibido por la heurística. La forma en la que se obtiene ese



coste de alcanzar la meta desde el estado dado, ese valor heurístico, no garantiza que tal coste nunca sea superior al real y, por tanto, **la heurística no es admisible**.

En el plan relajado de FF **las acciones no tienen por qué guardar un orden coherente**. Así, puede que una acción esté en la posición N del plan sin que las que están en las posiciones anteriores a N hayan hecho ciertas todas sus precondiciones. Por otro lado, la planificación relajada no tiene en cuenta en ningún momento el manejo numérico, esto es, las precondiciones y los efectos funcionales de las acciones, por lo que las acciones del plan tampoco tienen por qué guardar un orden coherente en cuanto al cumplimiento de las precondiciones funcionales se refiere. En ambos casos la heurística de FF no se resiente, puesto que el orden de las acciones le es indiferente. Ahora bien, no ocurre lo mismo en el caso de Aran donde, **según la utilización que haga el usuario de las precondiciones funcionales, pueden llegarse a dar errores en el cálculo de la heurística**, haciendo desaconsejable su uso en tales casos. Y es que, aunque hasta ahora no se ha comentado, calcular el coste del estado final requiere aplicar uno a uno los efectos funcionales de las acciones del plan relajado sobre el estado inicial, el recibido por la heurística. Sólo así puede obtenerse el efecto compuesto que se deriva de la concatenación de estos efectos. Siendo así el proceso, si alguno de los cálculos llevados a cabo por los efectos depende de que se esté cumpliendo la relación de orden de las acciones y, concretamente, la derivada de las precondiciones funcionales, puede producirse el mencionado error. Por ejemplo, una precondición funcional podría ser que cierta variable tuviera un valor mayor o igual que cero. Si el orden de las acciones del plan relajado no garantiza el cumplimiento de esa función y el valor de la variable es utilizado en una división, se producirá un error en tiempo de ejecución debido a una división por cero. Esta consecuencia se conoce y se asume, recordando que el objetivo de este proyecto es la consecución de un planificador con gestión de métrica, y para ello no es requisito el uso de precondiciones funcionales.

Otro efecto conocido y asumido del uso de esta heurística es que, como se recordará, la planificación relajada de FF concluye cuando el último nivel de hechos contiene todos los que se establecieron como metas del problema. Nuevamente, no tiene en cuenta el apartado numérico ni, por tanto, las metas funcionales. Así, en el caso de



que el problema sólo se basase en ese tipo de metas se produciría un efecto indeseado en la función **heurística: siempre devolvería el valor cero**. Ello se debe a que, **siendo las metas sólo funcionales**, la construcción del grafo de planificación relajada de FF se dará por concluido sin pasar del nivel inicial de hechos, puesto que dicho nivel ya contendría todos los hechos meta, ya que todo conjunto contiene el conjunto vacío. Así las cosas, el estado inicial de la planificación relajada sería también el final, con lo que la resta de sus costes daría cero. Se trata, de nuevo, de una consecuencia conocida y asumida que habrá de ser tratada según lo permita la situación. Por ejemplo, si el algoritmo empleado es el A* o el Weighted A*, la consecuencia de esto es que estarían calculando inútilmente la heurística, perdiendo eficiencia y pudiendo ser sustituidos por el algoritmo de Dijkstra. La consecuencia de esto en el Hill Climbing es que siempre se elegirá el primero de los sucesores y en Beam Search se hará lo propio con los b primeros. Para ambos algoritmos, esto supone una pérdida de calidad al no emplear, de hecho, información alguna que guíe la búsqueda.

Existen planificadores cuya heurística, más compleja, tiene en cuenta la parte numérica. El propio FF tiene una versión, Metric-FF, que lo hace parcialmente (pues no considera las metas numéricas). Sin embargo, la heurística de FF se entiende, incluso acompañada de los efectos comentados, suficiente para el propósito de este proyecto. Los casos puntuales que se ven afectados por su uso pueden ser resueltos prescindiendo de ella, pues el diseño del sistema así lo permite.

- Función de **asignación de padre**. En Aran, como ya se ha comentado, los estados guardan una referencia al padre, por lo que esta función se limita a establecer dicha relación entre dos estados dados, no utilizando ningún tipo de estructura auxiliar. Si el algoritmo encuentra una solución será un estado final, algo que no le sirve al sistema de planificación, pues él busca un plan. Será entonces cuando cobren sentido estas relaciones, pues recorriéndolas en orden inverso (de hijos a padres) podrá obtenerse dicho plan.
- Función de generación de **sucesores**. Esta función utiliza el selector de acciones para obtener las **acciones aplicables** al estado dado. Posteriormente, obtiene **un estado sucesor por cada una** de esas acciones para la que se cumpla su



precondición funcional. El estado sucesor se genera mediante la aplicación de la acción al estado dado que conlleva, entre otros, la asignación de la relación de parentesco entre el sucesor y su estado padre y, por consiguiente, la concatenación de las acciones que conforman el plan.

- Función de validación de **meta**. En Aran la función de validación verifica que el **estado contenga los hechos meta y cumpla las metas funcionales**. Pero, en ocasiones y según las necesidades del usuario, puede interesar considerar meta un estado cuando se cumplan otras condiciones que podríamos llamar de ruptura, como haber consumido una cierta cantidad de tiempo o haber alcanzado una cierta longitud de plan. Para dar cabida a este tipo de situaciones la función de validación de meta tiene en cuenta una condición independiente de acción. Así, tanto si se cumple dicha condición como si lo hacen, conjuntamente, los hechos meta y las metas funcionales, el estado será considerado final. Si se observa, se le da el mismo peso a la **condición de ruptura** que al resto, evidenciando así su cometido.

Se ha dicho al comienzo de este apartado que este subsistema contenía, además de lo ya descrito, un algoritmo dependiente de los elementos del núcleo. Se trata del algoritmo de FF, el **Enforced Hill Climbing**.

```
mientras el estado actual no sea solución
  buscar un mejor estado usando sólo las "helpful actions"
  si no se encuentra un mejor estado
    buscar un mejor estado usando todas las acciones
  si no se encuentra un mejor estado
    no se ha encontrado solución
  el mejor estado es ahora el actual
devolver el estado actual como solución
```

Ilustración 25: Pseudocódigo del algoritmo Enforced Hill Climbing

En el Enforced Hill Climbing, la búsqueda de un estado mejor se realiza en amplitud.

```
añadir los sucesores del estado a mejorar a la lista abierta
mientras la lista abierta no esté vacía
  para cada estado en la lista abierta, extraerlo y
    si es mejor que el estado a mejorar
      devolverlo como resultado
    si no es mejor que el estado a mejorar
      añadir sus sucesores a la lista abierta
si se alcanza esta línea, no se ha encontrado un estado mejor
```

Ilustración 26: Pseudocódigo del algoritmo de búsqueda de un mejor sucesor



Si en dicha búsqueda sólo se emplean las “*helpful actions*”, cuando se añaden los sucesores de un estado a la lista abierta sólo se utilizan las acciones que se encuentren dentro del conjunto estimado como útil. Y dicho conjunto no es otro que las que componen el plan relajado obtenido tomando ese estado como inicial. Es decir, se ejecuta una planificación relajada para ese estado y se obtienen las acciones del plan relajado resultante. Luego, para añadir los sucesores del estado a la lista abierta, sólo se considerarán aquellas acciones que, siendo aplicables, formen parte de ese plan relajado. Aparte de la poda que supone la aplicación de la “*helpful actions heuristic*”, el Enforced Hill Climbing emplea otra más que denomina “*added goal deletion heuristic*”. Ésta consiste en no añadir ningún sucesor de un estado a la lista abierta si alguna de las acciones aplicadas para generar a algún sucesor elimina una hecho meta.

La inclusión de este algoritmo debe a que se ha demostrado muy eficiente para ciertos problemas de planificación. Entre sus virtudes está el hecho de que las técnicas de poda que emplea evitan la aplicación de un buen número de acciones y, por tanto, reducen el número de estados explorados y con él el número de veces que se calcula una heurística que es, ciertamente, costosa.

4.3.8 Subsistema de traducción

Hasta ahora se ha hablado de cómo el usuario utiliza un sistema de representación concreto para transmitir su conocimiento al planificador en forma de conceptos abstractos y de cómo el planificador representa, a su vez, tales conceptos empleando estructuras de datos que le permitan realizar, en combinación con algún algoritmo de búsqueda, el proceso de razonamiento necesario para encontrar un plan que resuelva el problema planteado. Sin embargo, falta en esa cadena el eslabón que **se ocupa de la traducción de los conceptos abstractos transmitidos por el usuario a las estructuras de datos del núcleo**, puesto que éste se limita a definir su formato. De dicha traducción se ocupa este subsistema.

4.3.8.1 Codificación básica de *instancias*

Se habrá observado que el núcleo obvia una buena parte de los conceptos definidos por el subsistema de representación genérica. Esto es así porque pretende manejar la mínima información necesaria con objeto de aumentar la eficiencia del proceso de búsqueda del plan. Con el mismo objetivo, su representación se basa en



estructuras estáticas que hacen un uso profuso de identificadores numéricos. Esos números son siempre la posición que ocupa el elemento que identifican dentro de una lista de longitud invariable. Así se manejan listas de hechos, de variables y de acciones. Por tanto, uno de los principales problemas de los que debe ocuparse la traducción es el **establecimiento de correspondencias entre hechos, variables y acciones con las posiciones dentro de sus respectivas listas**. El problema es idéntico para hechos, variables y acciones y todos ellos comparten una serie de elementos en común que el subsistema de traducción explota en su beneficio y materializa a través de los conceptos de *signatura* e *instancia* de *signatura*.

Una **signatura** es la unión de un identificador de *signatura* y una lista de cero o más identificadores de tipo. Así, pueden representarse como *signaturas* los predicados, las funciones y los operadores. Por ejemplo, la *signatura* de un predicado que indica que una unidad militar está disponible podría ser “disponible(unidad)”, la de una función que devolviera la distancia entre dos lugares, “distancia(lugar, lugar)” y la de un operador que permitiera mover una unidad de un lugar a otro “mover(unidad, lugar, lugar)”. Ahí, “disponible”, “distancia” y “mover” son identificadores de *signatura* y “unidad” y “lugar” son identificadores de tipo. En lo sucesivo, se denominará **parámetro** a cada posición de la lista de una *signatura*.

Una **instancia** de *signatura* es la unión de un identificador de *signatura* y una lista de objetos de igual longitud a la de la lista de tipos de la *signatura* a la que se refiere el identificador. Además, el identificador de objeto de la posición p debe referirse a un objeto del mismo tipo que el parámetro p de la *signatura*. Así, pueden representarse como *instancias* de *signatura* los hechos, las variables y las acciones. Por ejemplo, la *instancia* de *signatura* de un hecho que indica que un cierto soldado está disponible podría ser “disponible(soldado)”, la de una variable que devolviera la distancia entre una montaña y un lago, “distancia(montaña, lago)” y la de una acción que permitiera mover el soldado de la montaña al lago “mover(soldado, montaña, lago)”. Ahí, “disponible”, “distancia” y “mover” son identificadores de las *signaturas* que antes han servido de ejemplo, “soldado” es un objeto de tipo “unidad” y “montaña” y “lago” lo son de tipo “lugar”. En lo sucesivo, a las *instancias* de *signatura* se las denominará simplemente *instancias*. Por



otro lado, cuando se hable de **argumentos** se estará haciendo mención a los objetos a los que se refieren los identificadores de la lista de una *instancia*.

A continuación se explicará el proceso que se emplea para traducir una cierta *instancia* a una posición y viceversa. Para ello se comenzará comentando cómo realizar esa traducción dentro de una lista destinada al almacenamiento de *instancias* de una una misma signatura para, posteriormente y de forma sencilla, generalizar el proceso a una lista de *instancias* de diferentes signaturas.

Dentro de la lista de una signatura, **la posición de una *instancia* la determinan sus argumentos** y se calcula a partir de un proceso de síntesis similar a la composición polinómica. Inversamente, es posible conocer a qué lista de argumentos corresponde una determinada posición, empleando esta vez un proceso similar a la descomposición polinómica.

Hablar de polinomios es hacerlo de números. Se hace, pues, necesaria la **traducción de los identificadores de los argumentos a números**. Para ello, se asignará a cada objeto un número que le distinga del resto de los que son de ese mismo tipo. A este respecto, siendo un tipo P el padre de otro H, P y H son considerados tipos distintos. Para hacer patentes las consecuencias de que H herede de P, a los objetos de tipo H también se les considerará de tipo P. Imagínese, por ejemplo, que se dispone de los tipos “arma” y “arma recargable”, que es un subtipo de “arma”, y se tienen los objetos “cuchillo” y “bate”, de tipo “arma”, y “pistola” y “ametralladora”, de tipo “arma recargable”. A la hora de asignar números se confeccionarán listas de objetos de cada tipo, estando la lista de objetos “arma” compuesta tanto por “cuchillo” y “bate” como por “pistola” y “ametralladora” mientras que sólo estas dos formarán la lista de “arma recargable”. En la asignación de números para el tipo “arma”, a los objetos “cuchillo”, “bate”, “pistola” y “ametralladora” se les asignarían los números 0, 1, 2 y 3 respectivamente (“arma 0”, “arma 1”,...). En la asignación de números para el tipo “arma recargable”, “pistola” y “ametralladora” recibirían los números 0 y 1 respectivamente. Nótese, por tanto, que un mismo objeto tendrá tantos números asignados como tipos a los que pertenezca. Por ejemplo, “pistola” es “arma 2” y “arma recargable 0”. Obsérvese que la



numeración es de base cero. Esto es importante de cara a la traducción *instancia*-número.

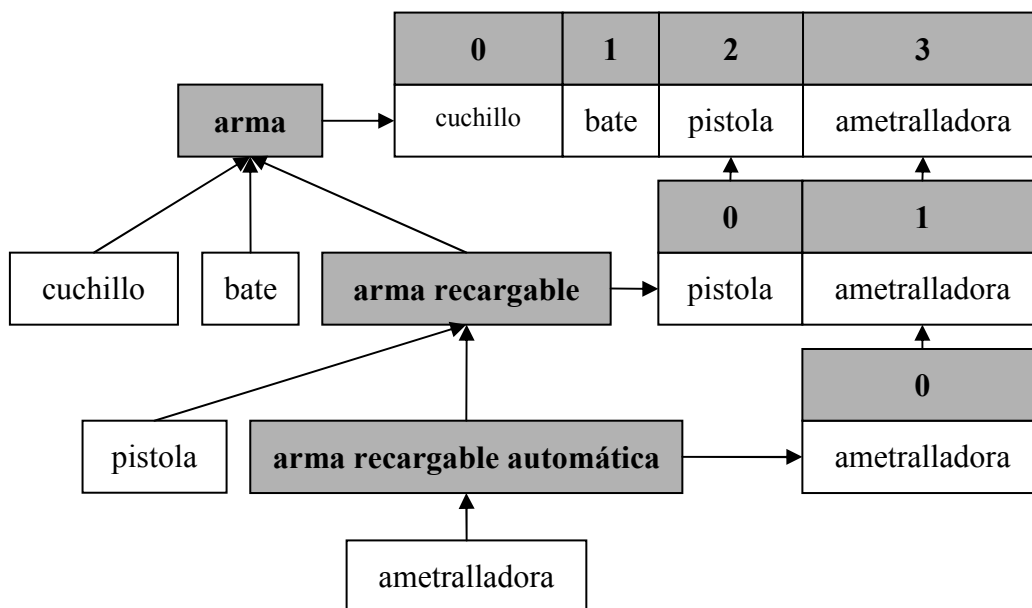


Ilustración 27: Traducción de objetos a números

Es común que los lenguajes de planificación dispongan de un **tipo raíz** del cual derivan todos los demás. Es útil de cara a la reutilización, pues cuanto más genérico es el tipo del parámetro de un predicado, una función o una acción, se podrá aplicar a más objetos. Ahora bien, explicado aquello en lo que se traducen las relaciones de parentesco, se considera innecesario obligar a todo dominio a disponer de ese tipo raíz pues, haciéndolo, se fuerza a la creación de una larga lista de objetos que puede no llegar a utilizarse nunca. Es por ello que en el diseño de Aran este tipo raíz es **opcional**.

Explicado el proceso de traducción de objetos a números dependiendo del tipo con el que actúen esos objetos, en lo sucesivo se hablará de los objetos T0, T1,... TN del tipo T, tal y como sucede en el proceso de traducción.

Para la explicación de la codificación de los argumentos de una *instancia* se utilizará una signatura con 3 parámetros de tipo A, B y C respectivamente. Considérese que, por otro lado, se dispone de 3 objetos de tipo A, 2 de tipo B y 4 de tipo C. Ello podría dar lugar a un total de 24 *instancias* de esa signatura, que se obtienen de combinar todos los argumentos de todas las formas posibles ($24\ ABC = 3\ A \times 2\ B \times 4\ C$). Identificando las *instancias* por su combinación de argumentos se tendría A0B0C0, A0B0C1, A0B0C2,..., A1B0C3,..., A2B1C0, etc. La codificación consiste en hacer



equivaler un número, de 0 a 23 (número de *instancias* – 1), a cada una de estas combinaciones de argumentos, de tal modo que $A0B0C0 = 0$, $A0B0C1 = 1$, $A0B0C2 = 2, \dots$, $A1B0C3 = 11, \dots$, $A2B1C0 = 20$, etc.

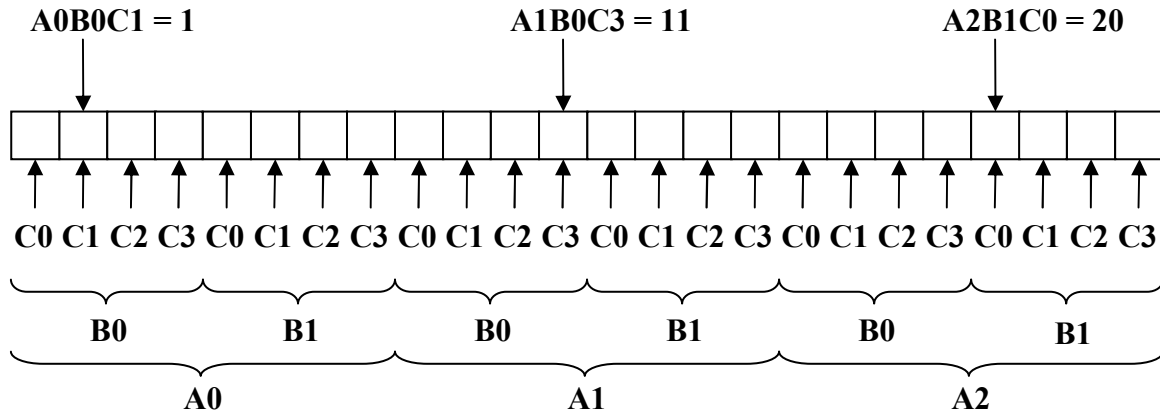


Ilustración 28: Traducción visual de argumentos de una *instancia*

La Ilustración 28 refleja el ejemplo mencionado, mostrando las posiciones de las *instancias* $A0B0C1$, $A1B0C3$ y $A2B1C0$. Dicha ilustración permite visualizar cómo la asociación *instancia*-número equivale a un árbol conceptual de selección en el que primero se localiza la *instancia* según el valor de A, luego según el valor de B y, finalmente, el de C. Así puede saberse a qué número equivale una *instancia* e, inversamente, a qué *instancia* equivale un número.

La composición/descomposición polinómica ya comentada es el equivalente matemático a esta traducción visual. Para poder llevarla a cabo es preciso, en primer lugar, **calcular el tamaño de las secciones de selección** que pueden verse en la Ilustración 28, que se hace como sigue. Recorriendo los parámetros de derecha a izquierda (en este caso, C-B-A), el tamaño de sección de un cierto parámetro es el resultado de multiplicar el tamaño de sección del parámetro anterior por el número de objetos del parámetro anterior. El tamaño de sección del primer parámetro, que carece de predecesor, es siempre 1. Siguiendo con el ejemplo, y como puede verse en la Ilustración 28, el tamaño de sección de C es 1 (es el primer parámetro en el recorrido), el de B es 4 (1×4 , 1 es el tamaño de sección del parámetro de tipo C y 4 el número de objetos de tipo C) y el de A es 8 (4×2). El tamaño de sección no depende, pues, del tipo del parámetro, sino de su posición. Obsérvese que si, por ejemplo, el parámetro de tipo A lo fuera de tipo C, su tamaño de sección seguiría siendo 8.



Conocidos los tamaños de sección, la **codificación de argumentos de una instancia** consiste en componer un número mediante una suma de tantos sumandos como argumentos, de tal modo que el sumando correspondiente al argumento N es el resultado de multiplicar el número que tiene ese objeto para el tipo del parámetro N por el tamaño de sección de dicho parámetro. Así, para la instancia A2B1C0, el sumando correspondiente al parámetro de tipo A sería 16 (2 x 8, 2 es el número del objeto para el tipo A y 8 el tamaño de sección del parámetro de tipo A), para el tipo B sería 4 (1 x 4) y para el tipo C sería 0 (0 x 1). Por tanto $A2B1C0 = 20$ ($16 + 4 + 0$). Del mismo modo se obtiene cualquier otro, como $A1B0C3 = 11$ ($1 \times 8 + 0 \times 4 + 3 \times 1$).

La **decodificación de un número de instancia** es el proceso inverso, esto es, una descomposición basada en un proceso iterativo de división. Así, el número debe ser dividido, en primer lugar, por el número de objetos del tipo del parámetro más a la derecha, el C en este caso. El resto de la división será el número correspondiente al argumento de ese parámetro y el cociente habrá de ser ahora dividido por el número de objetos del tipo del siguiente parámetro (B en este caso) para obtener el siguiente número de argumento. El proceso concluye cuando el cociente de una de las divisiones sea 0, siendo también 0 el valor de todos los argumentos que queden por averiguar en ese momento. Volviendo al ejemplo, tal y como muestra la Ilustración 29, el número de instancia 20 (A2B1C0) habría de ser dividido por 4 (el número de objetos de tipo C) dando 0 como resto (el argumento de tipo C). El cociente, 5, habría de ser dividido ahora por 2 (el número de objetos de tipo B) dando 1 como resto (el argumento de tipo B). Finalmente, el cociente, 2, habría de ser dividido por 3 (el número de objetos de tipo A) dando 2 como resto (el argumento de tipo A) y 0 como cociente, finalizando el proceso. Si aún quedasen por averiguar algunos números de argumento, éstos serían siempre 0. En este caso no queda ninguno por averiguar, pero sí sería el caso de, por ejemplo, el número de instancia 2 (A0B0C2).

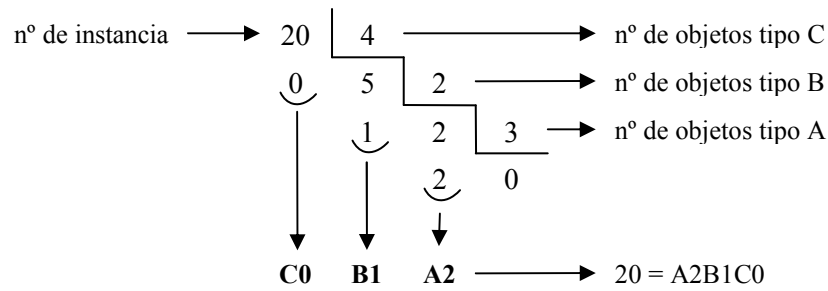


Ilustración 29: Decodificación de argumentos de una instancia

Conocida la forma de traducir de una *instancia* a un número y viceversa dentro de una lista de instancias de una misma *signatura*, un sencillo mecanismo de desplazamiento permite **aglutinar instancias de distintas signaturas** en una misma lista. Para ello es necesario disponer una tras otra las listas de cada *signatura*, de tal modo que a todos los números de *instancia* de una lista se les sume un **desplazamiento** (*offset*) igual a la suma del número de *instancias* de las listas que le preceden. Figúrese que se tienen las *signaturas* X, Y y Z con, respectivamente, 5, 3 y 7 *instancias*. Agrupar todas las *instancias* en una única lista significaría, tal y como puede observarse en la Ilustración 30, sumar 0 a las de X (puesto que no le precede ninguna lista), 5 a las de Y (las *instancias* de X) y 8 a las de Z (las *instancias* de X e Y).

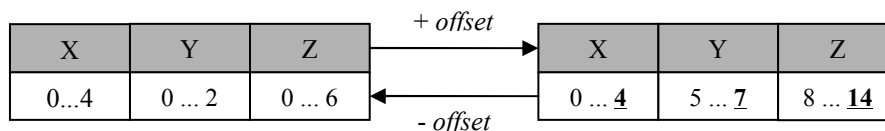


Ilustración 30: Agrupación de números de instancia

La misma cantidad que se suma en la codificación es necesario restarla como parte del proceso de decodificación. Para ello es preciso conocer el **número máximo de instancia de cada signatura** después del desplazamiento. En el ejemplo, la *instancia* máxima de X es 4, de Y es 7 y de Z es 14. Así, si un número de *instancia* es menor o igual a 4, entonces será de tipo X y habrá de restársele 0. Si no, si es menor o igual a 7 será de tipo Y y se le restará 5. Si no, si es menor o igual a 14 será de tipo Z y deberá restársele 8.

Téngase en cuenta que las **variables** constituyen un caso especial a la hora de realizar esta agrupación. Ello es debido a la existencia de **dos listas** distintas: la de variables del **estado** y la de variables **globales**. Este hecho requiere que se agrupen en la



primera lista sólo las funciones que generen variables de estado y en la segunda sólo las que hagan lo propio con las globales. Al no existir una lista conjunta de variables tampoco existe una numeración única. Las variables de estado serán numeradas con independencia de las globales pudiendo darse el caso de que en ambas listas se repitan los números de *instancia*. Esto no representa ningún problema a la hora de mezclar números de *instancia* de variables de estado y globales en una misma lista, como sucede en las acciones (que sólo tienen una lista de variables), ya que la operación que haga uso de la variable se encargará de hacer referencia a la lista apropiada.

Recapitulando, codificar una *instancia* consiste en obtener el número que se corresponde con sus argumentos y sumarle el desplazamiento correspondiente a su signatura. Por su parte, decodificar un número de *instancia* es restarle el desplazamiento asociado a su signatura y obtener los argumentos que se corresponden con el número resultante de dicha resta.

4.3.8.2 Codificación optimizada de *instancias*

Con lo comentado hasta ahora se puede realizar la traducción de hechos, variables y acciones a sus respectivas posiciones en las correspondientes listas manejadas por los elementos del núcleo. Ahora bien, quedarse aquí implicaría que el núcleo guardase siempre todas las *instancias* de todas las signaturas, algo que en verdad no sucede ya que el proceso de traducción conlleva una optimización que elimina las *instancias* innecesarias, abriendo huecos en las listas que han de ser gestionados.

De cara a su representación en el estado, se considera innecesario todo hecho cuyo valor se mantiene inalterado durante el proceso de búsqueda. Un caso extremo es el de los **predicados estáticos**, que son aquellos que no aparecen nunca en las listas de adiciones ni eliminaciones de los operadores. Es por ello que ningún hecho de un predicado estático formará nunca parte del estado. El caso más típico de predicado estático es el de desigualdad “`distinto(tipo, tipo)`”. De hecho, la mayoría de los planificadores lo incluyen por omisión, de tal modo que puede hacerse uso del mismo sin necesidad de definirlo previamente. En Aran, como sucedía en el caso del tipo raíz, se ha preferido no predefinir la desigualdad pero dar opción a poder hacerlo. Así, puede crearse la asociación entre un identificador de predicado y un tipo diciendo que constituyen un **predicado de desigualdad**. El subsistema de traducción se encargará, entonces, de crear un hecho de ese predicado por cada dos objetos distintos del mismo



tipo. Obsérvese que, en combinación con el tipo raíz, permite definir un predicado de desigualdad aplicable a cualquier par de objetos del problema.

Siguiendo con los hechos, otro caso es el de los **predicados de un solo sentido**, que son aquellos que sólo aparecen en las listas de adiciones o en las de eliminaciones, pero no en ambas. Si sólo aparecen en las adiciones se dice que el sentido es **de adición** y los hechos iniciales no necesitan ser parte del estado, puesto que su valor será siempre verdadero al no poder ser eliminados. Si sólo aparecen en las eliminaciones se dice que el sentido es **de eliminación**. Entonces todos los hechos no iniciales no necesitan ser parte del estado, puesto que su valor será siempre falso al no poder ser añadidos. Un ejemplo de predicado con sentido de eliminación sería “abierta(puerta)” cuando sólo existan acciones que cierren puertas y ninguna que las abra. Negando la lógica empleada, el predicado “cerrada(puerta)” tendría sentido de adición cuando se usase para reflejar el mismo conocimiento que con “abierta(puerta)”.

Por su parte, de cara a su representación en la lista de acciones, se consideran innecesarias las **acciones no alcanzables**, esto es, aquellas que nunca podrán llevarse a cabo porque sus argumentos no cumplen las precondiciones del operador del que derivan. Si se equiparan las acciones no alcanzables a los hechos iniciales, se está ante un caso idéntico al de los predicados con sentido de eliminación que se resuelve, también, de idéntico modo. En ambas situaciones se tienen unas **instancias iniciales** que serán las **máximas** a tener en cuenta durante el proceso de búsqueda, pues en su transcurso no podrán añadirse más *instancias*. Por tanto, pueden repartirse el total del nuevo conjunto de números reducido resultante de eliminar una cantidad igual a la de *instancias* innecesarias. En la Ilustración 31 se puede observar cómo de un conjunto de 11 *instancias* se eliminan todas menos las iniciales 1, 5, 7 y 8 de tal modo que el nuevo número de la *instancia* 1 es el 0, de la 5 es el 1, etc. Obsérvese que la **reasignación** es tan sencilla como dar a cada *instancia* el número de posición de base cero que ocupa en la lista ordenada de instancias iniciales. **Deshacer la reasignación** consiste en el proceso inverso, esto es, obtener el número de *instancia* inicial que se encuentra en tal posición.

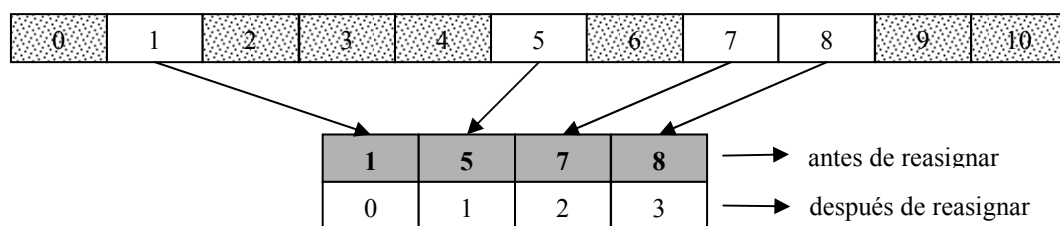


Ilustración 31: Reasignación de números de *instancias* máximas

El caso de los predicados con sentido de adición, las **instancias** (los hechos) iniciales son las **mínimas** que habrán de tenerse en cuenta durante el proceso de búsqueda, pues pueden irse añadiendo más durante el transcurso del mismo. Puesto que aquí las irrelevantes son las *instancias* iniciales, son éstas las eliminadas de la lista. Es por ello que la **reasignación** consiste en un desplazamiento (*offset*) de cada *instancia* no inicial igual al número de *instancias* iniciales que le preceden.

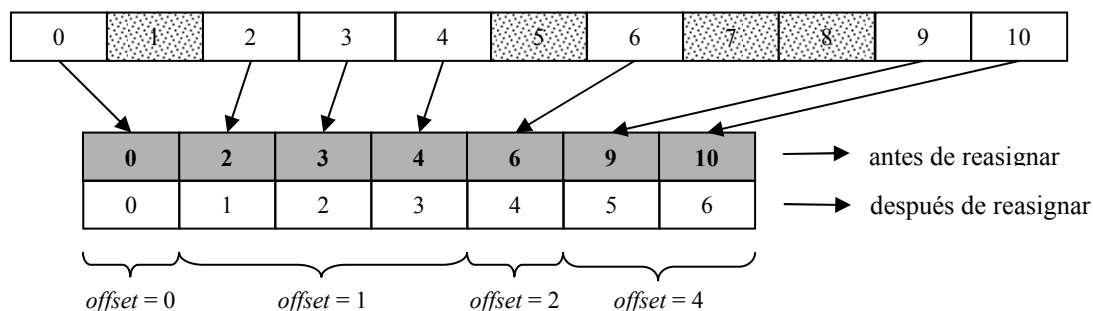


Ilustración 32: Reasignación de números de *instancias* mínimas

La Ilustración 32 muestra un ejemplo de este caso, partiendo de los mismos datos iniciales ya mostrados con anterioridad. En ella puede verse cómo, por ejemplo, el nuevo número de la *instancia* 3 es el 2 (puesto que le precede una *instancia* inicial, la 0) o de la 9 es el 5 (le preceden 4 *instancias* iniciales: 1, 5, 7 y 8). Obsérvese que en este caso el desplazamiento no es uniforme sino que varía por intervalos. Así, por ejemplo, de la *instancia* 2 a la 4 el desplazamiento es de 1 mientras que de la 9 a la 10 es de 4. Por tanto, para **deshacer la reasignación** habrá que restar el desplazamiento que corresponda según el intervalo al que pertenezca el número reasignado. Por ejemplo, al número reasignado 3 habrá que restarle 1 por pertenecer al intervalo [1, 3].

Recapitulando, codificar una *instancia* consiste en obtener el número que se corresponde con sus argumentos, reasignar a ese número el valor que le corresponde tras eliminar las *instancias* innecesarias de su signature y, finalmente, sumarle el desplazamiento correspondiente a dicha signature. Por su parte, decodificar un número



de *instancia* es restarle el desplazamiento asociado a su signatura, deshacer la reasignación de ese número y obtener los argumentos que se corresponden con el número resultante de deshacer la reasignación. Obsérvese que el desplazamiento asociado a una signatura puede verse afectado por la eliminación de *instancias* innecesarias en las que le preceden, pues a menor número de *instancias* en signaturas precedentes, menor desplazamiento.

Antes de concluir este apartado, conviene resaltar que las optimizaciones aquí explicadas permiten la **reducción del tamaño del estado** y, por tanto, de la memoria consumida durante el proceso de búsqueda. Por otro lado, también permite reducir el tamaño de las listas de acciones y, debido a ello, **de la tabla de selección** de las mismas, con el consecuente incremento en la velocidad de selección.

4.3.8.3 Generación de acciones y tabla de selección

Habiendo ya determinado el método de codificación y decodificación de *instancias*, el siguiente paso en la traducción es la generación de acciones. Para ello, por cada operador, se itera sobre todas las posibles **combinaciones de argumentos** que puede tener en virtud a los objetos presentes en el problema. Recordando el ejemplo de la Ilustración 28, en el que se tenían, por un lado, tres parámetros de tipos A, B y C y, por otro, 3 objetos de tipo A, 2 de tipo B y 4 de tipo C, habría que iterar sobre 24 combinaciones de argumentos.

Lo primero que debe decidirse para cada combinación de argumentos es si resulta en una acción no alcanzable y, por tanto, que no debe ser generada. Una acción no es alcanzable cuando alguno de los hechos de su lista de precondiciones es inicialmente falso y su predicado es estático o tiene sentido de eliminación. En esos casos es imposible que el hecho sea nunca verdadero y, por tanto, que se cumplan todas las precondiciones de la acción. Un **análisis de alcanzabilidad** exhaustivo debería ocuparse también de determinar si existen posibilidades de que las precondiciones funcionales puedan ser ciertas, pero tal análisis se descarta en Aran por su complejidad.

Para todo operador que tenga alguna acción alcanzable, el subsistema de traducción **genera las precondiciones y los efectos funcionales** del mismo, pues serán, como se explicó, comunes a toda acción de tal operador. Para la construcción de estos elementos se emplea el subsistema de extensión funcional.



El siguiente paso en la generación de toda acción alcanzable es su **identificación**, que se lleva a cabo mediante su número de *instancia*., obtenido por medio el método de codificación optimizada ya visto.

Finalmente, la acción se completa con la generación de las listas de **precondiciones, adiciones, eliminaciones y variables**, que contienen números de *instancia* que deben ser calculados mediante el mismo método. En la generación de estos números ha de tenerse en cuenta que un argumento puede tomar la forma de distintos tipos según la signatura en la que tome parte. Por ejemplo, imagínese que se tiene la acción “conducir(moto)”, donde “moto” es un objeto de tipo “vehículo de dos ruedas”. Ahora figúrese que el hecho “disponible(moto)” es una de sus precondiciones pero el predicado “disponible” requiere un objeto de tipo “vehículo”. Obviamente, el tipo “vehículo de dos ruedas” es un subtipo de “vehículo”. Ahora bien, si se recuerda, los subtipos son considerados distintos a sus tipos padre a la hora de codificar por lo que, por ejemplo, “moto” puede ser el “vehículo de dos ruedas 1” y, al tiempo, el “vehículo 3”. A la hora de codificar el hecho “disponible(moto)”, “moto” debe considerarse como el “vehículo 3” y no el “vehículo de dos ruedas 1”.

Como optimización, la lista de **precondiciones** de una acción carece de hechos derivados de **predicados estáticos**, si bien sí se tienen en cuenta en el análisis de alcanzabilidad. Estas precondiciones estáticas tampoco son reflejadas en la **tabla de selección de acciones**, por innecesarias. Dicha tabla se va generando al mismo tiempo que las acciones siguiendo el formato descrito cuando se trató el núcleo, reflejado en la Ilustración 18. Así, siguiendo con el ejemplo anterior, y suponiendo que “conducir” fuera el operador 3, “conducir(moto)” fuera la acción 2 y “disponible(moto)” fuera el hecho 5 y la precondición 4 de tal acción, habría de asignarse el valor verdadero a la celda de la fila 5 que estuviera en la columna correspondiente al operador 3, la precondición 4 y la acción 2.

4.3.8.4 Gestión de errores

El subsistema de traducción, durante la generación del resultado, realiza numerosas comprobaciones que pueden resumirse en la lista que figura a continuación. Las comprobaciones se notifican como advertencias o errores según su gravedad. Las



advertencias no impiden la generación del resultado mientras que los errores provocan el abortamiento de la traducción.

1. **Advertencia:** un tipo, objeto, parámetro, predicado, hecho, función, variable u operador que ha sido declarado no se utiliza.
2. **Advertencia:** un operador no es ejecutable, esto es, no existen objetos que puedan ser utilizados como argumentos del mismo.
3. **Advertencia o error:** el tipo o el valor de un operando no es el esperado.
4. **Error:** el número de argumentos de un hecho o una variable es distinto del número de parámetros de su correspondiente predicado o función.
5. **Error:** el tipo de un argumento no es el mismo que el del parámetro correspondiente.
6. **Error:** un hecho está presente más de una vez una misma lista.
7. **Error:** un hecho está presente, al mismo tiempo, en la lista de adiciones y, o en la de precondiciones o en la de eliminaciones.
8. **Error:** dos tipos, objetos, parámetros, predicados, hechos, funciones o variables tienen el mismo identificador.
9. **Error:** se hace referencia a un tipo, objeto, predicado, hecho, función o variable no declarado.
10. **Error:** el tipo de una operación no es el esperado.
11. **Error:** un operador no tiene efectos.
12. **Error:** un hecho meta no es alcanzable, es decir, no es inicial y su predicado es estático o tiene sentido de eliminación.

4.3.9 Subsistema de planificación

Para poder tener un sistema de planificación es necesario **combinar los componentes vistos hasta ahora y organizarlos de algún modo con el fin de** cumplir íntegramente el proceso de planificación, desde el momento en el que se define el problema a resolver hasta que se ofrece una solución al mismo o, en su defecto, se informa de la imposibilidad de resolverlo. Ese modo de combinarlos y organizarlos será el que aporte la funcionalidad final conjunta y limite el ámbito e utilización del sistema generado. El subsistema de planificación es el resultado de uno de esos modos de combinación y organización, que se centra en **resolver problemas expresados en ficheros XML y ofrecer su solución, también, en forma de fichero XML**. El resto de



posibles combinaciones habrá de realizarlas el usuario de Aran de acuerdo a sus necesidades. La razón de ser de este subsistema es, de hecho, ofrecer un planificador funcional que pueda servir de referencia a otros que puedan implementarse utilizando los componentes de Aran.

En verdad hay poco que añadir sobre este subsistema a lo ya indicado cuando se habló de la arquitectura del sistema. El diseño de Aran relega, adrede, a este subsistema a un segundo plano, restándole toda la funcionalidad posible y derivándola a los componentes ya vistos con el fin de maximizar su reutilización. Por tanto, su misión se reduce a organizar la secuencia de pasos que componen el proceso de planificación, evitando así que un usuario que quiera resolver problemas utilizando ficheros XML tenga que reproducir tales pasos cada vez que quiera hacer uso del sistema. Aunque parece obvio, conviene resaltar que los ficheros XML manejados por este subsistema han de estar escritos utilizando el lenguaje definido por el subsistema de representación específica.

Una de las tareas del subsistema de planificación es ofrecer al usuario una interfaz de más alto nivel que lo visto hasta ahora. Para ello se sirve de unos pocos **parámetros de configuración** que influirán en el proceso de planificación. Dichos parámetros son los reflejados a continuación.

- **Nombres de los ficheros.** Se asignan por omisión los nombres de los ficheros de entrada (“domain.xml” y “problem.xml”) y salida (“plan.xml”), ofreciendo la posibilidad de cambiarlos.
- **Tipo raíz.** Se define por omisión un tipo raíz (“object”), ofreciendo la posibilidad de eliminarlo o sustituirlo por otro.
- **Predicados de desigualdad.** Se define por omisión el predicado de desigualdad para el tipo raíz (“notEqual”), permitiendo eliminarlo o establecer cualquier otro para ese o cualquier otro tipo.
- **Algoritmos.** La planificación por omisión tiene lugar haciendo uso del algoritmo A^* . Se ofrece, por otro lado, la posibilidad de establecer cualquier secuencia de algoritmos, de cualquier longitud mayor que cero, pudiendo emplear cualquiera de los comentados en esta memoria. Se define un valor por omisión (2) para los parámetros de b y w de los algoritmos Beam Search y Weighted A^* , permitiéndose su cambio.



- **Condición de parada.** Por omisión, la planificación termina cuando se encuentra solución al problema o se agotan los algoritmos de búsqueda de la lista sin que ninguno de ellos haya dado un resultado positivo. Sin embargo, se permite detener la planificación cuando se haya alcanzado una cierta longitud de plan y/o se haya invertido más de un cierto tiempo en planificar.

Puede observarse que la configuración ha sido diseñada con un objetivo en mente: **minimizar el número de cambios** que el usuario deba hacer cuando decida hacer uso del planificador. Así, incluso si el usuario no decide modificar valor alguno, el planificador estará listo para ofrecer una salida. Por otro lado, el número de parámetros a configurar y, por tanto, de acciones a realizar utilizando el subsistema de planificación, es mucho menor que haciendo uso directo de los componentes que se han venido comentando a lo largo de la memoria. Por tanto, se cumplen los principios de simplicidad y utilidad que rigen el diseño de este subsistema.

Una vez configurado, cuando el subsistema de planificación recibe la orden de iniciar **el proceso de planificación**, se ocupa en primer lugar de verificar la existencia de los ficheros de entrada. Hecho esto, combina el subsistema de traducción con el de representación específica, al cual proporciona los datos de entrada extraídos de los ficheros. Tras ello, utiliza el resultado devuelto por el subsistema de traducción para crear los parámetros requeridos por el primero de los algoritmos de búsqueda. Así, crea la función de coste a partir de la función independiente de acción que devuelve la traducción de la métrica y establece las condiciones de parada como condiciones de ruptura de la función de validación de meta, a la cual se le indica como objetivo el estado final devuelto en la traducción. Además, si el algoritmo lo requiere, establece otros parámetros, como el generador de sucesores, la función de asignación de padre o la heurística, que requieren otros elementos derivados de la traducción, como la lista de acciones o la tabla de selección. Completada la construcción del algoritmo, lo ejecuta partiendo del estado inicial, también devuelto por la traducción. Si la ejecución resuelve el problema, la planificación finaliza. Si no, se construye el siguiente algoritmo y se ejecuta en busca de la solución. El proceso de planificación concluye cuando algún algoritmo encuentra solución o se agotan todos los de la secuencia establecida. En caso de encontrar solución, el subsistema de planificación extrae, a partir del estado final devuelto por el algoritmo, la secuencia de identificadores de las acciones que componen



el plan. Finalmente, emplea de nuevo el resultado devuelto por el subsistema de traducción para decodificar dichos identificadores y obtener, para cada acción, su identificador de operador y sus argumentos. Por último, se ocupa de escribir el plan en el fichero de salida, verificando previamente que se trata de una ruta correcta y accesible. Si en algún momento del proceso de planificación se produce algún evento que provoque su cese, el subsistema de planificación lo notifica mediante mensajes de texto descriptivos. Entre esos mensajes, destacan los derivados de la traducción de los ficheros de salida, en los cuales se indica el fichero, la línea y la posición en la que se localiza el error. Aparte de los mencionados mensajes, el subsistema de planificación ofrece otra información acerca del proceso de planificación, como el tiempo invertido en la traducción o en la búsqueda.

4.3.10 Subsistema de interacción

El subsistema de interacción es, simplemente, una **interfaz textual de usuario a través de la cual poder dar órdenes al subsistema de planificación y, al tiempo, recibir la información emitida por éste**. Concretamente, se trata de un comando pensado para ser utilizado desde la consola del sistema. La idea es que este comando sirva de herramienta funcional a aquellos que cumplan el perfil de investigador tratado al comienzo de este apartado, incluyendo a desarrolladores de videojuegos que quieran probar sus dominios y problemas en formato textual antes de transcribirlo a otro más apropiado para la integración en el juego. El diseño del comando se basa en estándares ampliamente extendidos, que a continuación se comentan.

En primer lugar, el comando muestra un mensaje breve de identificación, denominado logo o **cabecera**, en el que se incluye el nombre de la herramienta, su versión y su autor, acompañados de una nota de aviso que **indica que es software libre carente de garantías**, así como los pasos a seguir para obtener más información acerca de la licencia del producto.

Mostrado el logo, el comando configura el subsistema de planificación basándose en los argumentos recibidos y lo ejecuta. Durante la ejecución, va mostrando en tiempo real al usuario la **información transmitida por el planificador**, como los mensajes de error en la traducción, el tiempo invertido en traducir, el empleado en planificar, el algoritmo de búsqueda en uso o si se encontró o no solución. Es preciso resaltar que en caso de búsqueda fructífera, **la solución no se muestra** al usuario. Esta



decisión se debe a que se considera que el formato del lenguaje XML empleado hace difícil la lectura de cantidades de texto no pequeñas en una interfaz de consola, por lo que se prefiere que el usuario haga uso del medio que estime oportuno para visualizarla según sus preferencias. No ocurre lo mismo con los **mensajes de error** referidos a fallos en los ficheros de entrada. En ese caso se muestra, para cada error, un extracto de la línea en la que se produce. Aquí la limitación a una única línea por error se estima suficientemente legible como para ser mostrado por consola.

Los argumentos que puede recibir el comando son, prácticamente, una correspondencia 1:1 con los admitidos por el subsistema de planificación, algo normal siendo, como es, su interfaz textual. Siguiendo con los estándares, entre los argumentos, los **identificadores de parámetro han de estar precedidos de un prefijo**. Lo común es que en sistemas Windows dicho prefijo sea “/”, mientras que en los sistemas Unix se prefiere “-”. Siendo el comando una herramienta que podría estar funcionando en cualquiera de esos sistemas, se ha estimado conveniente la admisión de ambos tipos de formato, si bien a la hora de mostrar la ayuda se emplea el más común en la plataforma en la que esté teniendo lugar la ejecución. Continuando con el formato de los **parámetros**, el comando incluye algunos **de uso común** en todo tipo de comandos, como “verbose”, “noLogo” o “help”, y en los destinados al procesamiento de lenguajes, como “noWarn” o “warnAsError”. Además, los parámetros más habituales disponen de **formato corto** (como “v” para “verbose”) para agilizar su uso. Mención especial requiere el correspondiente al mencionado “help” pues, otra vez, en sistemas Windows se suele utilizar uno (“?”) distinto al de los sistemas Unix (“h”). Nuevamente, se aceptan ambos, si bien la ayuda muestra el más común en la plataforma de ejecución. Al respecto de los formatos cortos, decir que los algoritmos de búsqueda más típicos disponen también de ellos (por ejemplo, “EHC” para el “Enforced Hill Climbing”).

Como puede observarse, **los parámetros están en inglés**, en consonancia con los objetivos del proyecto. Ahora bien, ello no quiere decir que la interfaz no soporte **otros idiomas**. De hecho, admite también el español y cuantos quieran añadirse en un futuro, pues se ha preparado para ello. Por omisión, el comando tomará el lenguaje predeterminado del sistema en el que se ejecute, permitiendo el establecimiento de uno concreto mediante la opción “language”. Es importante señalar que **la localización afecta sólo a elementos que no constituyan parte de la interfaz de uso del comando**.



En esa interfaz de uso se incluyen los nombres de los comandos, de los ficheros de entrada y salida por omisión, del tipo raíz, de su predicado de desigualdad y de los algoritmos de búsqueda. También forma parte de ese conjunto el formato de los argumentos numéricos, siendo este el correspondiente al inglés de Gran Bretaña. Esto hace que, por ejemplo, el separador de decimales de los números presentes en los argumentos sea siempre el punto (“.”) frente a la coma (“,”) o cualquiera que sea el separador correspondiente a la cultura local.

La independencia de cultura respecto a los elementos descritos se ha decidido con el fin de facilitar el uso del comando por parte de *scripts*. De este modo, el editor del *script* no ha de preocuparse por la cultura del sistema en el que se ejecute, pues no tendrá efecto sobre la forma de utilizar el comando. Otro aspecto también tenido en cuenta para facilitar su integración en *scripts* es el uso de **códigos de terminación**. Siguiendo nuevamente los estándares, el comando devuelve el valor 0 si todo transcurre sin problemas. La devolución de cualquier otro número será indicador de error, empleándose un número diferente según el tipo de error que se haya producido. A estos efectos, no encontrar solución al problema se considera un error.

El último aspecto de diseño del comando que merece la pena ser resaltado es el **nombre del fichero ejecutable**: “`aran.exe`”. Se trata de un nombre corto que, al tiempo que identifica claramente su vínculo con el sistema al que pertenece, Aran, agiliza su uso. Además, el hecho de estar en minúsculas facilita su utilización en sistemas Unix, en los que se distinguen las mayúsculas a la hora de localizar ficheros. Por su parte, la extensión “.exe” no debe llevar a engaños. No sólo es el estándar de los ficheros ejecutables en Windows, sino que lo es también en la plataforma .NET. Por tanto, independientemente de la plataforma en la que se ejecute, el usuario esperará que un fichero de estas características tenga esa extensión.

4.4 Diseño detallado

El nombre de este apartado no debe dar lugar a equívocos. Aquí detallado **no quiere decir más extenso, sino menos abstracto**. Así, si el interés del anterior apartado era la explicación de los conceptos sobre los que se sustenta el diseño de Aran, lo que aquí se pretende es exponer cómo se materializan dichos conceptos en forma de **elementos directamente vinculados a la plataforma de desarrollo**. Este apartado no pretende ser, ni mucho menos, un recorrido exhaustivo por todos y cada uno de esos



elementos, pues eso es algo de lo que ya se ocupa la documentación de referencia elaborada al efecto como parte de este proyecto. Se hará, pues, mención sólo a aquellas partes consideradas de especial interés para comprender, en conjunción con lo explicado hasta el momento, cómo está construido el sistema. Para ello, se llevarán a cabo las simplificaciones sobre la realidad que se estimen oportunas, en pro de una mejor comprensión.

4.4.1 Espacios de nombres

Los conceptos comentados hasta ahora se transforman, en la plataforma .NET, en tipos relacionados entre sí y agrupados en **espacios de nombres**¹. Para confeccionar dicha agrupación se ha partido de la definición hecha de cada uno de los subsistemas, dando lugar a una correspondencia directa entre ambos conceptos, reflejada en la siguiente tabla.

Subsistema	Espacio de nombres
Representación genérica	Aran.Representation
Representación específica	Aran.Representation.Xml
Núcleo	Aran.Core
Extensión funcional	Aran.Functional
Búsqueda genérica	Aran.Searching
Búsqueda específica	Aran.Searching.Specialized
Traducción	Aran.Translation
Planificación	Aran.Planning.Xml
Interacción	Aran.Planning.Xml.Command

Tabla 3: Correspondencias entre subsistemas y espacios de nombres

Como puede observarse, todos los espacios de nombres están **en inglés**, como lo estarán el resto de los elementos descritos a lo largo del apartado y como lo han estado ya otros tratados con anterioridad. Por otro lado, la nomenclatura que siguen es la recomendada en la guía de desarrollo de Microsoft [MSDN, 2005] que, de hecho, dicta los **estándares a seguir** en la plataforma .NET. El diseño de Aran trata de ajustarse al máximo a dichos estándares, pues se considera un requisito indispensable en un proyecto que, como este, está pensado para ser utilizado y/o continuado por terceros.

¹ En .NET, los espacios de nombres son el equivalente a los paquetes de UML



Se indicaba al comienzo que el diagrama está simplificado. Se ha hecho así con el fin de mejorar su entendimiento. Una de las simplificaciones ha consistido en no representar determinados tipos, como la excepción **RepresentationException**, que la implementación ha de lanzar cuando se produzca algún tipo de error durante el manejo del modelo de objetos. Las enumeraciones **MetricType**, **OperandType** y **OperationType** tampoco han sido incluidas en el diagrama. Para mayor claridad, decir que los valores de **MetricType** son *Minimization* y *Maximization* mientras que los de **OperandType** son *Variable*, *Constant* y *Operation*. Los valores de **OperationType** se recogen en la siguiente tabla, con sus símbolos equivalentes entre paréntesis.

OperationType	
Assignment (=)	Inferiority (<)
SumAssignment (+=)	NonSuperiority (≤)
SubtractionAssignment (--)	Equality (=)
MultiplicationAssignment (*=)	Inequality (< >)
DivisionAssignment (/=)	NonInferiority (≥)
Sum (+)	Superiority (>)
Subtraction (-)	Conjunction (∧)
Multiplication (*)	Disjunction (∨)
Division (/)	

Tabla 4: Valores del enumerado **Aran.Representation.OperationType**

Las licencias representativas se refieren a la forma en la que se han reflejado determinados elementos en el diagrama. Por ejemplo, todos los campos¹ mostrados se corresponden realmente con **propiedades** de sólo lectura². Se ha obviado el empleo de estereotipos por simplicidad, ya que las interfaces no pueden tener campos y no es posible confundir ambos elementos en la representación. Por otro lado, las **relaciones 1:1** se corresponden con métodos de obtención. Por ejemplo, el método `GetKnowledge():IKnowledge` de `IKnowledgeProvider` se deriva de la relación existente entre `IKnowledgeProvider` y `IKnowledge`. Algo similar sucede con las **relaciones de agregación**. En este caso se traducen en una propiedad de sólo lectura

¹ Se prefiere esta denominación para hacer referencia a los atributos de UML, porque en .NET el término atributo está reservado a un tipo especial destinado a la programación declarativa.

² Lo que en otros lenguajes se conocen como métodos de acceso o *getters*.



que devuelve el número de elementos y un método que permite iterar sobre ellos. Así, por ejemplo, la relación de agregación entre `IProblem` y `TFact` con rol `InitialFactIds` se traduce en la propiedad `InitialFactCount:int` y el método `GetInitialFactIds():IEnumerable<TFact>`¹. Si se observa el elevado número de relaciones de este tipo, podrá comprenderse que hacer explícitas estas correspondencias habría complicado la representación y, en consecuencia, el entendimiento de las relaciones entre tipos, yendo contra la filosofía del diagrama.

Mención aparte requiere la licencia tomada con respecto a los parámetros **genéricos**². Éstos no se incluyen en los nombres de las interfaces y, en verdad, todas hacen uso de esta característica. Tampoco se explicitan los argumentos de *instanciación* (*bindings*) de dichos parámetros en las relaciones. Sí se han empleado, no obstante, los tipos genéricos en los miembros de las interfaces e, incluso, se ha representado alguno explícitamente para mejorar la legibilidad. Son tipos genéricos `TType`, `TOp`, `TObj`, `TFact`, `TVar`, `TPred` y `TFunc` (los dos últimos no están en el diagrama) que representan, respectivamente, los tipos de los identificadores de tipo, operador, objeto, hecho, variable, predicado y función. El usuario de estas interfaces deberá *instanciar* estos tipos genéricos con otros concretos, a su conveniencia, de forma similar a como sucede en otros lenguajes como C++. Son también genéricos `TIns` y `TSig` que representan, respectivamente, los tipos de los identificadores de *instancia* y *signatura*. Estos parámetros son *instanciados* por las interfaces de este espacio de nombres, nunca por quien las implemente. Aunque sus *bindings* no se reflejan en el diagrama, el par `TIns-TSig` es *instanciado* como `TFact-TPred` o `TVar-TFunc` dependiendo de si están participando en la definición de hechos o variables. Nuevamente, se considera que incluir este tipo de detalles en el diagrama habría jugado en contra de su entendimiento.

4.4.3 Espacio de nombres `Aran.Representation.Xml`

La Ilustración 34 recoge algunos elementos representativos de todos los que componen este espacio de nombres. Aprovechando esta simplificación, aquí sí se ha querido hacer explícita la presencia de los parámetros genéricos para resaltar que **todos los identificadores son cadenas de caracteres** (`string`). Se entenderá recordando que este sistema de representación se corresponde con un *parser* de ficheros de texto.

¹ En .NET, la interfaz `IEnumerable` es parte del patrón de diseño *Iterator*.

² En .NET, los genéricos son el equivalente a los parámetros de clase de lenguajes como UML y C++.

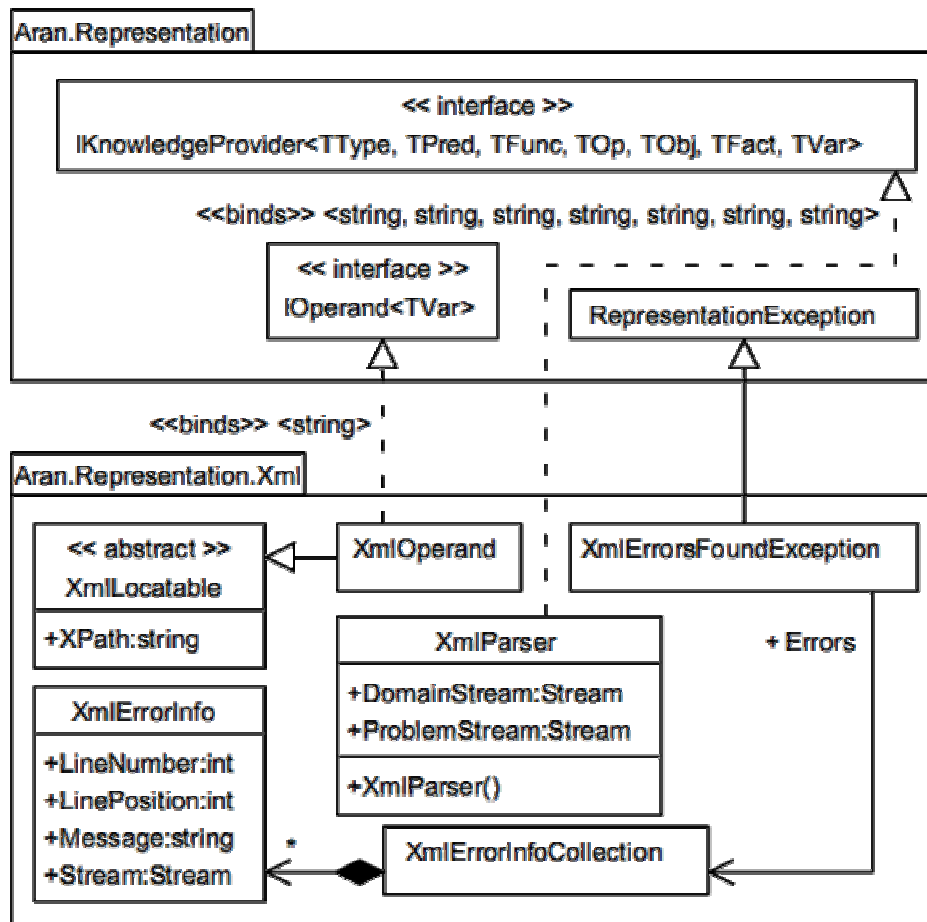


Ilustración 34: Diagrama de clases de `Aran.Representation.Xml`

Este espacio de nombres dispone de **una clase por cada interfaz presente en `Aran.Representation`** de tal modo que cada clase implementa su correspondiente interfaz. El diagrama incluye, a modo de ejemplo, la correspondencia entre `IOperand` y `XmlOperand`. Todas estas clases heredan de `XmlLocatable`, lo que les permite disponer de una propiedad (**`xPath`**) a través de la cual puede localizarse el lugar del fichero del cual se han extraído los datos que componen un cierto objeto de esa clase.

Dejando de lado este tipo de clases que siguen un mismo patrón, conviene resaltar la clase **`XmlParser`**, que hace las veces de proveedor de conocimiento en este modelo, como puede verse en el diagrama. Las propiedades `DomainStream` y `ProblemStream` permiten asignarle (son las únicas que no son de sólo lectura) al **parser** las **fuentes de datos para el dominio y el problema**. Obsérvese que las fuentes **son streams**, por lo que realmente no tienen por qué limitarse a un fichero, si bien será el caso más típico. El origen de esos *streams* corre a cargo del usuario del *parser*.



Por último, si durante el tratamiento de los datos provistos por las fuentes de datos se produjera algún error, se lanzaría una excepción (`XmlErrorsFoundException`), que contiene una colección de **mensajes de error** (`XmlErrorInfo`) que incluye su descripción y el número de línea en el que se produjeron.

En general, cuando quiera crearse un nuevo subsistema de representación específica habrá de realizarse una implementación de las interfaces de `Aran.Representation`. Tal y como se hace en este espacio de nombres, esa implementación podrá incluir elementos propios que ayuden a la integración con otros sistemas. Asimismo, podrán emplearse otros tipos como argumentos de los parámetros genéricos. Bien aprovechada, esta característica ofrece una interesante vía de integración con otros sistemas. Por ejemplo, los identificadores de objeto en una representación ligada a un juego de estrategia en el que las unidades se representan mediante la clase `Unit`, tal clase puede hacer las veces de identificador de objeto en su implementación específica de las interfaces de `Aran.Representation`.

4.4.4 Espacio de nombres `Aran.Core`

La Ilustración 35 muestra la mayoría de los tipos que pueden encontrarse dentro de este espacio de nombres. En general, el diseño de dichos tipos gira entorno a la maximización de la eficiencia en tiempo de ejecución. Para ello, se toma como premisa que todos los datos manejados contienen valores válidos en todo momento. Un entorno que garantice esa premisa se considera seguro. Sólo en entornos seguros está garantizado el buen funcionamiento de **los tipos** aquí contenidos, puesto que, entre otras medidas, **no realizan comprobación alguna de los valores manejados y suprimen el principio de encapsulación** en sus diseños. Aquí no existen propiedades, todo son campos públicos. Por otro lado, algunas estructuras devueltas son reutilizadas entre llamadas. Son, por tanto, tipos eficientes a costa de ser inseguros. El entorno habrá de ocuparse de proporcionar un nivel de seguridad apropiado para su ejecución de la forma en que estime conveniente. Externalizando la seguridad del entorno se maximiza también la reutilización de este espacio de nombres, pieza clave, por central, del diseño de Aran.

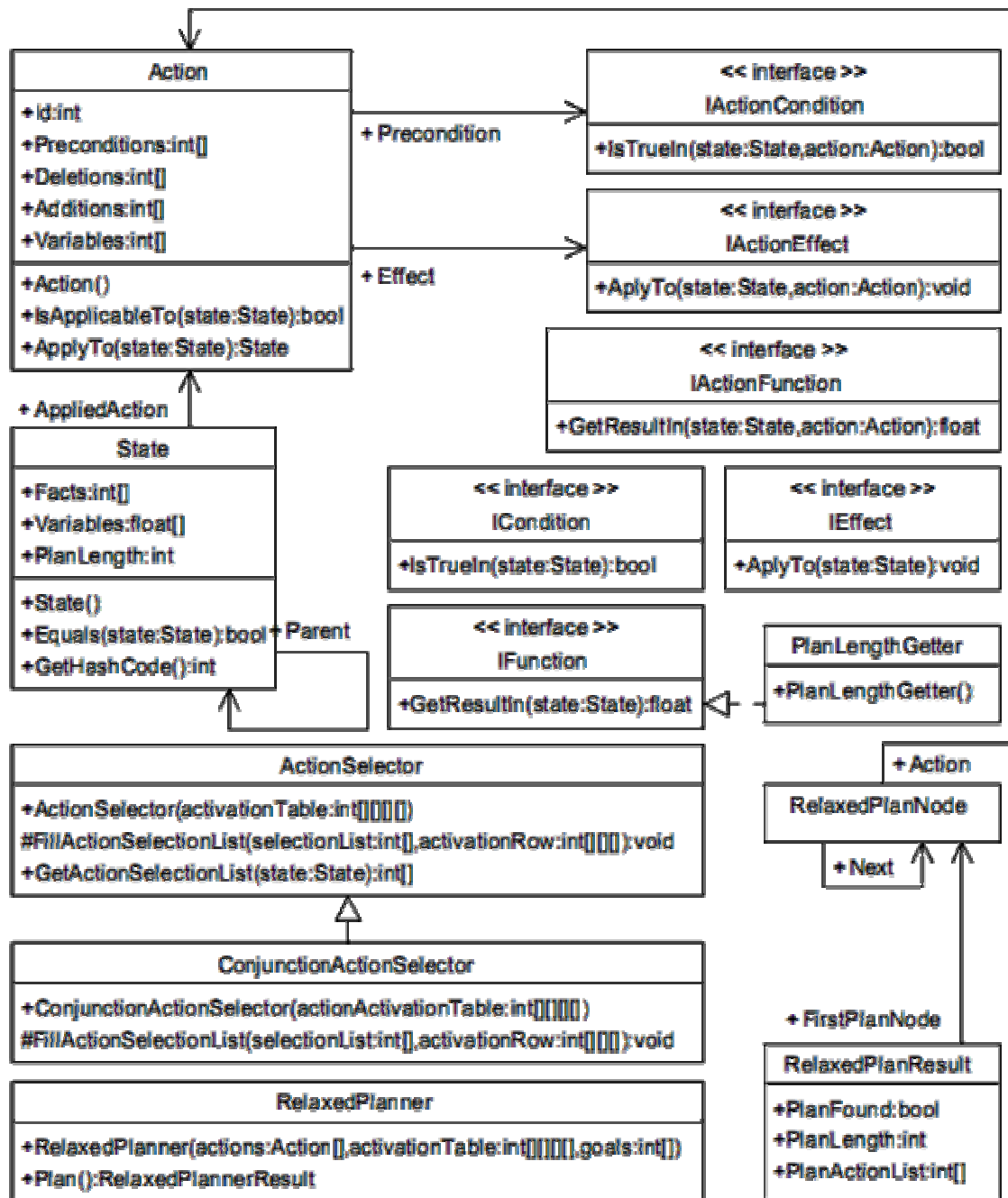


Ilustración 35: Diagrama de clases de Aran.Core

En la ilustración puede observarse en qué se traducen los distintos elementos tratados en el diseño conceptual. Por ejemplo, **las operaciones se representan en forma de interfaces** a implementar por terceros (dada la variedad posible), recogiendo su dependencia o no de acción tanto en el nombre (ICondition frente a IActionCondition, por ejemplo) como en los parámetros de sus métodos (que reciben o no una Action). Obsérvese el **enfoque funcional** dado a dichas interfaces, que sólo contienen la definición de un método. Para tal enfoque también podrían



haberse empleado delegados¹ en lugar de interfaces de método único. Se opta por interfaces con el fin de evitar el mayor nivel de indirección del que hacen uso los delegados² en las llamadas a método, pues tal indirección se traduce en pérdida de eficiencia.

Las clases **Action** y **State** sirven para representar, respectivamente, acciones y estados. Puede observarse cómo las listas que los componen se han traducido a la estructura de datos más eficiente de la plataforma: el *array*. Las **listas de números** tienen una correspondencia directa siendo bien **arrays de float** (cuando se almacenan valores de variables) o **de int** (cuando son números identificadores de hecho). No sucede lo mismo con las **listas de booleanos**, pues no son *arrays* de *bool*, sino de *int*. Estos *arrays* de *int* representan **arrays de bits** (el tipo *bit* no existe) guardando 32 valores en cada posición siguiendo un formato idéntico al empleado por la clase *BitArray* del espacio de nombres *System.Collections* de la BCL, si bien tal clase no se emplea en ningún momento por ser menos eficiente. **La elección de int** como tipo base no es casual. Se debe a las pruebas de rendimiento realizadas con ese y otros tipos enteros, como el *byte* (8 bits), el *short* (16) o el *long* (64). Dichas pruebas probaron que un array del mismo número de posiciones de *byte*, *short* o *int* se tarda en procesar idéntico tiempo teniendo, sin embargo, mucha más cantidad de información en el último caso (el doble que con *short* y el cuádruple que con *byte*). Este hecho se debe a que las operaciones a pila del CLR no emplean nunca valores inferiores a 32 bits. Por su parte, *long* se demostró más eficiente que *int* sólo cuando el número de elementos presentes en el array es alto (cerca de los 30.000), algo que no se estima común en Aran.

Las clases **ActionSelector** y **ConjunctionActionSelector** se ocupan de encapsular la funcionalidad del selector de acciones. En verdad es la segunda la que refleja fielmente tal funcionamiento, siendo la primera una clase base para otras implementaciones. Esa base se ocupa de “aplanar” la tabla de selección dando lugar a una única columna, siguiendo el proceso explicado en el diseño conceptual. Luego el método *FillActionSelectionList* es el encargado de procesar esa columna para rellenar los valores de la lista de selección de acciones. Cumpliendo el patrón *Template*

¹ En .NET, los delegados son el equivalente a los punteros a función de otros lenguajes como C.

² Los delegados no son en verdad punteros, sino objetos que llaman a otros objetos.



Method, el diseño delega en `ConjunctionActionSelector` para que rellene la lista aplicando la operación de conjunción a las precondiciones de las acciones. Si se quisiera aplicar cualquier otra operación, simple o compuesta, bastaría con heredar de `ActionSelector` e implementar `FillActionSelectionList` de otro modo.

Por su parte, la clase **RelaxedPlanner** incluye la funcionalidad del planificador relajado. El método `Plan` se ocupa de lanzar un proceso de planificación que sigue de forma fiel y directa lo descrito por [Hoffmann & Nebel, 2001] haciendo uso de elementos auxiliares privados no representados en la Ilustración 35. La clase **RelaxedPlanResult** recoge el resultado de la planificación relajada incluyendo dos elementos a destacar. Por un lado, la lista de acciones que componen el plan. Por otro, un *array* de bits de tantas posiciones como acciones totales existen en el cual tienen valor 1 las posiciones correspondientes a las acciones presentes en el plan. Si se observa, las acciones están identificadas por un número entero (el número de *instancia*). Ese número es el de la posición que estará a 1 en el *array* en caso de que la acción forme parte del plan. Para aumentar la eficiencia, todas **las estructuras devueltas** por `RelaxedPlanner`, sin excepción, **son reutilizadas en cada llamada**. Es labor del entorno realizar una copia de dichas estructuras si lo considera oportuno, lo cual se estima poco común.

Tanto el selector de acciones como el planificador relajado hacen uso de **la tabla de selección**. Ésta ha sido representada como un *array* de `int` de 4 dimensiones. Dicho *array* contiene tantos *arrays* de 3 dimensiones como hechos forman parte del estado. Cada uno de esos *arrays* tridimensionales alberga tantos *arrays* de 2 dimensiones como operadores existen. A su vez, cada uno de los *arrays* bidimensionales contiene tantos *arrays* unidimensionales como precondiciones tiene el operador al que corresponde. Finalmente, cada uno de esos *arrays* unidimensionales representa un *array* de bits de tantas posiciones como acciones se corresponden con ese operador. Las posiciones con valor 1 se corresponden con las acciones que tienen esa precondición como cierta cuando el hecho correspondiente es también cierto, tal y como se explicó conceptualmente. Se trata de una estructura de datos simple pero que requiere algunos datos adicionales para ser manejada, como el número de acciones por operador. La representación simplificada que supone la Ilustración 35 no muestra estos datos para facilitar su legibilidad.



Antes de terminar con este espacio de nombres, merecen ser mencionadas dos clases que no han sido representadas en el diagrama que se muestra en la ilustración. Se trata de **FactsOnlyStateEqualityComparer** y **SomeVariablesStateEqualityComparer**. Ambas clases son comparadores de igualdad de estado que implementan la interfaz `IEqualityComparer<T>` del espacio de nombres `System.Collections.Generic` de la BCL y permiten comparar los estados teniendo en cuenta sólo los hechos o sólo algunas variables, respectivamente. En el caso de `SomeVariablesStateEqualityComparer`, su constructor recibe un array de bool que indica cuáles de las variables del estado han de ser tomadas en consideración y cuáles no.

4.4.5 Espacio de nombres Aran.Functional

Más simple que otras mostradas hasta ahora es la Ilustración 36, que contiene sólo algunos ejemplos de los tipos que componen este espacio de nombres.

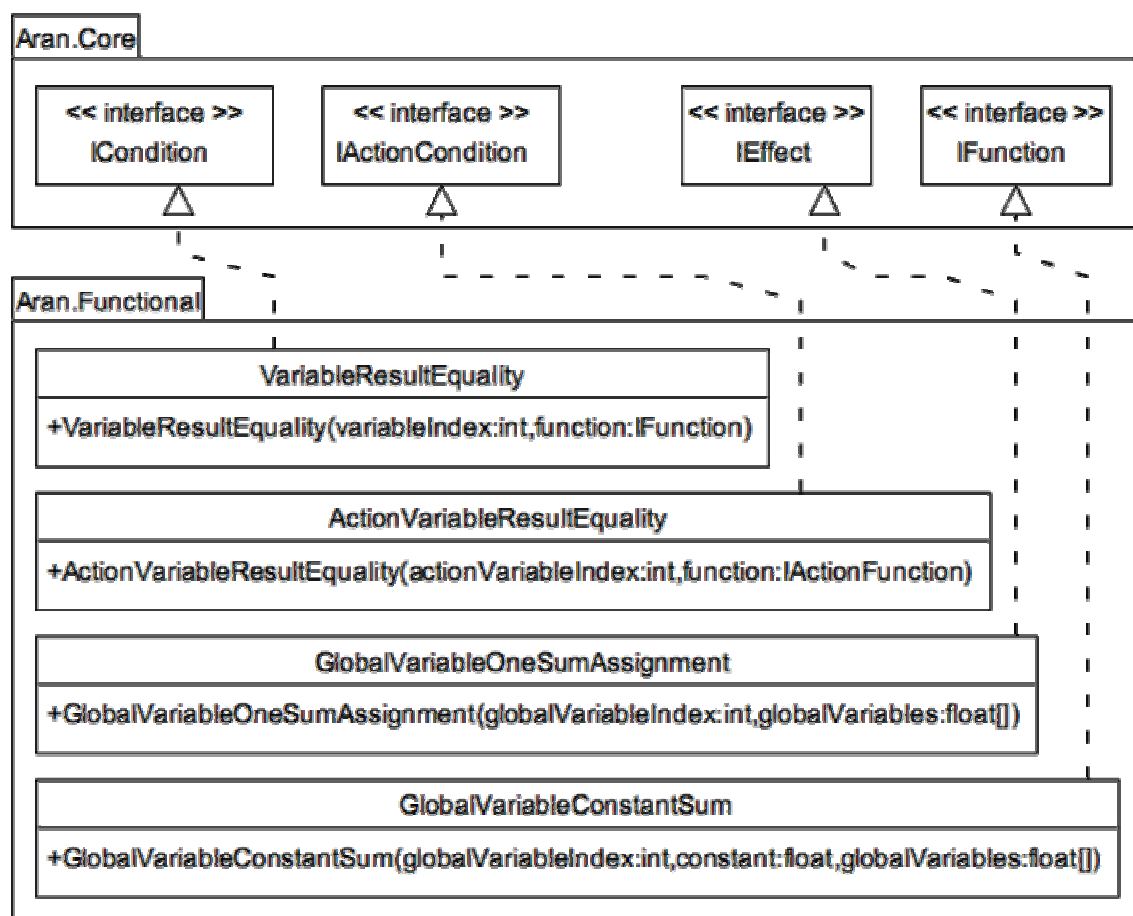


Ilustración 36: Diagrama de clases de `Aran.Functional`



Todos los tipos de `Aran.Functional` siguen las mismas directrices. Se trata siempre de **clases simples de contenido mínimo cuyo nombre refleja claramente la operación a la que corresponden**. Por ejemplo, `VariableResultEquality` se corresponde con la condición de igualdad entre una variable y el resultado devuelto por una función. Como todo tipo en este espacio de nombres, implementa una interfaz de operación perteneciente a `Aran.Core`. En este caso, `ICondition`, por ser una condición independiente de acción. Puede verse cómo **la dependencia de acción** de su contrapartida, `ActionVariableResultEquality`, **se refleja tanto en su nombre como en la interfaz que implementa**, `IActionCondition`. Por otro lado, todo tipo de este espacio de nombres recibe como **argumentos de su constructor los datos necesarios para llevar a cabo la operación**, relativos a los tipos de los operandos. Así, `VariableResultEquality` recibe el número de variable y la función, de tal modo que en tiempo de ejecución comparará la variable de estado que se encuentre en la posición `variableIndex` con el valor devuelto por el método `GetResultIn` de `function`. Obsérvese que la dependencia se propaga a los operandos. Así, `function` es de tipo `IFunction` en `ActionVariableResultEquality` y de tipo `IActionFunction` en `ActionVariableResultEquality`. Otra diferencia entre ambos tipos de operación es la que se explicó en el diseño conceptual acerca de la doble indirección que tiene lugar en las operaciones dependientes de acción. Entonces también se habló de otros tipos de operandos, como las constantes o las variables globales. En esos, tal y como muestra la Ilustración 36, los nombres de las clases reflejan los tipos de los operandos con `Constant` o `GlobalVariable`, según el caso, recibiendo en el constructor un valor `float` (el de la constante) o `int` (el de la posición de la variable global). En el caso de la variable global se recibe, además, un *array* de `float` que representa la lista de variables global. Por su parte, las operaciones para las que se haya estimado útil, dispondrán de versiones con constantes de valor implícito cero o uno, que se verá reflejado en el nombre de la clase (`Zero` o `One`) y en el constructor, que no recibirá el parámetro correspondiente.

Tal y como sucede en `Aran.Core`, los tipos de este espacio de nombres también presuponen su **ejecución dentro de un entorno seguro**, por lo que no realizan comprobación alguna de los datos que manejan, a fin de favorecer la eficiencia.



4.4.6 Espacio de nombres Aran.Searching

En la Ilustración 37 se representa, siempre de forma simplificada, uno de los algoritmos de búsqueda que se encuentran implementados en Aran, el A*, que servirá para ejemplificar al resto, puesto que todos siguen la misma tónica en su diseño.

En primer lugar, cabe decir que este espacio de nombres es, quizá, el que refleje con más fuerza el **diseño orientado a componentes** que se ha seguido en Aran con el fin de flexibilizar el marco de trabajo que conforma. Esto se ve claramente en las dependencias que la clase `AStarAlgorithm` tiene con el resto de tipos representados y lo simple de su interfaz, que se limita a implementar el único método de `ISearchAlgorithm`.

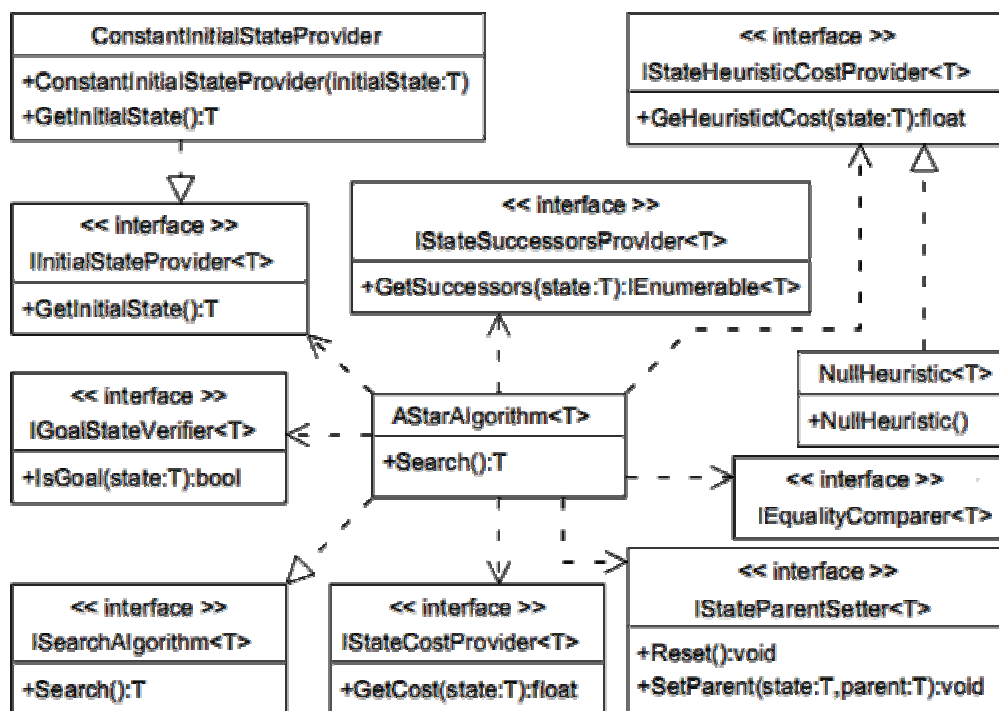


Ilustración 37: Diagrama de clases de Aran.Searching

La interfaz `ISearchAlgorithm` no es la única de un solo método, sino que esa ha sido la tendencia seguida en el diseño de este espacio de nombres con claros tintes **funcionales**. Así, se representan de modo similar las funciones de coste (`IStateCostProvider`), heurística (`IStateHeuristicCostProvider`), de validación de meta (`IGoalStateVerifier`) o de asignación de padre (`IStateParentSetter`), así como otros componentes como el generador de sucesores

(`IStateSuccessorsProvider`) o el proveedor del estado inicial (`IInitialStateProvider`).

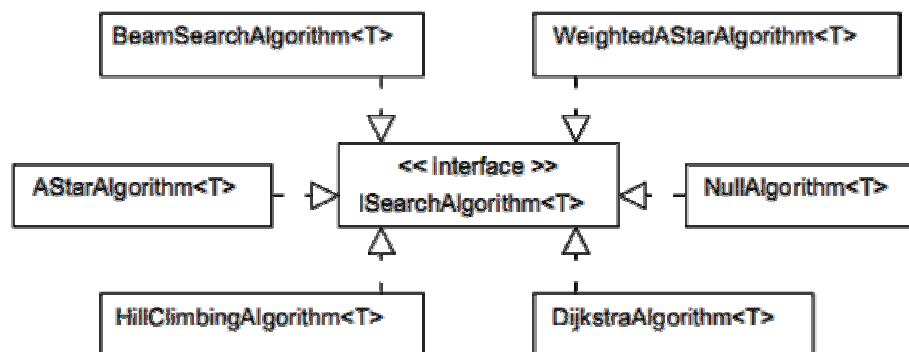


Ilustración 38: Diagrama de clases de los algoritmos de Aran. Searching

Como sucede con el A* y la clase `AStarAlgorithm`, en `Aran.Searching` **cada algoritmo se corresponde con una clase** sin relación pública con el resto, tal y como refleja la Ilustración 38. Aunque, por simplicidad, no ha sido representado en la Ilustración 37, el algoritmo recibe en su constructor los componentes de los que depende y de los cuales hace uso una vez se invoca al método `Search`. Entre esos componentes se encuentra uno al que aún no se ha hecho referencia pero que no es menos importante que el resto: el **comparador de igualdad entre estados**, representado por la interfaz `IEqualityComparer<T>` del espacio de nombres `System.Collections.Generic` de la BCL. Este componente indica el modo en el que se determinará si dos estados son iguales o no. Ya se indicó que `Aran.Core` se incluyen dos implementaciones de esta interfaz. Una tercera la proporciona la propiedad `Default` de la clase `EqualityComparer<T>` del espacio de nombres `System.Collections.Generic` de la BCL. Esa implementación hace uso de los métodos definidos por `State` para realizar estas tareas, por lo que, al contrario de las ofrecidas en `Aran.Core`, tiene en consideración todos los hechos y variables.

Como viene siendo habitual, no todas las clases de este espacio de nombres han sido representadas en las ilustraciones. Destaca la ausencia de la excepción **`ResultNotFoundException`**, lanzada por el método `Search` cuando no encuentra solución. En otro orden de cosas, puede observarse en ambas ilustraciones la presencia del patrón ***Null Object***, representado mediante las clases `NullHeuristic` (que siempre devuelve el valor 0) y `NullAlgorithm` (que nunca devuelve solución). Por otro lado, el patrón ***Strategy*** está presente tanto en los propios algoritmos como en sus



componentes. Finalmente, puede verse cómo el **parámetro genérico T** está presente en la práctica totalidad de los tipos que conforman este espacio de nombres. Dicho parámetro representa el **tipo del estado** y convierte en genérico el espacio de nombres al completo.

4.4.7 Espacio de nombres Aran.Searching.Specialized

La siguiente ilustración muestra un diagrama simplificado de los tipos que conforman este espacio de nombres.

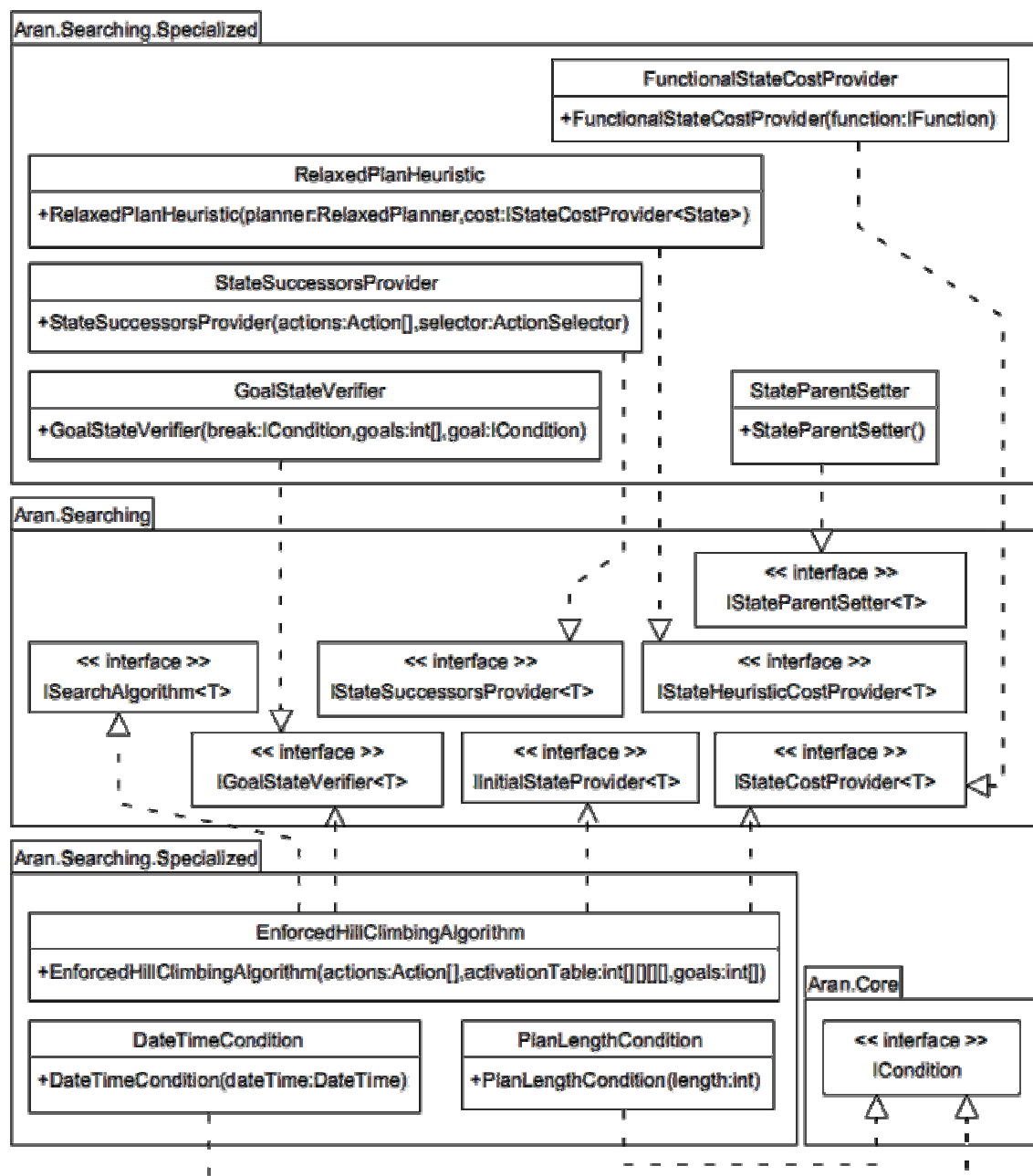


Ilustración 39: Diagrama de clases de Aran.Searching.Specialized



Como puede observarse, `Aran.Searching.Specialized` es una extensión de `Aran.Searching` que implementa los elementos vistos en ese espacio de nombres, especializándose en la búsqueda de planes a través de un espacio de estados. De hecho, y aunque no se refleja en el diagrama por simplicidad, en todas las relaciones de realización el parámetro genérico `T` se *instancia* como `State`.

Por esta vía se proporciona la función de coste (`FunctionalStateCostProvider`), la heurística (`RelaxedPlanHeuristic`), la de validación de meta (`GoalStateVerifier`), la de asignación de padre (`StateParentSetter`) y el generador de sucesores (`StateSuccessorsProvider`) que pueden emplearse con cualquiera de los algoritmos presentes en `Aran.Searching` cuando se utilicen en la búsqueda de estados.

En general, todos los tipos aquí contenidos reciben todos los elementos que necesitan en su constructor. Tal es el caso de **`FunctionalStateCostProvider`**, que se ocupa de adaptar la interfaz de una función de `Aran.Core` cualquiera, que recibe en su constructor, a la de la función de coste definida en `Aran.Searching`. Sigue, por tanto, el patrón de diseño *Adapter*. Siguiendo con la descripción de tipos, **`StateParentSetter`** se limita a asignar un estado al campo `Parent` de otro dado, actualizando, al tiempo, la longitud del plan (que es la longitud del plan del padre + 1). En cuanto a **`StateSuccessorsProvider`**, emplea la lista y el selector de acciones para devolver una lista de estados (en forma de `IEnumerable<State>`) sucesores de uno dado. Respecto a **`GoalStateVerifier`**, se ocupa de implementar el concepto de verificación de parada ya comentado, teniendo en cuenta una condición de ruptura, los hechos que tiene que contener la meta y la condición que ésta debe cumplir. Finalmente, **`RelaxedPlanHeuristic`** lleva a cabo la heurística del plan relajado que se expuso en el apartado conceptual, partiendo del plan devuelto por el planificador relajado y empleando una función de coste para evaluar los estados.

Además de definir los componentes expuestos, este espacio de nombres contiene la implementación del Enforced Hill Climbing, materializado en la clase **`EnforcedHillClimbingAlgorithm`**. Como el resto de algoritmos, tiene dependencias con algunos elementos definidos en `Aran.Core` que, por simplicidad, no han sido reflejadas en el diagrama. Concretamente, depende del proveedor de estado



inicial y las funciones de coste y validación de meta. Se incluye aquí y no en `Aran.Core` por ser un algoritmo específico. Su especificidad se refleja también en su independencia de otros componentes de ese espacio de nombres, como el generador de sucesores. Si se observa, por medio de su constructor recibe de forma directa los elementos necesarios para realizar las tareas de las que se ocupan esos componentes. Ello se debe a que las lleva a cabo de un modo específico difícilmente generalizable.

Para terminar, comentar que se incluyen aquí también las dos condiciones de parada. Por un lado, `DateTimeCondition` representa una parada por tiempo por medio de una condición que se hace cierta cuando se ha alcanzado una determinada fecha y hora. En cuanto a `PlanLengthCondition`, hace lo propio con una cierta longitud de plan.

4.4.8 Espacio de nombres `Aran.Translation`

La Ilustración 40 refleja los tipos principales de este espacio de nombres, como siempre, de forma simplificada para facilitar su legibilidad.

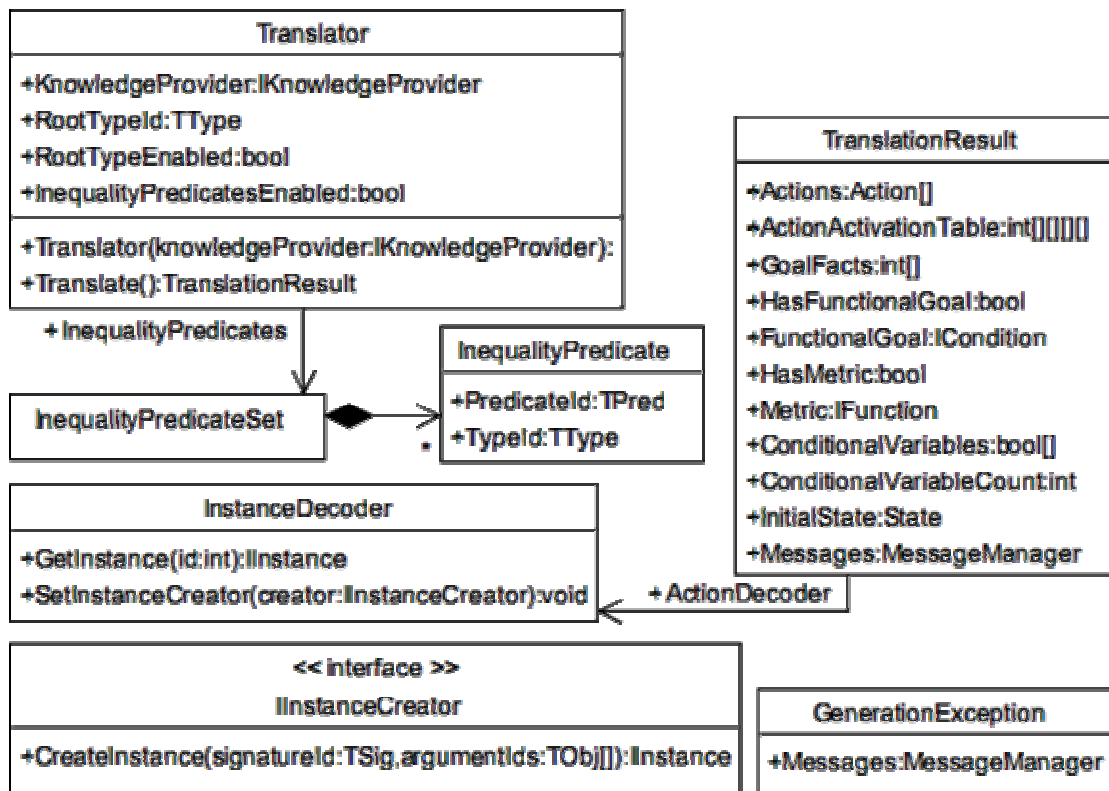


Ilustración 40: Diagrama de clases de `Aran.Translation`



Translator es el tipo principal, el encargado de implementar el proceso de traducción. Para ello, el traductor debe completarse proporcionándole un proveedor de conocimiento por medio de su constructor o de la propiedad correspondiente (las de **Translator** son las únicas de lectura/escritura, el resto sólo son de lectura). También con propiedades es posible asignar un tipo raíz y un conjunto de predicados de desigualdad, representado mediante la clase **InequalityPredicateSet**. Una vez establecidos estos parámetros, el método **Translate** se ocupa de iniciar la traducción, devolviendo el resultado generado en un objeto de tipo **TranslationResult** o lanzando una excepción de tipo **GenerationException** en caso de error. **TranslationResult** contiene toda la información necesaria para llevar a cabo la planificación por medio del algoritmo de búsqueda que se estime oportuno. En la Ilustración 40 se muestran los elementos más importantes de esa información, como la lista de acciones, la tabla de selección, la métrica, el estado inicial o las variables condicionales. Estas últimas son aquellas de las que depende alguna condición incluida en el dominio o el problema. Son, por tanto, las que deben tenerse en cuenta en la comparación de estados. Otro elemento resultante de la traducción es el decodificador de acciones, representado genéricamente mediante un decodificador de instancias, **InstanceDecoder**. El decodificador recibe un número de *instancia*, el identificador de acción, y devuelve la *instancia*, la acción. La *instancia* está representada mediante la interfaz **IInstance** de **Aran.Representation**. Esta interfaz, que no se comentó en su respectivo apartado, consiste en una versión de **IIidentifiedInstance** sin identificador de *instancia*. **InstanceDecoder** puede delegar la creación de *instancias* en un objeto de tipo **IInstanceCreator**. Aunque no se explicitan en el diagrama, todos estos tipos tienen los mismos parámetros genéricos que los empleados en **Aran.Representation**.

Si se observa, sea cual sea el resultado de invocar al método **Translate** (excepción o resultado), se obtiene un conjunto de mensajes representados mediante la clase **MessageManager**. Tal y como se muestra en la Ilustración 41, esta clase dispone de colecciones de mensajes según el tipo (de **Aran.Representation**) del origen. Por ejemplo, si el mensaje se originó a partir de información relativa a un hecho, se incluye en **FactMessages**. Las colecciones de mensajes están representadas por la clase **MessageCollection**, que se muestra en la Ilustración 42.

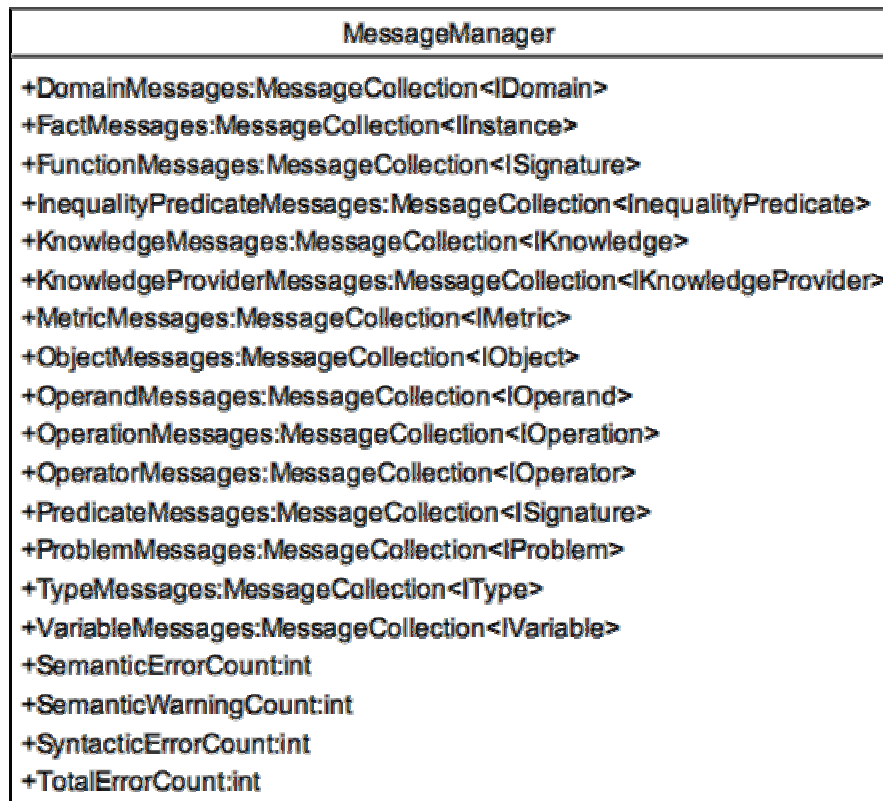


Ilustración 41: Diagrama de la clase `Aran.Translation.MessageManager`

Como puede verse, el parámetro genérico `T` de `MessageCollection` es el tipo del origen. Por ejemplo, en `TypeMessages`, `T` es *instanciado* como `IType`.

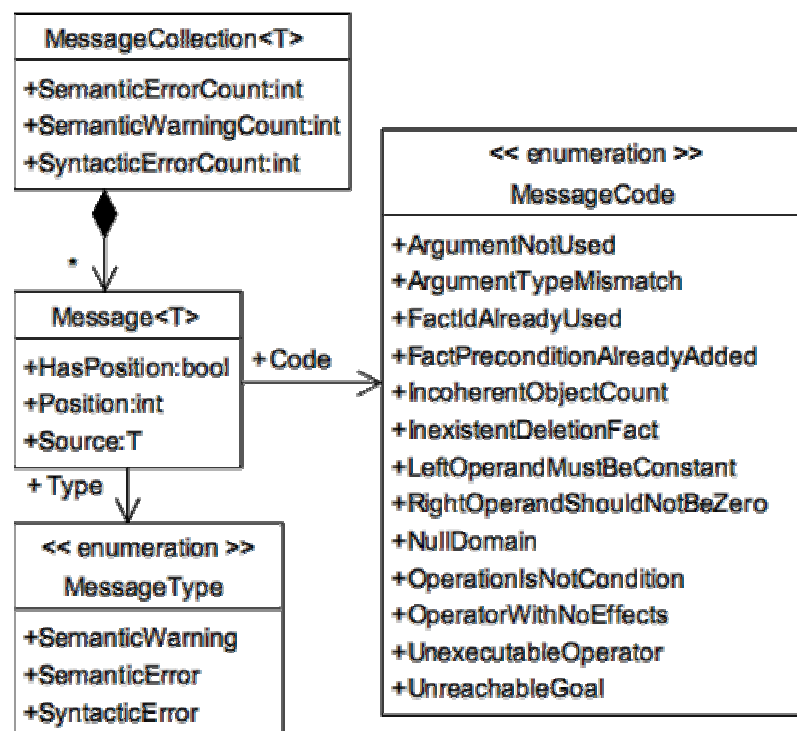


Ilustración 42: Diagrama de clases de mensajes en `Aran.Translation`



Tanto `MessageManager` como `MessageCollection` contienen estadísticas simples acerca del número de mensajes de cada tipo siendo las primeras globales y las segundas sólo referentes a una colección. Cada mensaje, que aparece en la Ilustración 42, representado en la clase **Message**, puede ser una advertencia semántica o un error, semántico o sintáctico, tal y como refleja la enumeración **MessageType**. Además del origen del mensaje, éste puede contener una posición que complete la información que puede extraerse del origen. Por ejemplo, si el mensaje hace referencia a un hecho de una lista de precondiciones, contendrá la posición del hecho en la lista. Finalmente, el código de mensaje, representado en la enumeración **MessageCode**, será el que dote de significado al conjunto. En la Ilustración 42 se recogen sólo algunos de los 160 códigos de mensaje existentes.

4.4.9 Espacio de nombres **Aran.Planning.Xml**

Los tipos más importantes de este espacio de nombres se muestran en la Ilustración 43. De entre ellos, el principal es **XmlPlanner**, que representa el planificador basado en ficheros XML. Posee una sencilla interfaz que contiene propiedades de lectura/escritura a través de las cuales se puede establecer la configuración. Así se establecen, entre otros, las rutas de los ficheros de entrada y salida, el tipo raíz, los predicados de desigualdad o las condiciones de parada. Todo ello siempre a un mayor nivel del visto hasta ahora. En este sentido, destaca la sencillez con la cual se configura la secuencia de algoritmos de búsqueda a utilizar, representada en la clase **XmlPlannerSearchAlgorithmSequence**, que permite gestionar los algoritmos partiendo únicamente de sus identificadores. Éstos, conocidos a priori, están representados en la enumeración **XmlPlannerSearchAlgorithmId**. La clase **XmlPlannerSearchAlgorithm** abstrae el algoritmo de búsqueda concreto y permite, por medio de sus propiedades, configurar los parámetros específicos de cada uno, como la anchura del haz o el peso de la heurística. Configurados los parámetros, el método `Plan` de `XmlPlanner` inicia el proceso de planificación, que puede cancelarse con el método `Cancel`. En caso de no encontrar un plan, se lanza una excepción del tipo **XmlPlanNotFoundException** indicando mediante mensajes contenidos en ella las causas del error.

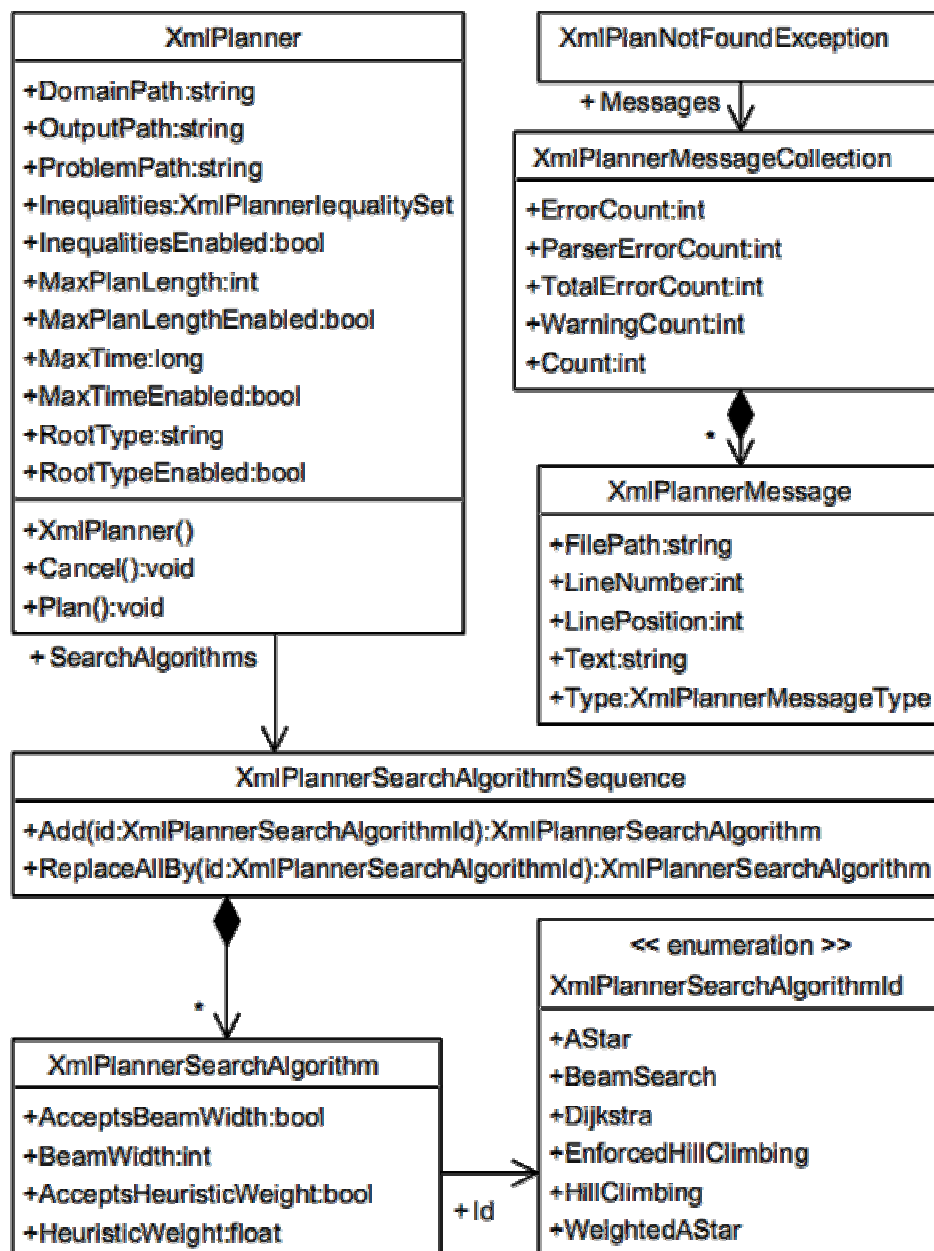


Ilustración 43: Diagrama de clases de `Aran.Planning.Xml`

Los mensajes están representados en la clase `XmlPlannerMessage` y agrupados en `XmlPlannerMessageCollection`. Para cada mensaje se devuelve información tal como la ruta del fichero al que se refiere, el número de línea o el texto explicativo, además del tipo de mensaje (error, advertencia...).

4.4.10 Espacio de nombres `Aran.Planning.Xml.Command`

Aunque este espacio de nombre contiene varios tipos, en verdad ninguno de ellos merece ser destacado más allá del representado en la Ilustración 44, pues son todos auxiliares de éste.

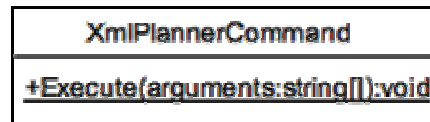


Ilustración 44: Diagrama de clases de `Aran.Planning.Xml.Command`

`XmlPlannerCommand` representa la interfaz en línea de comandos del planificador XML. Su método `Execute`, que hace las veces de punto de entrada del ejecutable, se ocupa de procesar los argumentos y llevar a cabo la funcionalidad relacionada con el comando. Aunque se trata de un tipo accesible públicamente, es poco probable que se utilice por la vía programática, pues en ese caso se antoja más factible un uso directo de la clase `XmlPlanner` de `Aran.Xml.Planning`.

4.5 Implementación

Los siguientes párrafos exponen algunos detalles de interés acerca de la fase de implementación de Aran. Al contrario de lo que sucede con los relativos al diseño, la estructuración de este apartado no se centrará en los distintos módulos que componen el sistema, sino en los problemas que han debido resolverse. La naturaleza dispar de dichos problemas resulta en un apartado heterogéneo que intentará ser, a pesar de todo, lo más claro posible en sus explicaciones de cara al entendimiento de esta fase.

4.5.1 Entorno de desarrollo

Para la construcción de Aran se ha dispuesto de un único entorno de desarrollo, cuya configuración *hardware* es la siguiente:

- Ordenador **Pentium IV 1,7 GHz** con 768 MB de RAM. Equipo sobre el cual se ha ejecutado todo el *software* de desarrollo.
- Videoconsola **Xbox 360** con disco duro. Conectada a la misma red de área local que el ordenador, para realizar tareas de despliegue y depuración.

Como puede verse, no es necesario un equipo de última generación para llevar a cabo un desarrollo como este. Cualquier laboratorio universitario actual cuenta con equipos preparados en este sentido. Respecto a la videoconsola, actualmente una nueva adquisición de un modelo como el mencionado requiere una inversión de 269,99 € en tiendas de venta al público. Ambas máquinas han sido empleadas como entorno de



pruebas de las versiones de Aran destinadas a sus respectivas arquitecturas. Pasando ahora al lado del *software*, la configuración empleada ha sido la siguiente:

- Microsoft **Windows XP SP 2**. Sistema operativo sobre el cual se ha ejecutado todo el *software* del equipo de desarrollo.
- Microsoft **Visual C# 2005 Express** Edition¹. Se ha empleado la versión en inglés, para que los textos de los ficheros generados por el IDE estén en dicho idioma.
- Microsoft **XNA Game Studio**² 2.0. El proyecto se comenzó con el Game Studio Express pero fue migrado a esta versión en la última fase del desarrollo.
- Microsoft **XSD Inference**³ 1.0. Herramienta capaz de generar un esquema XML a partir de uno o más ejemplos de ficheros XML que lo sigan.
- Microsoft **XSD Object Generator**⁴ 1.4.4.1. Utilidad capaz de generar, a partir de un esquema XML, el código fuente de un conjunto de clases que sirven para gestionar ficheros XML que lo siguen.
- Microsoft **FxCop**⁵ 1.35. Herramienta de análisis de código que avisa del incumplimiento de normas y recomendaciones recogidas en las guías de estilo, abarcando áreas tan dispares como el diseño, el rendimiento o la nomenclatura.
- Microsoft **Sandcastle**⁶ January 2008 Release. Conjunto de comandos para la generación de documentación de referencia.
- **Sandcastle Help File Builder**⁷ 1.6.0.4. Interfaz gráfica para Sandcastle.
- **Tortoise SVN**⁸ 1.4.3. Herramienta para el control de versiones de los ficheros fuente.
- **NAnt**⁹ 0.85. Utilidad para la automatización del proceso de construcción del *software*, de filosofía similar al comando `make` de Unix.
- **NAntContrib**¹⁰ 0.85. Conjunto de extensiones para NAnt que, entre otras cosas, posibilitan la interacción con el compilador de Visual Studio.
- **Inno Setup**¹ 5.2.2. Compilador de *scripts* para la generación de instaladores.

¹ Más información en <http://www.microsoft.com/spanish/msdn/vstudio/express/VCS/default.msp>

² Más información en <http://msdn2.microsoft.com/en-us/library/bb200104.aspx>

³ Más información en <http://msdn2.microsoft.com/en-us/xml/bb190622.aspx>

⁴ Más información en <http://msdn2.microsoft.com/en-us/xml/bb190622.aspx>

⁵ Más información en [http://msdn2.microsoft.com/en-us/library/bb429476\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/bb429476(VS.80).aspx)

⁶ Más información en <http://www.codeplex.com/Sandcastle>

⁷ Más información en <http://www.codeplex.com/SHFB>

⁸ Más información en <http://tortoisesvn.tigris.org>

⁹ Más información en <http://nant.sourceforge.net>

¹⁰ Más información en <http://nantcontrib.sourceforge.net>



- **ISTool²** 5.2.1. Interfaz gráfica para generar *scripts* de Inno Setup.
- **XNA Game Studio Connect**. *Software* de comunicación entre la videoconsola y el equipo de desarrollo, actualizado a 10/03/2008 [MSDN, 2008c].

Todo el *software* mencionado se refiere a instalaciones en el equipo de desarrollo, salvo el XNA Game Studio Connect, destinado a la videoconsola. Por otro lado, decir que todos los productos de Microsoft aquí listados pueden conseguirse sin coste alguno con fines académicos. El resto es *software* libre gratuito. Finalmente, añadir que también se han requerido las siguientes subscripciones a servicios, todas ellas necesarias para poder utilizar el XNA Game Studio Connect:

- Cuenta de **usuario Silver de la red Xbox LIVE³**.
- **Pertenencia al XNA Creators Club⁴**.
- **Conexión a Internet** de banda ancha.

Retomando el desembolso requerido por este entorno, recordar que la pertenencia al XNA Creators Club con fines académicos es gratuita por un periodo de 12 meses. Las cuentas de usuario Silver en Xbox LIVE no requieren ningún coste. Finalmente, la comunidad universitaria posee conexiones a Internet de banda ancha de acceso gratuito para sus miembros.

4.5.2 Ficheros fuente

La gran mayoría de las herramientas presentadas están destinadas a la gestión de ficheros fuente. Centrando la atención ahora en ellos, es preciso resaltar que una de las premisas en las que se ha basado la implementación de Aran ha sido la **separación de los ficheros fuente sobre la base de si dependen o no de una herramienta** en concreto. A este respecto, el repositorio de ficheros ha guardado, por un lado, los relativos a los proyectos de Visual Studio (de extensión `.sln` y `.csproj`), NAnt (`.build`), Sandcastle Help File Builder (`.shfb`) e Inno Setup (`.iss`), amén de otro auxiliares (como ficheros de imagen). Por otro lado se han almacenado los ficheros de código C# (`.cs`), los de recursos (`.resx`), los esquemas XML (`.xsd`) y los de pares de clave RSA (`.snk`). Estos últimos, los independientes de herramienta, son referenciados

¹ Más información en <http://www.jrsoftware.org>

² Más información en <http://www.istool.org>

³ Más información en <http://www.xbox.com/es-ES/live/Join.htm>

⁴ Más información en <http://creators.xna.com>



por los demás. Ello facilita, por ejemplo, una posible migración a otro IDE (como los de *software* libre SharpDevelop¹ o MonoDevelop²), siempre y cuando no se pretenda desarrollar para Xbox 360, algo sólo posible en Visual Studio a día de hoy. En algunos casos existen dos versiones de un mismo fichero cuando su formato es incompatible con las dos plataformas de destino, PC y Xbox 360. Tal es el caso, por ejemplo, de los proyectos de Visual Studio y los pares de clave RSA, como se explicará más adelante.

Aran será publicado bajo licencia **GNU Lesser General Public License 2.1**³. Todos los ficheros independientes de herramienta basados en texto incluyen una cabecera haciendo alusión a la misma.

En total hay más de 850 ficheros, todos con longitudes de línea pensadas para ser leídas en una **resolución** mínima de 1024x768 píxeles⁴, de los cuales el 90% son de **código C# 2.0** [González Seco, 2001]. Se ha hecho uso explícito de características presentes en dicha versión del lenguaje⁵, como los genéricos, los iteradores automáticos, las clases estáticas y la visibilidad asimétrica de propiedades. El código se ha escrito intentando seguir las **guías de estilo y buenas prácticas** más extendidas, con la ayuda añadida que supone la supervisión realizada a través de FxCop. La única recomendación que no se ha seguido de forma deliberada es la referente a la no utilización de prefijos en los nombres de campo para denotarlos como tales. Se considera que la anteposición de este tipo de prefijos (en este caso, el guión bajo “_”) facilita la legibilidad del código frente a alternativas como el uso del parámetro implícito `this`. No se ha empleado esta nomenclatura cuando los campos son de visibilidad pública o protegida y, por tanto, accesibles desde otros lenguajes de programación, ya que el guión bajo no cumple la CLS. Y es que **todos los tipos públicamente accesibles cumplen la CLS**, y como tal es indicado en sus metadatos mediante el atributo **CLSCompliant**⁶. Todos estos tipos están marcados también con el atributo **Serializable**⁷, indicando si son o no *serializables* (lo son la mayoría). Por último, todos los elementos que componen el código (clases, métodos, propiedades, campos...) están **documentados** (en inglés), independientemente de su nivel de visibilidad (pública, privada, protegida...), si bien los

¹ Más información en <http://www.icsharpcode.net/OpenSource/SD>

² Más información en <http://www.monodevelop.com>

³ Más información en <http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html>

⁴ Reconocida como la resolución más extendida en la actualidad

⁵ Más información en [http://msdn2.microsoft.com/en-us/library/7cz8t42e\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/7cz8t42e(VS.80).aspx)

⁶ Más información en [http://msdn2.microsoft.com/en-us/library/system.clscompliantattribute\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/system.clscompliantattribute(VS.80).aspx)

⁷ Más información en [http://msdn2.microsoft.com/en-us/library/system.serializeattribute\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/system.serializeattribute(VS.80).aspx)



elementos más importantes gozan de mayor detalle en las descripciones. Los comentarios no se limitan a la documentación, sino que explican también la implementación interna de los métodos más relevantes.

De todos los ficheros escritos en C#, aproximadamente el 70% se corresponde con el espacio de nombres **Aran.Functional**. Recuérdese que el diseño seguido en ese espacio de nombres era el de muchos tipos con poco código. Ahora bien, dadas sus características, salvo unos pocos, esos tipos no han sido codificados manualmente sino **generados de forma automática** mediante una sencilla herramienta desarrollada ex profeso. Por medio de un sistema de plantillas, la herramienta se ha ocupado de generar todas las condiciones, los efectos y las funciones que se obtienen al combinar los distintos operadores con los tipos posibles de sus operandos, con el posterior descarte de las combinaciones no válidas. Por simplicidad, los tipos generados por dicho sistema de plantillas **no hacen uso de la herencia**, a pesar de que podrían agruparse por similitud estructural.

Los ficheros del espacio de nombres `Aran.Functional` no son los únicos que han sido generados con ayuda de herramientas automáticas. Los tipos del espacio de nombres `Aran.Representation.Xml` son la cara pública del *parser* XML, pero el grueso de la implementación reside en un subespacio denominado **`Aran.Representation.Xml.Internal`** cuyos tipos han sido **generados de forma semiautomática**. De hecho, estos tipos son quienes en verdad implementan el *parser*, siendo los del espacio de nombres padre meros adaptadores a las interfaces expuestas en `Aran.Representation`. Su implementación se basa en las facilidades que el .NET Framework ofrece para la *serialización* XML. Siguiendo unas sencillas reglas de implementación es posible hacer tipos que pueden ser directamente *serializados/deserializados* en textos XML, incluso comprobando durante la *deserialización* que se cumple un cierto esquema. Los tipos del subespacio `Aran.Representation.Xml.Internal` mantienen una implementación minimalista que **obvia el principio de encapsulación**, por innecesario, y permite *deserializar* un árbol de objetos a partir del texto XML. Ese árbol es extraído por la clase `XmlParser` y “enfundado” por las clases de `Aran.Representation.Xml` para cumplir las interfaces requeridas. Esta implementación permitió un desarrollo rápido y sin dificultades pero tiene otros efectos menos deseables, aunque asumidos. Y es que el



rendimiento de algo así, tanto en memoria como en tiempo, no es el mejor posible. Por un lado, se están utilizando tipos de la BCL que, por su filosofía, están diseñados para ser lo más genéricos posibles, lo que conlleva la realización de determinadas tareas, incluso aunque no le sean necesarias al *parser* que requiere Aran. Es, por tanto, una implementación más lenta de lo que podría ser una *ad-hoc*. Por otro lado, el mencionado árbol supone la carga permanente en memoria de una estructura de datos que es, por genérica, relativamente pesada. De nuevo, guarda información que no es necesaria para el *parser* requerido por Aran, siendo más exigente en memoria y no permitiendo un control sobre su consumo que sí se tendría en una implementación *ad-hoc*. Rendimiento aparte, uno de los requisitos de la *serialización* XML en .NET es que los tipos a *serializar*, así como sus miembros principales, han de ser de **visibilidad pública**. Se trata de una pega de importancia en casos en los que, como éste, la *serialización* forma parte de una implementación que, por interna, nunca debería ser pública. Como se ha indicado, estos inconvenientes fueron aceptados en pro de un rápido desarrollo, teniendo en cuenta que, por su arquitectura, Aran puede recibir otras implementaciones futuras en lo que al sistema de representación específica se refiere. Incluso, esta misma implementación podría ser cambiada, eliminando estos tipos forzados a ser públicos. Con esto en mente, la documentación de dichos tipos indica expresamente que forman parte de una implementación interna y nunca deberían ser utilizados de forma directa. Por si fuera poco, **la documentación de referencia mantiene oculta la parte relativa a este espacio de nombres**, contribuyendo así a que no sea utilizado. Como contrapartida, la implementación de la **escritura de planes** no emplea la *serialización* XML. Por su sencillez, resultaba más rentable una implementación manual y de visibilidad privada, como así ha sido.

La implementación del espacio de nombres `Aran.Representation.Xml.Internal` fue ideada para ser, incluso, más rápida de desarrollar de lo que realmente fue. Se confió en la herramienta **XSD Inference** para evitar escribir explícitamente un **esquema XML** que describiera el formato de los ficheros. Sin embargo, aunque la herramienta ofreció resultados bastante buenos, no fueron suficientes. Su incapacidad para inferir definiciones recursivas (como que una operación puede contener otra operación) requirió que, finalmente, el esquema fuera desarrollado **a mano**. Con el esquema definido, se empleó la herramienta **XSD Object Generator** para obtener el conjunto de tipos necesarios para conformar el árbol en



memoria tras la *deserialización*. A pesar de tener también limitaciones, esta herramienta sí cumplió con sus expectativas, haciendo innecesaria la programación manual de esta parte. Esa implementación se mantuvo hasta la fase final de desarrollo, pudiendo cambiarla rápidamente cuantas veces se estimó necesario, gracias a la **automatización** de su generación. Cuando la implementación quedó estable, se realizó un **refinamiento manual** con el fin de hacer uso de genéricos y eliminar la encapsulación de la que disponía, dado su nulo aporte. No hay que olvidar que el *parser* nunca expone públicamente objetos de estos tipos que, por otro lado, son necesariamente públicos.

En contraposición a la generación automática de estos tipos, las clases de **gestión de recursos** han sido codificadas manualmente, a pesar de que Visual Studio incorpora capacidades de autogeneración a este respecto. Se ha preferido la codificación manual por varias razones. Principalmente porque la implementación que ofrece Visual Studio es, forzosamente, una clase estática (ni siquiera sigue el patrón *Singleton*). Ello implica que todas las instancias de, por ejemplo, el *parser* XML que se empleen en una aplicación han de estar localizadas al mismo lenguaje. Esta implementación no se considera apropiada, por lo que esta parte se ha realizado de forma manual y no estática. Esto ha tenido consecuencias en el tiempo de desarrollo, pues las clases de gestión de recursos contienen, por lo general, un abultado número de miembros. Por otro lado, la implementación de Visual Studio se limita a devolver los recursos sin modificación alguna. La implementación realizada, por contra, elabora algunos recursos antes de ser devueltos. Una última razón, que refuerza la idea de la **implementación manual**, es que así se mantiene el 100% del código fuera de la influencia de una herramienta concreta como es Visual Studio que, además, no es multiplataforma ni gratuita fuera del uso académico.

4.5.3 Optimización del rendimiento

Se ha hablado ya acerca de las deficiencias en rendimiento que lastra la implementación del *parser* XML que, a pesar de todo, han sido asumidas. Sin embargo, la implementación de Aran recoge otras actuaciones destinadas a la mejora del rendimiento en otras áreas.

Las medidas llevadas a cabo comienzan en el nivel más bajo: la **codificación**. La plataforma .NET y el lenguaje C# ofrecen una serie de elementos que, utilizados



convenientemente, favorecen un mejor rendimiento en tiempo de ejecución. Tal es el caso, por ejemplo, de los modificadores relacionados con las clases no heredables (**sealed**), las constantes (**const**) o las variables de sólo lectura (**readonly**). Esos modificadores se han empleado allí donde se ha estimado oportuno, no sólo para limitar el uso de los elementos que modifican sino, también, para permitir al compilador y al CLR realizar optimizaciones basadas en esas limitaciones. También se han realizado otras optimizaciones que requieren un mayor conocimiento de la implementación interna de las características de C# como, por ejemplo, el uso de **for en lugar de foreach cuando se ha requerido usar la variable de indización** (interna e inaccesible en el caso de `foreach`) para realizar operaciones adicionales a la iteración sin tener, por ello, que declarar y mantener una variable paralela de forma innecesaria.

Las optimizaciones de bajo nivel están, sobre todo, presentes **en el núcleo de Aran y en los módulos de búsqueda** pues se han considerado los más críticos en este sentido, por ser los responsables de llevar a cabo el mayor número de operaciones masivas (en las que los bucles son parte inherente). En estos módulos se ha **evitado también al máximo la delegación** (muy presente en los patrones de diseño) pues requiere, por lo general la invocación de un mayor número de métodos, afectando negativamente al rendimiento en tiempo. Por poner un ejemplo, cuando alguno de los elementos del núcleo no está presente (por ejemplo, no hay efecto en una acción), simplemente ese elemento se asigna a `null`, no utilizando el patrón *Null Object*, que sí se presenta en otros módulos de alto nivel, con menores requisitos de rendimiento. Los patrones, en general, suelen requerir también la **construcción/destrucción** de un mayor número de objetos, algo que también se ha **evitado en la medida de lo posible** por sus implicaciones negativas en la memoria, tanto en el consumo como en el tiempo necesario para gestionarla. De ahí que, por ejemplo, las clases relacionadas con la planificación relajada reutilicen el objeto que devuelven como resultado entre llamadas.

Siguiendo con los módulos de bajo nivel y alto rendimiento, otra medida llevada a cabo en ellos es la **asunción de la seguridad del entorno**, comentada con anterioridad. Asumiendo un entorno confiable se evitan comprobaciones reiterativas en los datos de entrada y aumenta el rendimiento en tiempo. Un ejemplo sencillo de esto es que el planificador relajado asume que en las listas de precondiciones no existen elementos repetidos. Con ello, se permite mantener un contador por cada acción que se



limita a aumentar cada vez que una precondition es alcanzada. Sin la asunción que realiza, sólo debería aumentarse el contador cuando la precondition fuera distinta a alguna ya alcanzada. El planificador relajado no realiza estas comprobaciones. Confía en que quien construyó la acción ya se ocupó de evitar estas situaciones.

Otro apartado importante en la optimización efectuada ha sido la **complejidad algorítmica**, que se ha intentado **reducir** en la medida de lo posible. Un ejemplo de este tipo de optimizaciones se encuentra en la generación del plan relajado respecto de lo explicado en [Hoffmann & Nebel, 2001]. En ese artículo se propone que la selección de la mejor acción de entre las que permiten alcanzar un hecho se realice durante la extracción del plan. Ello conllevaría recorrer todas las acciones que produjeron ese hecho. Sin embargo, la presencia del hecho es precisamente consecuencia de haber recorrido esas acciones anteriormente y, concretamente, sus listas de adiciones. Por tanto, la selección se ha movido a ese momento, el de la adición del hecho. Para ello se guarda en el hecho un enlace a la mejor de las acciones que lo añaden, siendo la que finalmente apunte ese enlace aquella que habría de ser seleccionada en el momento propuesto en [Hoffmann & Nebel, 2001]. Y, sin embargo, se evita ese doble recorrido al que se hacía mención al comienzo de las explicaciones.

Por supuesto, en la optimización, las **estructuras de datos** juegan un papel fundamental que se ha intentado potenciar. En primer lugar, el uso de los ya comentados **arrays de bits** permite un menor consumo de memoria tanto en la tabla de selección de acciones como, más importante aún, en los estados. Dejando de lado la implementación para almacenar valores booleanos, en general se ha hecho un **uso intenso del array**, por ser la estructura más eficiente de .NET, tanto en consumo de memoria como en tiempo de acceso directo. Por el contrario, se ha evitado el empleo de estructuras de datos generalistas, como las presentes en la BCL bajo el espacio de nombres `System.Collections`. Este tipo de estructuras sólo han sido utilizadas cuando su implementación no era rentable, como el caso de las **tablas hash** (representadas por la clase `Dictionary`, de `System.Collections.Generic`), aún a sabiendas de que, por generalistas, su consumo en tiempo y memoria es mayor que el que podría tener una implementación *ad-hoc*. Las tablas hash, en concreto, **se han utilizado con profusión en la implementación del traductor**, para mantener pares identificador-datos, indexados por identificador. Por ejemplo, para poder acceder a las estadísticas de un



predicado a partir del identificador de dicho predicado. Las tablas hash están **también presentes en las implementaciones de los algoritmos de búsqueda**, utilizadas para evitar visitar dos veces el mismo estado. Esto, que puede resultar un mecanismo natural para algoritmos de alto consumo en memoria, como el A*, resulta contraproducente en ese sentido para otros con bajo consumo, como Beam Search. Sin embargo, ha sido la mejor alternativa encontrada para solucionar un problema que, de no solventarse, puede dar lugar a bucles infinitos durante la búsqueda. Las tablas hash son las únicas de la BCL presentes en módulos de alto rendimiento, por no considerarse rentable su implementación *ad-hoc*, pero también se han utilizado con menos frecuencia las **listas** (representadas por la clase `List`, de `System.Collections.Generic`) en módulos de menor exigencia en rendimiento. Entre las estructuras implementadas ex profeso, destaca el **montón (*heap*) utilizado en los algoritmos Best First** para representar la cola de prioridad que supone la lista abierta. Esta estructura está considerada la más eficiente en tiempo para este cometido.

Otro aspecto tenido en cuenta es que en .NET la memoria de los objetos se libera de forma automática, pero nunca mientras el objeto en cuestión esté siendo referenciado. Así pues, una buena **gestión de memoria** requiere que los objetos estén referenciados por las estructuras de datos el tiempo mínimo imprescindible. Esto se ha llevado a la práctica especialmente en `Aran.Translation`, donde los datos asociados a los distintos tipos de elementos manejados por el subsistema de traducción están, en ocasiones, divididos “artificialmente” en dos clases cuando, si la gestión de memoria no fuera importante, podrían pertenecer a la misma. En general, se ha empleado el sufijo **Info** para marcar las clases empleadas para almacenar contenido temporal, sólo referenciado durante el proceso de traducción, mientras que aquellas de idéntico nombre pero sin dicho sufijo almacenan contenido persistente entre traducciones. Tal es el caso, por ejemplo, de las clases privadas `Predicate` y `PredicateInfo`. La información recogida en ambas clases es relativa a un predicado, pero se separa para poder liberar la memoria de lo almacenado en un objeto `PredicateInfo` cuanto antes y, como máximo, cuando finalice el proceso de traducción. Se sabe que su información ya no será nunca más necesaria, algo que no puede decirse de lo almacenado en `Predicate`.

Todas estas optimizaciones no han podido evitar una característica inherente a la plataforma .NET y, en general, aquellas con un sistema de ejecución similar, como Java.



Se trata del hecho de que **el rendimiento en tiempo es siempre inferior la primera vez que se ejecuta el código**, pues es entonces cuando tiene lugar su compilación a código máquina. Así, por ejemplo, si Aran se ejecuta varias veces dentro de un mismo proceso para resolver un problema idéntico, la primera vez tardará siempre más en resolverlo que las subsiguientes. La diferencia temporal dependerá de la relación entre el tiempo requerido por el proceso y el tamaño del código que ejecuta. Esta característica ha de tomarse en cuenta a la hora de evaluar el rendimiento de cualquier sistema .NET, y Aran en particular.

4.5.4 Diferencias entre versiones de PC y Xbox 360

Tal y como se ha venido comentando a lo largo de este documento, Aran es un sistema multiplataforma en la medida en que los componentes de .NET lo son. Como se ya se mencionó en su momento, el componente principal, el CLR, tiene diferentes implementaciones destinadas a distintas plataformas, como el PC, bajo Windows o Unix (Linux, Mac, Solaris...), la Xbox 360 o los dispositivos móviles (PDA, teléfonos móviles...). **Aran ha sido desarrollado y probado en PC bajo Windows y en Xbox 360, si bien debería funcionar también en PC bajo Unix** en la medida en que la implementación de .NET para esa plataforma esté libre de errores¹. Por otro lado, la producción de una versión para dispositivos móviles sería trivial, si bien no se ha realizado por no disponer del material adecuado para ello y, sobre todo, no considerarse prioritario.

Ahora bien, las versiones de Aran para las distintas plataformas no son idénticas. Conviene recordar que el .NET Framework no sólo es el CLR, sino que juegan un papel fundamental las bibliotecas que le acompañan. Como se recordará, la Xbox 360 emplea la versión Compact del .NET Framework, cuyas bibliotecas no sólo contienen menor funcionalidad sino que, además, están identificadas de un modo diferente. Esto, que hasta ahora no se ha tratado en esta memoria, es una de las principales diferencias entre versiones. En .NET, existen básicamente dos formas de dependencias² con bibliotecas externas. Sin entrar a valorar la otra opción, decir que la empleada para referenciar bibliotecas del .NET Framework es la más segura y restrictiva³. Este modo requiere que cada biblioteca esté identificada no sólo por su nombre sino, además, por otros

¹ Por ejemplo, la ejecución del sistema sobre Mono 1.9.1 no se realiza de forma totalmente correcta.

² Que, en la terminología de la plataforma, se denominan referencias

³ Más información en [http://msdn2.microsoft.com/en-us/library/wd40t7ad\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/wd40t7ad(VS.80).aspx)



elementos, entre los que destaca su número de versión y una firma digital. De este modo, los productos .NET, como Aran, sólo enlazan en tiempo de ejecución con aquellas bibliotecas que tienen un identificador idéntico al que se especificó durante la compilación. Explicado esto, el problema radica en que **las bibliotecas del .NET Framework y las de su versión Compact difieren en número de versión y firma**. Es por ello que, en principio, deben compilarse dos versiones diferentes del mismo producto (una para cada versión del .NET Framework), aun cuando se estén utilizando los mismos elementos de sus bibliotecas en ambas versiones. Según la documentación de Microsoft¹, es posible evitar esto, pues el .NET Framework tiene una directiva de seguridad que permite, por defecto, que se ejecuten productos compilados para la versión Compact. El mecanismo empleado consiste en realizar una correspondencia dinámica y transparente a usuario y desarrollador entre las versiones de las distintas bibliotecas. Este mismo mecanismo es el que emplea Mono² para proveer sus propias implementaciones de las bibliotecas de Microsoft, puesto que no tienen la misma firma que aquellas. Ahora bien, el problema añadido es que **las bibliotecas del Compact Framework de Xbox 360 difieren en identificación, de nuevo, con respecto a las del Compact Framework estándar** (distinta versión y firma). Se desconoce si es ese el motivo, pero lo cierto es que la versión de Aran compilada para Xbox 360 no funciona en Windows, ni viceversa. Así pues, **la solución ha sido** la que inicialmente se planteó: **dos versiones para dos Frameworks, PC y Xbox 360**.

Forzada la necesidad de dos versiones distintas, se ha aprovechado la circunstancia para ir algo más allá de las dependencias en lo que a diferencias se refiere. Por un lado, y aunque .NET no obliga a ello, Aran opta también por un mecanismo de identificación seguro. Por tanto, y para poder distinguirlas, **cada una de las dos versiones realizadas está firmada con un par de claves RSA distinto**. Ello implica que las aplicaciones de PC habrán de referenciar exactamente a la versión de PC de Aran y los juegos de Xbox 360 a la versión de Xbox 360, siendo ambas no intercambiables. Otro aspecto que diferencia ambas versiones es que **la de PC contiene metadatos referidos a los permisos** necesarios para ejecutar Aran (que son los mínimos). Esto no existe en la versión de Xbox 360 porque su BCL no dispone de los

¹ Más información en [http://msdn2.microsoft.com/en-us/library/4swehb60\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/4swehb60(VS.80).aspx)

² Más información en http://www.mono-project.com/Assemblies_and_the_GAC



elementos necesarios¹. Tampoco dispone de lo necesario para que las **excepciones** sean **serializables**², por lo que **sólo lo son en la versión de PC**. Una última diferencia destacable es que **el comando `aran.exe`**, resultante de la compilación del espacio de nombres `Aran.Planning.Xml.Command`, **no está presente en la versión de Xbox 360**, pues se considera innecesaria. Se estima que en esa plataforma la interfaz de manejo del planificador XML será siempre su API³, recogida en el espacio de nombres `Aran.Planning.Xml`, que sí está presente en las dos versiones.

Las diferencias existentes entre versiones son, en verdad, mínimas. Es por ello que se ha optado por mantener **el código de ambas dentro de los mismos ficheros fuente**, excluyendo el código incompatible con Xbox 360 por medio del uso de la **constante de compilación `XBOX360`**, que el IDE establece cuando compila para esa plataforma de destino. Por otro lado, el repositorio de ficheros fuente mantiene **algunos directorios y ficheros específicos de cada plataforma**. Tal es el caso, por ejemplo, de los pares de firmas, que se almacenan `AranPC.snk` para la versión PC y en `AranXbox360.snk` para la de Xbox 360. También se mantienen los directorios PC y Xbox360, que guardan respectivamente los ficheros de proyecto (de Visual Studio y otras herramientas) referentes a cada una de las plataformas. La misma metodología habría de seguirse si, en un futuro, se desease tener una versión para dispositivos móviles. Tan sólo es necesario saber que las bibliotecas a referenciar deberían ser las del Compact Framework estándar, no el de Xbox 360, y que la constante de compilación que emplea el IDE en esos casos es `PocketPC`. Por último, no habría diferencias de código entre la versión de Xbox 360 y la de dispositivos móviles, pues la primera no se sale nunca de los límites comunes al Compact Framework estándar.

4.5.5 Productos generados

El comando `aran.exe` ha aparecido ya en varias ocasiones a lo largo de esta memoria pero no es, ni mucho menos, el único producto de este proyecto. De hecho, **cada espacio de nombres ha dado lugar a un ensamblado**⁴, tal y como se refleja en la siguiente tabla.

¹ Su espacio de nombres `System.Security.Permissions` está prácticamente vacío

² No existe el constructor protegido `Exception(SerializationInfo, StreamingContext)`

³ Siglas de “Application Programming Interface”

⁴ En .NET, los ensamblados son los ficheros binarios que contienen código ejecutable por el CLR



Espacio de nombres	Ensamblado
Aran.Representation	Aran.Representation.dll
Aran.Representation.Xml	Aran.Representation.Xml.dll
Aran.Core	Aran.Core.dll
Aran.Functional	Aran.Functional.dll
Aran.Searching	Aran.Searching.dll
Aran.Searching.Specialized	Aran.Searching.Specialized.dll
Aran.Translation	Aran.Translation.dll
Aran.Planning.Xml	Aran.Planning.Xml.dll
Aran.Planning.Xml.Command	aran.exe

Tabla 5: Correspondencias entre espacios de nombres y ensamblados

Como puede verse, salvo `aran.exe`, todos los ensamblados tienen el mismo nombre que el espacio de nombres que contienen, siguiendo así las guías de estilo de la plataforma. El espacio de nombres `Aran.Representation.Xml.Internal` se incluye en la biblioteca `Aran.Representation.Xml.dll`, por tratarse de una agrupación lógica interna, no destinada a su uso explícito. El caso del comando `aran.exe` es también especial, pues debe tener un nombre sencillo, pensado para agilizar su uso, de ahí que no concuerde con el espacio de nombres que alberga.

Otro producto generado ha sido la documentación de referencia. Ésta es extraída de los comentarios presentes en los ficheros fuente por parte del IDE utilizado para la compilación, dando lugar a ficheros XML del mismo nombre que los ensamblados que documentan (por ejemplo, `Aran.Core.xml` documenta `Aran.Core.dll`). Ubicados en el mismo directorio que dichos ensamblados, estos ficheros XML proporcionan al IDE **documentación inteligente asociada al sistema de autocompleción de código**¹, algo que se agradece enormemente durante el desarrollo. Dichos ficheros XML dan lugar también a **documentación de referencia en forma de páginas HTML**, preparada para ser leída por humanos (el formato de los XML es ilegible). Para ello han sido procesados por una utilidad al efecto, como es Sandcastle.

¹ Cuya implementación en Visual Studio se denomina *IntelliSense*

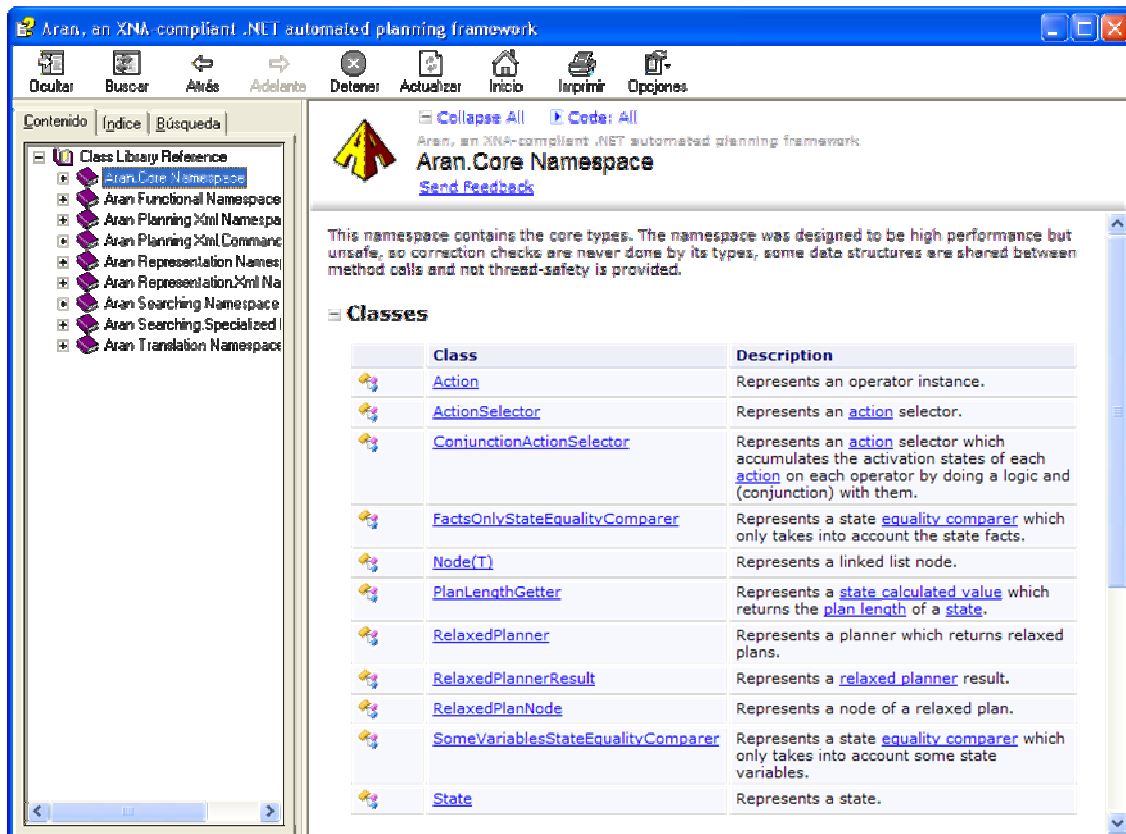


Ilustración 45: Documentación de referencia

La documentación de referencia **sólo incluye las partes que podrán ser utilizadas por los desarrolladores**, lo cual se traduce únicamente en elementos de visibilidad pública o protegida. Como ya se ha comentado en otra ocasión, el espacio de nombres `Aran.Representation.Xml.Internal` tampoco se incluye en esta documentación en formato HTML. Otro aspecto a tener en cuenta es que, dadas las escasas diferencias entre plataformas de destino, **sólo existe una versión** de la documentación. Simplemente, cuando algún elemento no está presente en la versión de Xbox 360 se indica apropiadamente en los comentarios.

El último de los productos generados es un **instalador**. Dicho programa se ocupa del proceso de **implantación de los productos** ya mencionados (ensamblados, ficheros de autocompleción, documentación de referencia...) **en un PC con SO Windows** de forma sencilla. Incluye todos los ensamblados, tanto de PC como de Xbox 360, sirviendo al mismo tiempo como instalador de programa (el comando `aran.exe`) y de kit de desarrollo.

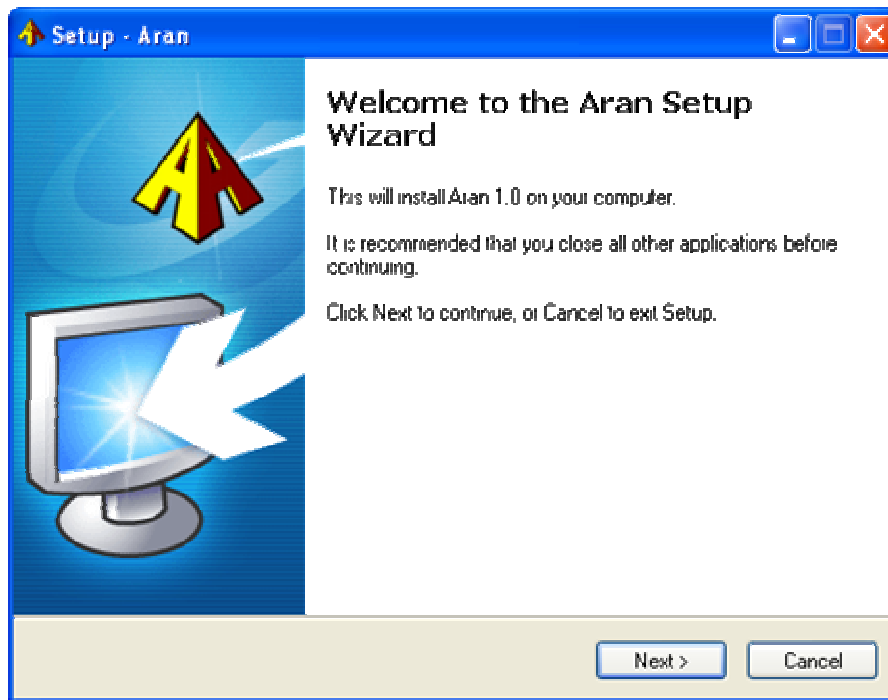


Ilustración 46: Instalador para Windows

El instalador da la opción de **hacer visible el comando `aran.exe` a todo el SO** y establece la configuración necesaria para que **las bibliotecas de PC y Xbox 360 sean accesibles intuitivamente desde el IDE**, facilitando así su uso por parte de los desarrolladores. De este modo, cuando el desarrollador esté construyendo un sistema para PC el IDE sólo le ofrecerá las bibliotecas de la versión PC. Por el contrario, cuando se ocupe de un juego de Xbox 360 tendrá sólo accesibles las de la versión de esta videoconsola. Por último, el instalador ha sido **desarrollado con el fin de permitir la convivencia futura de distintas versiones** de las mismas bibliotecas en un mismo PC (pudiendo así realizar desarrollos paralelos dependientes de diferentes versiones), algo común en herramientas avanzadas, como el propio XNA Game Studio.

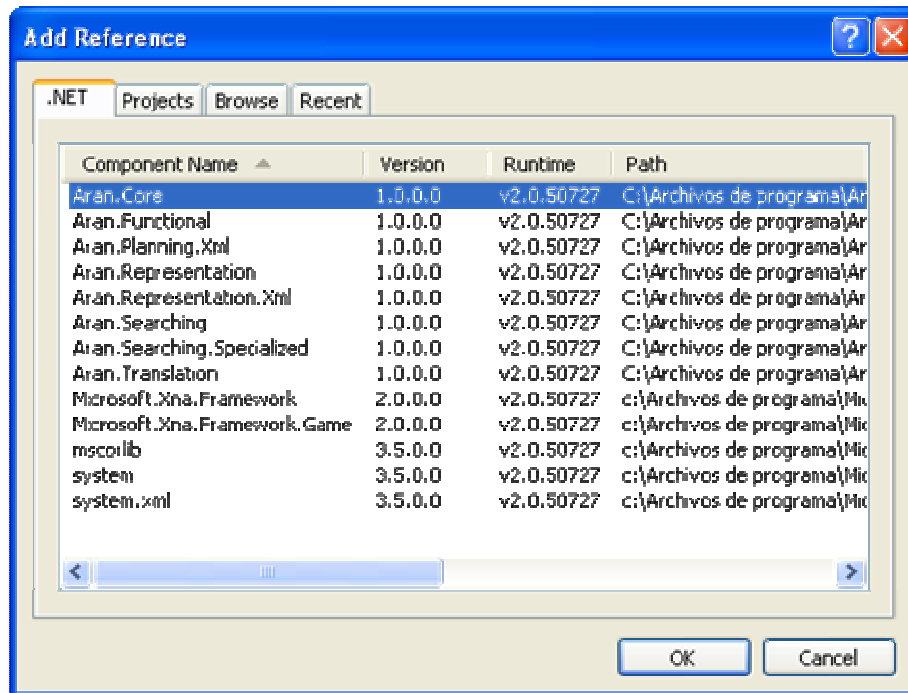


Ilustración 47: Integración de Aran con Visual Studio

Tanto los elementos expuestos en este apartado como el código fuente serán publicados en CodePlex bajo el URL <http://www.codeplex.com/Aran>. De cara a su identificación como producto, Aran dispone de **logotipo propio**, presente en todas las páginas de esta memoria y en el instalador. Será utilizado también en la página de publicación, cuando ésta tenga lugar (tras la presentación y defensa pública del proyecto).



Ilustración 48: Logotipo de Aran

CAPÍTULO 5

RESULTADOS

*Al final,
no os preguntarán qué habéis sabido,
sino qué habéis hecho*



5 Resultados

Explicado el proceso de construcción de Aran, es turno ahora de evaluar los resultados de la puesta en funcionamiento del sistema. En este apartado se llevará a cabo un **análisis del comportamiento observado sobre un entorno de pruebas** desarrollado para la ocasión.

5.1 Entorno de pruebas

Una de las principales ventajas con las que cuentan las videoconsolas con respecto al PC es que sus especificaciones técnicas permanecen inalteradas durante todo su ciclo de vida, independientemente de si los interiores del *hardware* sufren o no revisiones durante ese tiempo, como suele ser habitual. La **Xbox 360** no es un caso aparte en ese sentido, por lo que **el entorno de pruebas se ha construido sobre ella** con el fin de que pueda ser fácilmente reproducible en términos de rendimiento. En verdad, XNA es un entorno fuertemente dependiente del *software* (principalmente, de la implementación del CLR¹) y éste es actualizable en toda videoconsola moderna, como es la Xbox 360, por lo que ni siquiera asegurando un mismo *hardware* se asegura idéntico rendimiento. En cualquier caso, esto también sucede en PC, por lo que la videoconsola sigue siendo la mejor opción.

La forma de proceder consiste en ejecutar varios problemas sobre el motor principal de un juego desarrollado explícitamente para presentar Aran al concurso anual **Dream Build Play**², de Microsoft, cuya temática en la edición 2008, bajo el título “Silicon Minds”, era la Inteligencia Artificial. Aunque finalmente no pudo ser acabado a tiempo para ser presentado, sus características le convierten, de hecho, en un dominio de planificación gráfico, editor de problemas y visualizador de soluciones incluidos. Es por ello que se ha considerado oportuno reutilizarlo.

Es de reseñar el hecho de que este entorno sea el resultado de la **aplicación práctica** de Aran, pues se diseñó para el concurso, no para estas pruebas. Esto, que puede parecer una sutileza, no lo es. Siendo práctico, los límites al dominio fueron impuestos por el diseño del juego, no por las características del sistema que se quieren

¹ En la actual implementación del .NET Compact Framework para Xbox 360, el recolector de basura es más simple e ineficiente que el de la versión de PC [MSDN, 2006c] y las operaciones en coma flotante no están optimizadas, a pesar de hacer uso de *hardware* específico de la consola [MSDN, 2006d].

² Más información en [http:// www.dreambuildplay.com](http://www.dreambuildplay.com)

probar, como sucede en otros casos. Ello ayudó enormemente a la depuración del proyecto final, sacando a relucir bastantes fallos no detectados hasta entonces.

Un último apunte de interés acerca del entorno de pruebas es que el motor del videojuego hace uso de **un único núcleo** de los 3 de los que dispone la CPU de la Xbox 360, empleando también **un único thread** de los dos disponibles en el *hardware* del mismo.

5.2 Descripción del dominio

Tal y como muestra la Ilustración 49, el dominio consiste en un tablero de **baldosas** a través de las cuales puede moverse un personaje con el fin de lograr su objetivo: recopilar todas las **llaves** presentes en el mapa en el menor tiempo posible. El personaje puede **moverse de una baldosa origen a otra de destino** siempre y cuando origen y destino sean adyacentes vertical u horizontalmente, pero no diagonalmente. En total, una baldosa puede llegar a tener un máximo de 4 vecinas. No existe sentido de circulación: el personaje puede volver a una baldosa origen desde una de destino.

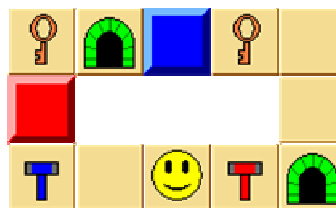


Ilustración 49: Ejemplo de problema para el dominio de pruebas

El problema se complica con la presencia de **bloques** de colores que impiden la ocupación de la baldosa en la que se encuentran por parte del personaje. Para poder ocupar dicha posición, el personaje deberá **destruir el bloque** previamente, utilizando un **martillo** del mismo color que éste, y estando posicionado en una baldosa adyacente a la bloqueada. Los martillos se consumen cuando son utilizados, por lo que cada uno sólo puede desbloquear una baldosa. Los martillos se encuentran esparcidos por el mapa, por lo que el personaje tendrá que recogerlos previamente antes de poder usarlos. No hay límite en el número de martillos que el personaje puede llevar al mismo tiempo y puede haber varios martillos del mismo color. Asimismo, puede haber varios bloques del mismo color y no tiene por qué haber el mismo número de martillos que de bloques para un cierto color.



Siguiendo con los colores, el último tipo de elemento presente es el portal. Los **portales** pueden verse como un tipo especial de baldosa que permite al personaje, además de moverse a alguna adyacente, tele transportarse a otra que tenga un portal del mismo color. La comunicación entre portales es bidireccional, esto es, si el personaje entra por una de las bocas sale por la otra, independientemente de cuál de ellas sea origen o destino. Sólo puede haber dos portales de un mismo color.

Tanto el personaje como los martillos tienen **peso**. Cada vez que el personaje coge un martillo, aumenta su peso tantas unidades como pese el martillo. Cada vez que utiliza un martillo y, por tanto, lo consume, se reduce su peso en tantas unidades como pesaba el martillo. La **velocidad** del personaje es inversamente proporcional a su peso. Le cuesta moverse de una baldosa a otra tantas unidades de tiempo como unidades pese en ese momento. La tele transportación de un portal a otro cuesta siempre una unidad de tiempo, independientemente de la distancia entre los portales. Por otro lado, los bloques tienen **resistencia**. Desbloquear una baldosa ocupada por un bloque cuesta tantas unidades de tiempo como unidades de resistencia tenga ese bloque. Finalmente, **coger una llave o un martillo** cuesta 1 unidad de tiempo. Todas las unidades se miden en números enteros de 0 a 5.

En “Anexo B: Ejemplos de utilización del lenguaje” puede verse una implementación de este dominio en formato XML, así como el escenario representado en la ilustración. Sin embargo, como se ha venido insistiendo a lo largo de esta memoria, Aran está diseñado para admitir otras implementaciones del **subistema de representación**. Aprovechando esta característica, el entorno de pruebas implementa un **modelo de objetos en memoria** totalmente dependiente del juego. Aunque la implementación no está optimizada, permite aumentar con creces el rendimiento respecto al del *parser* XML, al tiempo que el juego maneja de forma directa los objetos del modelo, pues son parte inherente de su arquitectura, acercándose al resultado que podría obtenerse integrando Aran en un videojuego.

5.3 Descripción de los problemas

Llegado a este punto, y aunque no se puede asegurar al 100%, se le presupone al sistema un buen funcionamiento en la medida en que éste ha sido probado y depurado. No es, por tanto, objetivo de estas pruebas verificar que el sistema hace lo que debe, sino comprobar su nivel de rendimiento ante distintas situaciones. Para ello, se le ha

sometido a una **batería de pruebas de dificultad diversa** con cuyos resultados poder evaluar el sistema desde diferentes puntos de vista. Hay que destacar que los aquí mostrados no son los únicos problemas evaluados, sino un conjunto representativo de los mismos. En todos los casos existe solución y, salvo que se indique lo contrario, el personaje parte con un **peso** de 1 unidad, los martillos no pesan nada y todos los bloques tienen 1 unidad de **resistencia**. El empleo de estos valores simplifica el entendimiento de los resultados y, al tiempo, no evita que el sistema deba realizar las operaciones numéricas pertinentes (con la correspondiente penalización en tiempo). A continuación se realiza una muestra gráfica y una descripción para cada caso de prueba:

- **Problema 1:** Se trata del problema más sencillo, bien formado, que puede plantearse con el dominio dado. Con él se pretende obtener información que permita estimar una cota mínima del coste en rendimiento que supone la utilización de Aran en un videojuego. Este problema tiene un único, y obvio, plan posible.



Ilustración 50: Captura de pantalla del problema 1

- **Problema 2:** Con este caso se pretende aumentar ligeramente la complejidad respecto al primero planteado incrementando, al tiempo, la cantidad de planes posibles. El objetivo es observar las variaciones en el rendimiento del sistema dentro de cotas mínimas de complejidad.



Ilustración 51: Captura de pantalla del problema 2

- **Problema 3:** En esta ocasión se vuelve a aumentar levemente la complejidad, añadiendo una meta más e incrementando tanto la longitud mínima de los planes posibles como su cantidad. Se pretende con ello complementar la información obtenida con los casos previos acerca del rendimiento en cotas de complejidad bajas.



Ilustración 52: Captura de pantalla del problema 3



- **Problema 4:** Aquí se vuelve a aumentar el número de elementos a manejar y, con ellos, la longitud mínima de los planes solución. Se trata, a su vez, del primer problema de esta batería para el cual un humano necesitaría realizar cálculos adicionales de coste a fin de determinar qué plan elegir. A pesar de todo, este problema continúa en la línea de baja complejidad de los anteriores.

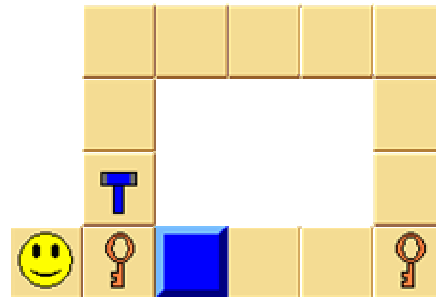


Ilustración 53: Captura de pantalla del problema 4

- **Problema 5:** Se trata de una variación del problema 4 a la que simplemente se le ha añadido una meta adicional para comprobar en qué medida afecta a los resultados.

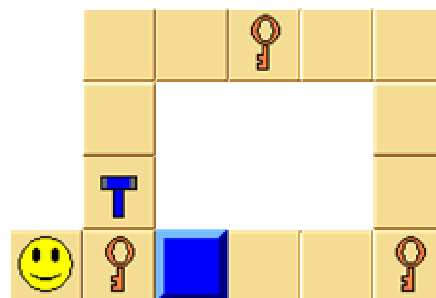


Ilustración 54: Captura de pantalla del problema 5

- **Problema 6:** Este es otra variación alternativa del problema 4 en el cual, como sucedía en el problema 5, se ha añadido un único elemento. La diferencia es que esta adición no modifica el número de metas. Se pretende con ello poder comparar los resultados obtenidos en ambas modificaciones.

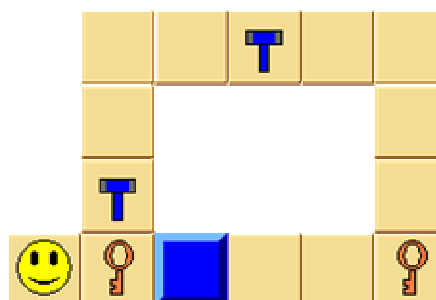


Ilustración 55: Captura de pantalla del problema 6

- **Problema 7:** Este problema supone un salto en el nivel de complejidad respecto a los anteriores. El número de planes posibles es notablemente más numeroso, así como su longitud siendo, de hecho, el primer problema de esta batería para el cual un humano no podría dar una solución rápida, como sí ocurría en los casos anteriores.

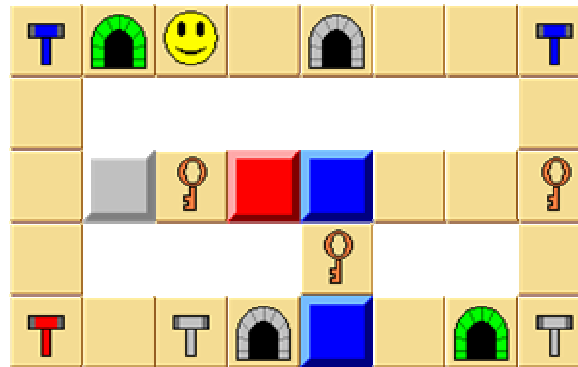


Ilustración 56: Captura de pantalla del problema 7

- **Problema 8:** Nuevamente, este problema es una variación de uno previo, el 7. A diferencia de las anteriores, en esta modificación no se añade ningún elemento adicional, sino que se modifican algunos valores numéricos. Concretamente, se penaliza el uso del martillo azul más a la derecha dándole un peso de 5 unidades, al tiempo que se facilita la destrucción de los bloques centrales, otorgándoles una resistencia de 0 unidades. Para mayor claridad, en la Ilustración 57 se superponen los nuevos valores sobre el problema modificado. Con este caso se persigue observar la respuesta del sistema ante variaciones en los recursos numéricos que complican su resolución óptima por parte de humanos.



Ilustración 57: Captura de pantalla del problema 8



- **Problema 9:** Con este problema se vuelve a producir un salto de complejidad notorio, haciendo prácticamente imposible su resolución óptima por parte de humanos en un tiempo razonable. Se trata de un caso expresamente creado para tantear los límites del sistema.



Ilustración 58: Captura de pantalla del problema 9

- **Problema 10:** A pesar de su apariencia, este último problema está específicamente ideado para poner en apuros al sistema, forzándole a la ineficiencia con algo para lo que un humano no tardaría en encontrar una solución óptima. Con él, se persigue evidenciar la existencia de carencias en las capacidades del sistema relacionadas, eso sí, con casos especiales como este.



Ilustración 59: Captura de pantalla del problema 10



5.4 Presentación de los datos obtenidos

En este apartado se mostrará un conjunto de tablas de datos recogidos durante el proceso de pruebas. Puesto que los datos relativos a la **traducción** son independientes del algoritmo de **búsqueda** utilizado, se ha creído conveniente separar ambas presentaciones. De este modo, se muestran en primer lugar los resultados obtenidos en la traducción de cada uno de los problemas y, posteriormente, se hace lo propio con los obtenidos en la búsqueda de una solución para los mismos.

Los resultados de traducción recogen el tiempo empleado por la Xbox 360 para traducir el problema, tanto haciendo uso del subsistema de representación específico del juego (fila **Trad.**) como el *parser* XML (**Trad. XML**). Por otro lado, se muestra la cantidad de elementos presentes en la representación del problema: el número de objetos (**Objetos**), el número de hechos iniciales (**H. iniciales**), el número de hechos meta (**H. meta**) y el número de variables (**Variables**). También se indican datos relativos a la representación del estado resultante de la traducción: tanto el número de hechos (**H. estado**) como de variables (**V. estado**) que forman parte del mismo, como su tamaño en memoria (**T. estado**). Finalmente, se indica el número de acciones que resultan de la traducción del problema (**Acciones**), cuánto ocupan en memoria (**T. acciones**) y cuánto la tabla de selección de las mismas (**T. tabla sel.**).

En cuanto a los resultados de búsqueda, éstos reflejan, para cada algoritmo, el tiempo empleado (columna **Tiempo**), el número de nodos generados (**N. gen.**), el número de nodos expandidos (**N. exp.**), si encontró o no un plan solución (**Sol.**), la longitud del plan (**Lon.**), el coste de ejecutar el mismo (**Cos.**) y si se trata o no de un resultado de coste óptimo (**Opt.**). Sirva de aclaración que el número de nodos generados es la cantidad de nodos del árbol de búsqueda que genera cada algoritmo, mientras que el número de nodos expandidos es la cantidad de nodos generados para los cuales se han generado hijos.

Entrando ahora en el formato de los datos, decir que todos los **tiempos** están expresados de la forma **MM:ss:mm**, donde **MM** son minutos, **ss** segundos y **mmm** milisegundos. En cuanto a las cantidades de **memoria**, se expresan en bytes si son menores a un KB y en KB en caso contrario, indicando siempre la unidad de medida. El resto de datos están medidos en **unidades** de sus respectivos ámbitos. Por ejemplo, la



longitud del plan en número de acciones y el coste del mismo en unidades de tiempo del juego. Cuando un dato es desconocido por alguna razón, se indica con la cadena “--”.

Como punto final antes de dar paso a los datos, se comentará el modo en que han sido realizadas las mediciones. En primer lugar, **las ejecuciones se han hecho siempre en modo de no depuración**. Esto es importante, porque de hacerse en modo depuración, el JIT no aplica optimizaciones y los tiempos resultantes son mayores de los que se obtendrían en una ejecución sobre la Xbox 360 de un usuario final¹. En segundo lugar, **los tiempos han sido medidos de forma aislada**, esto es, midiendo exactamente el intervalo que va desde el inicio de la operación hasta el final de la misma. Por tanto, no se ven influidos por ninguna característica propia del juego más allá del dominio que representa y de la implementación de su sistema de representación. Como ya se ha comentado en alguna otra ocasión, la acción del JIT penaliza el tiempo empleado en ejecutar cualquier porción de código por primera vez. Se entiende que en un juego el código será ejecutado siempre más de una vez, por lo que en todos los casos los tiempos reflejan **ejecuciones posteriores a la primera**. Además, para obtener datos más precisos se han llevado a cabo 10 ejecuciones por problema y **se ha obtenido la media** de todos ellos. En cuanto a los **tamaños en memoria**, se han calculado teniendo en cuenta la estructura de los distintos tipos afectados. Por ejemplo, las N variables del estado ocupan $\text{IntPtrSize} + (N \times \text{FloatSize})$ bytes, siendo $\text{FloatSize} = 4$ (en todos los CLR) e $\text{IntPtrSize} = 4$ (en el CLR la Xbox 360, en otros puede ser distinto). Obsérvese que esto implica que en casos como este existirá siempre un tamaño mínimo (en este caso IntPtrSize), independientemente de la cantidad de datos manejados. Eso hace que, por ejemplo, el tamaño mínimo de un estado sea de 20 bytes y el de una acción de 28.

Dicho esto, a continuación se muestran los datos resultantes de la **traducción** de los distintos problemas.

¹ Más información en <http://msdn2.microsoft.com/en-us/library/ms241594.aspx>



Problema	1	2	3	4	5
Trad.	00:00:005	00:00:007	00:00:012	00:00:024	00:00:024
Trad. XML	00:00:073	00:00:083	00:00:093	00:00:102	00:00:104
Objetos	3	8	14	19	20
H. iniciales	4	14	25	37	38
H. meta	1	1	2	2	3
Variables	2	4	6	4	4
H. estado	4	11	20	23	25
V. estado	2	2	2	2	2
T. estado	32 bytes	32 bytes	32 bytes	32 bytes	32 bytes
Acciones	3	12	22	35	36
T. acciones	144 bytes	632 bytes	1, 14 KB	1, 7 KB	1, 74 KB
T. tabla sel.	144 bytes	1, 12 KB	2, 03 KB	2, 34 KB	2, 54 KB

Tabla 6: Resultados de traducción para los problemas 1 al 5

Problema	6	7	8	9	10
Trad.	00:00:025	00:00:094	00:00:092	00:00:250	00:00:162
Trad. XML	00:00:104	00:00:166	00:00:165	00:00:253	00:00:173
Objetos	20	40	40	65	49
H. iniciales	39	88	88	140	105
H. meta	2	3	3	7	24
Variables	5	11	11	13	2
H. estado	25	53	53	82	73
V. estado	2	2	2	2	2
T. estado	32 bytes	36 bytes	36 bytes	40 bytes	40 bytes
Acciones	38	92	92	139	104
T. acciones	1, 88 KB	4, 73 KB	4, 73 KB	6, 91 KB	4, 88 KB
T. tabla sel.	2, 54 KB	6, 21 KB	6, 21 KB	10, 29 KB	3, 14 KB

Tabla 7: Resultados de traducción para los problemas 6 al 10

Mostrados los resultados de las traducciones de los diferentes problemas, seguidamente se hace lo propio con los obtenidos en la **búsqueda** de soluciones para los mismos. En ellos, los nombres de algunos algoritmos se han abreviado: HC (Hill Climbing), EHC (Enforced Hill Climbing), BS (Beam Search), WA* (Weighted A*). Además, en el caso de los paramétricos se ofrece entre paréntesis el valor del parámetro (anchura del haz en Beam Search y peso de la heurística en Weighted A*).



Prob. 1	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:000	3	2	sí	2	2	sí
EHC	00:00:000	2	2	sí	2	2	sí
BS (2)	00:00:000	3	2	sí	2	2	sí
BS (4)	00:00:000	3	2	sí	2	2	sí
BS (6)	00:00:000	3	2	sí	2	2	sí
BS (8)	00:00:000	3	2	sí	2	2	sí
BS (10)	00:00:000	3	2	sí	2	2	sí
A*	00:00:000	3	2	sí	2	2	sí
WA* (1,3)	00:00:000	3	2	sí	2	2	sí
WA* (1,6)	00:00:000	3	2	sí	2	2	sí
WA* (2)	00:00:000	3	2	sí	2	2	sí
WA* (3)	00:00:000	3	2	sí	2	2	sí
WA* (4)	00:00:000	3	2	sí	2	2	sí
WA* (5)	00:00:000	3	2	sí	2	2	sí
Dijkstra	00:00:000	3	2	sí	2	2	sí

Tabla 8: Resultados de búsqueda para el problema 1

Prob. 2	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:001	15	8	sí	8	8	sí
EHC	00:00:001	11	9	sí	8	8	sí
BS (2)	00:00:001	18	10	sí	8	8	sí
BS (4)	00:00:001	18	10	sí	8	8	sí
BS (6)	00:00:001	18	10	sí	8	8	sí
BS (8)	00:00:001	18	10	sí	8	8	sí
BS (10)	00:00:001	18	10	sí	8	8	sí
A*	00:00:001	16	10	sí	8	8	sí
WA* (1,3)	00:00:001	15	9	sí	8	8	sí
WA* (1,6)	00:00:001	15	9	sí	8	8	sí
WA* (2)	00:00:001	15	9	sí	8	8	sí
WA* (3)	00:00:001	15	9	sí	8	8	sí
WA* (4)	00:00:001	15	9	sí	8	8	sí
WA* (5)	00:00:001	15	9	sí	8	8	sí
Dijkstra	00:00:001	19	12	sí	8	8	sí

Tabla 9: Resultados de búsqueda para el problema 2



Prob. 3	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:001	22	11	no	--	--	no
EHC	00:00:002	23	12	no	--	--	no
BS (2)	00:00:004	70	36	sí	20	20	sí
BS (4)	00:00:005	94	49	sí	20	20	sí
BS (6)	00:00:005	94	49	sí	20	20	sí
BS (8)	00:00:005	94	49	sí	20	20	sí
BS (10)	00:00:005	94	49	sí	20	20	sí
A*	00:00:005	74	39	sí	20	20	sí
WA* (1,3)	00:00:004	62	33	sí	20	20	sí
WA* (1,6)	00:00:004	62	33	sí	20	20	sí
WA* (2)	00:00:004	62	33	sí	20	20	sí
WA* (3)	00:00:004	60	32	sí	20	20	sí
WA* (4)	00:00:003	47	26	sí	22	22	no
WA* (5)	00:00:003	47	26	sí	22	22	no
Dijkstra	00:00:002	104	55	sí	20	20	sí

Tabla 10: Resultados de búsqueda para el problema 3

Prob. 4	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:002	26	11	sí	11	11	sí
EHC	00:00:002	15	12	sí	11	11	sí
BS (2)	00:00:004	48	19	sí	11	11	sí
BS (4)	00:00:006	69	31	sí	11	11	sí
BS (6)	00:00:006	89	42	sí	11	11	sí
BS (8)	00:00:007	107	51	sí	11	11	sí
BS (10)	00:00:007	115	55	sí	11	11	sí
A*	00:00:004	39	16	sí	11	11	sí
WA* (1,3)	00:00:002	26	12	sí	11	11	sí
WA* (1,6)	00:00:002	26	12	sí	11	11	sí
WA* (2)	00:00:002	26	12	sí	11	11	sí
WA* (3)	00:00:002	26	12	sí	11	11	sí
WA* (4)	00:00:002	26	12	sí	11	11	sí
WA* (5)	00:00:002	26	12	sí	11	11	sí
Dijkstra	00:00:003	152	74	sí	11	11	sí

Tabla 11: Resultados de búsqueda para el problema 4



Prob. 5	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:004	54	24	sí	24	24	no
EHC	00:00:005	49	27	sí	22	22	no
BS (2)	00:00:007	98	45	sí	24	24	no
BS (4)	00:00:006	96	45	sí	14	14	sí
BS (6)	00:00:009	135	62	sí	14	14	sí
BS (8)	00:00:011	166	77	sí	14	14	sí
BS (10)	00:00:013	192	90	sí	14	14	sí
A*	00:00:004	42	20	sí	14	14	sí
WA* (1,3)	00:00:004	42	20	sí	14	14	sí
WA* (1,6)	00:00:004	44	21	sí	14	14	sí
WA* (2)	00:00:004	44	21	sí	14	14	sí
WA* (3)	00:00:004	44	21	sí	14	14	sí
WA* (4)	00:00:004	48	23	sí	14	14	sí
WA* (5)	00:00:004	48	23	sí	14	14	sí
Dijkstra	00:00:007	331	156	sí	14	14	sí

Tabla 12: Resultados de búsqueda para el problema 5

Prob. 6	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:002	26	11	sí	11	11	sí
EHC	00:00:002	15	12	sí	11	11	sí
BS (2)	00:00:003	48	19	sí	11	11	sí
BS (4)	00:00:005	74	33	sí	11	11	sí
BS (6)	00:00:006	94	44	sí	11	11	sí
BS (8)	00:00:008	114	53	sí	11	11	sí
BS (10)	00:00:009	129	60	sí	11	11	sí
A*	00:00:004	39	16	sí	11	11	sí
WA* (1,3)	00:00:002	26	12	sí	11	11	sí
WA* (1,6)	00:00:002	26	12	sí	11	11	sí
WA* (2)	00:00:002	26	12	sí	11	11	sí
WA* (3)	00:00:002	26	12	sí	11	11	sí
WA* (4)	00:00:002	26	12	sí	11	11	sí
WA* (5)	00:00:002	26	12	sí	11	11	sí
Dijkstra	00:00:004	198	94	sí	11	11	sí

Tabla 13: Resultados de búsqueda para el problema 6



Prob. 7	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:002	9	4	no	--	--	no
EHC	00:00:018	83	36	no	--	--	no
BS (2)	00:00:030	145	61	sí	33	33	no
BS (4)	00:00:053	257	107	sí	29	29	sí
BS (6)	00:00:075	373	157	sí	29	29	sí
BS (8)	00:00:099	473	202	sí	29	29	sí
BS (10)	00:00:120	590	250	sí	29	29	sí
A*	00:00:197	806	332	sí	29	29	sí
WA* (1,3)	00:00:082	333	138	sí	29	29	sí
WA* (1,6)	00:00:043	181	77	sí	29	29	sí
WA* (2)	00:00:039	165	70	sí	29	29	sí
WA* (3)	00:00:038	159	67	sí	29	29	sí
WA* (4)	00:00:042	176	74	sí	29	29	sí
WA* (5)	00:00:044	184	78	sí	29	29	sí
Dijkstra	00:01:150	34.581	14.774	sí	29	29	sí

Tabla 14: Resultados de búsqueda para el problema 7

Prob. 8	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:002	9	4	no	--	--	no
EHC	00:00:018	83	36	no	--	--	no
BS (2)	00:00:020	90	39	no	--	--	no
BS (4)	00:00:060	288	121	sí	33	33	no
BS (6)	00:00:100	502	219	sí	40	33	no
BS (8)	00:00:103	494	202	sí	29	29	no
BS (10)	00:00:124	600	250	sí	29	29	no
A*	00:00:084	357	150	sí	29	29	no
WA* (1,3)	00:00:067	273	115	sí	29	31	no
WA* (1,6)	00:00:057	239	103	sí	31	31	no
WA* (2)	00:00:066	271	116	sí	29	31	no
WA* (3)	00:00:065	272	119	sí	31	31	no
WA* (4)	00:00:084	358	157	sí	31	31	no
WA* (5)	00:00:078	334	148	sí	46	44	no
Dijkstra	00:00:352	11.415	4.926	sí	29	27	sí

Tabla 15: Resultados de búsqueda para el problema 8



Prob. 9	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:014	31	13	no	--	--	no
EHC	00:00:031	67	34	no	--	--	no
BS (2)	00:00:192	447	186	no	--	--	no
BS (4)	00:00:339	859	350	sí	90	90	no
BS (6)	00:00:456	1.194	494	sí	86	86	no
BS (8)	00:00:521	1.317	546	sí	72	72	no
BS (10)	00:00:628	1.602	666	sí	71	71	no
A*	00:50:358	108.393	45.362	sí	59	59	no
WA* (1,3)	00:10:487	24.455	10.149	sí	58	58	no
WA* (1,6)	00:03:498	8.217	3.391	sí	62	62	no
WA* (2)	00:00:954	2.243	915	sí	62	62	no
WA* (3)	00:00:577	1.384	573	sí	73	73	no
WA* (4)	00:00:454	1.048	429	sí	87	87	no
WA* (5)	00:00:421	981	402	sí	82	82	no
Dijkstra	03:09:368	1.192.692	519.569	sí	57	57	sí

Tabla 16: Resultados de búsqueda para el problema 9

Prob. 10	Tiempo	N. gen.	N. exp.	Sol.	Lon.	Cos.	Opt.
HC	00:00:074	237	65	no	--	--	no
EHC	00:00:033	156	57	no	--	--	no
BS (2)	00:00:144	545	148	sí	75	75	no
BS (4)	00:00:274	1.018	282	sí	72	72	no
BS (6)	00:00:347	1.292	348	sí	60	60	no
BS (8)	00:00:482	1.814	486	sí	63	63	no
BS (10)	00:00:605	2.265	616	sí	64	64	no
A*	--	--	--	--	--	--	--
WA* (1,3)	--	--	--	--	--	--	--
WA* (1,6)	01:08:786	161.852	43.120	sí	53	53	no
WA* (2)	00:16:097	51.184	13.690	sí	63	63	no
WA* (3)	00:00:693	2.574	701	sí	70	70	no
WA* (4)	00:00:352	1.281	348	sí	71	71	no
WA* (5)	00:00:346	1.277	349	sí	75	75	no
Dijkstra	--	--	---	--	--	--	--

Tabla 17: Resultados de búsqueda para el problema 10

5.5 Análisis de los datos obtenidos

A lo largo de las próximas líneas se comentarán los datos presentados en el apartado anterior desde distintos puntos de vista. Para ello, cuando se estime conveniente, se hará uso de diversos gráficos que facilitarán el entendimiento de las explicaciones. Tal y como sucedió con la presentación de los datos, se empezará comentando los resultados de **traducción** para pasar posteriormente a los de **búsqueda**.

El primero de los datos que se quiere destacar es el **rendimiento en tiempo del traductor**, el cual depende de la implementación del subsistema de representación específica del que se esté haciendo uso. La Ilustración 60 muestra los tiempos empleados en traducir los distintos problemas planteados, tanto haciendo uso del subsistema de representación específico del problema, como empleando el subsistema de representación en XML.

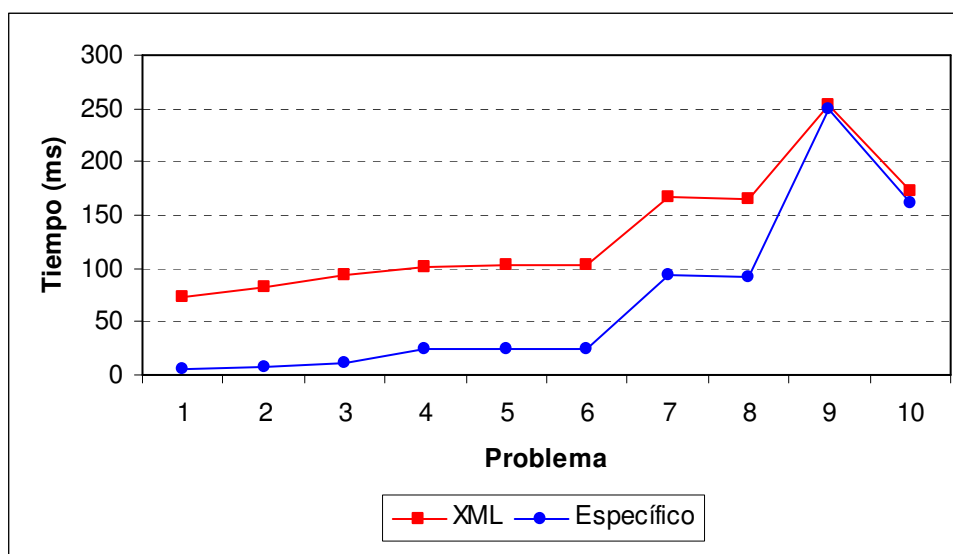


Ilustración 60: Rendimiento en tiempo del traductor

Tal y como se viene indicando a lo largo de este documento, el rendimiento de la representación en **XML es siempre inferior**, algo bastante notorio en problemas sencillos. Sorprendentemente, en los problemas más complejos (9 y 10) se observa un rendimiento prácticamente idéntico entre ambas opciones, algo inesperado para lo cual no se tiene una explicación fehaciente. Se estima que este comportamiento se debe detalles de implementación, bien de las clases de gestión de XML que contiene la BCL, bien del CLR. Por ejemplo, se sabe que el CLR mantiene una caché de cadenas de caracteres¹ que permite su reutilización, y dicha reutilización tiene beneficios varios en cuestión de rendimiento en tiempo, que son tanto más visibles cuanto más carga de procesamiento exista, como sucede en los problemas complejos.

En cualquier caso, la representación en XML siempre tendrá un peor rendimiento general en un videojuego que una específicamente ligada al mismo, pues la modificación del modelo de datos XML habrá de ser realizada de forma ineludible

¹ Más información en [http://msdn2.microsoft.com/en-us/library/system.string.intern\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/system.string.intern(VS.80).aspx)



mediante el manejo de cadenas de caracteres, algo que es relativamente lento en .NET, dado el carácter inmutable de las mismas. Este y otros beneficios siguen justificando lo genérico del subsistema de representación en Aran.

Se dirigirá ahora la atención a los resultados obtenidos con la utilización del **sistema de representación específico del problema**, pues se entiende que será el caso más típico a la hora de hacer uso de Aran un videojuego. Observando sus números en la Ilustración 60, puede decirse que el **rendimiento** obtenido en las pruebas en las que ha sido utilizado es bastante **satisfactorio**. En el caso más sencillo de entre los planteados apenas tarda **5 ms** en devolver el resultado, lo cual permite intuir el coste mínimo de una traducción. Siguiendo con los casos sencillos sobre el mismo sistema de representación, puede verse cómo la cota máxima se sitúa entorno a los 25 ms, mientras que en el caso de los más complicados esta cantidad es unas 10 veces mayor, rondando los **250 ms**. Se trata, en todo caso, de tiempos aceptables teniendo en cuenta que, si bien la eficiencia ha sido uno de los objetivos del proyecto, la labor realizada no ha estado destinada a alcanzar el grado óptimo en esta área. Existe, pues, margen de mejora.

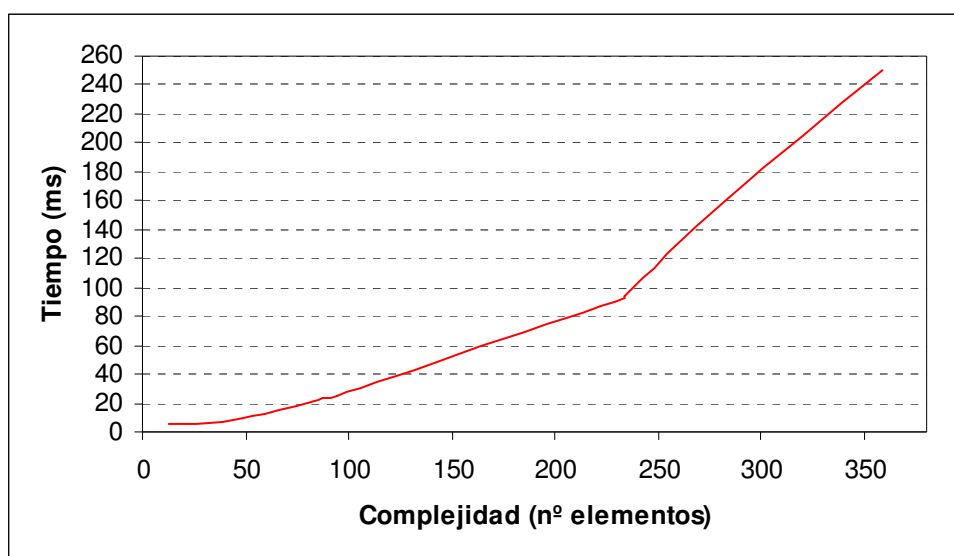


Ilustración 61: Relación entre complejidad de problema y tiempo de traducción

El rendimiento del traductor depende en buena media de la cantidad de elementos con los que debe tratar, entendiendo como tales los objetos, los hechos (iniciales y metas), las variables y las acciones, entre otros. A mayor número de elementos, mayor tiempo de respuesta, tal y como evidencia la Ilustración 61. En ella puede observarse la tendencia moderadamente creciente de su pendiente, que denota

una **proporción de orden lineal entre la complejidad del problema y el tiempo en traducirlo**. Ello no hace sino corroborar las buenas impresiones iniciales al respecto del rendimiento de esta parte del sistema.

Pasando ahora a comentar el **impacto del número de variables del problema en la representación del estado**, la Ilustración 62 muestra los datos que arrojan las traducciones de los distintos casos de prueba. En ella puede observarse cómo el número de variables representadas en el estado se mantiene constante (dos: tiempo y peso del personaje) mientras que el resto de las requeridas por el problema permanecen fuera del mismo, gracias al concepto de **variable global** que se ha comentado en esta memoria. Obsérvese cómo, por ejemplo, en el problema 9 llegan a representarse en el estado sólo el 15% de las variables del problema. Se trata de un hecho cuya importancia va más allá de lo concerniente al consumo de memoria, ya que las operaciones sobre estados de menor tamaño requieren también menos tiempo en ejecutarse, con el consecuente aumento de rendimiento en tiempo.

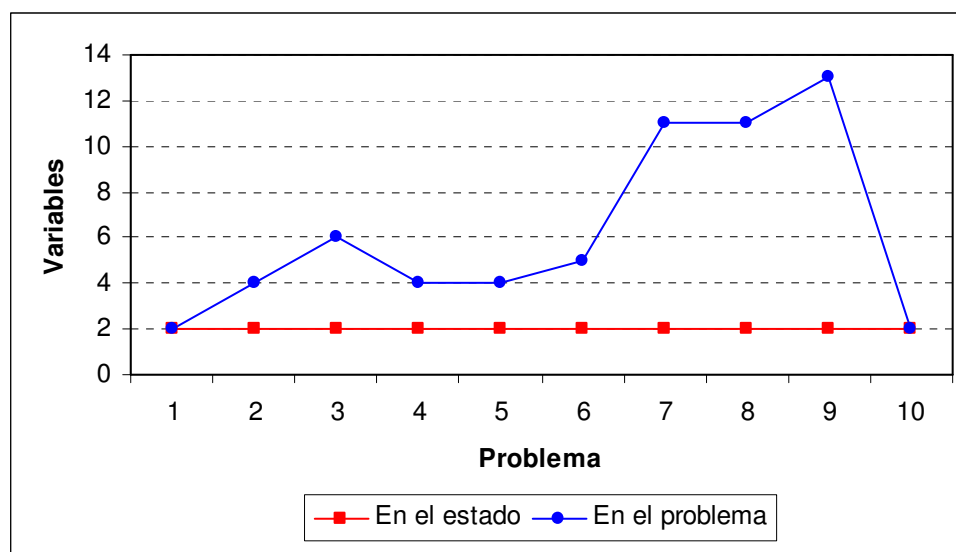


Ilustración 62: Variables en el problema frente a representadas en el estado

Con los hechos puede hacerse un análisis similar al realizado con las variables. La Ilustración 63 muestra el ahorro en la representación de hechos en el estado que suponen las técnicas empleadas en relación a los **predicados estáticos y de un solo sentido**. Como puede observarse, el ahorro que supone su aplicación es bastante notable en este dominio, debido especialmente al carácter estático de las conexiones entre baldosas. Es por ello que es mayor cuanto mayor es el mapa.

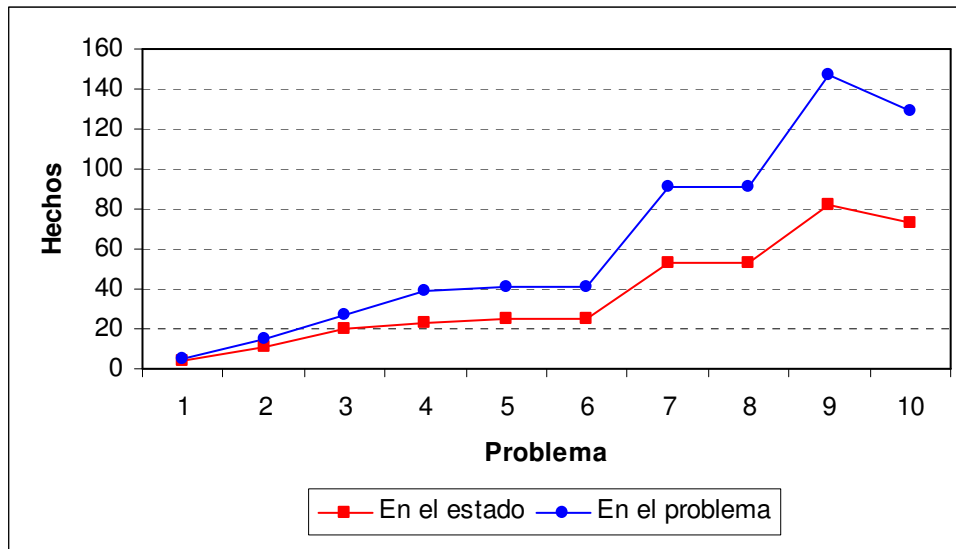


Ilustración 63: Hechos en el problema frente a representados en el estado

Puesto que el impacto que el número de variables representadas en el estado se mantiene constante para este dominio, la Ilustración 64 recoge una gráfica del patrón que sigue la variación del tamaño del estado en función de los hechos presentes en él.

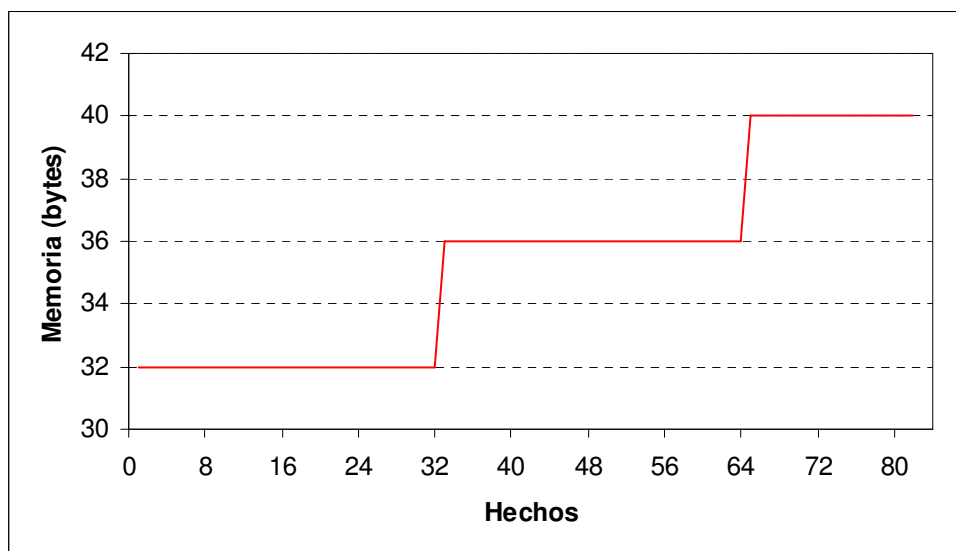


Ilustración 64: Impacto del número de hechos en el tamaño del estado

Como puede verse, **el estado aumenta en 4 bytes su tamaño por cada 32 hechos, debido a la representación en arrays de bits**, tal y como se ha expuesto en apartados previos. Esta representación, unida a los ya analizados ahorros en hechos y variables presentes en el estado, permite un consumo de memoria durante la búsqueda bastante satisfactorio a la par que **esperado**. De hecho, si se centra la atención sobre los casos resueltos en tiempos razonables y se realiza una sencilla multiplicación entre el

número de nodos generados y el tamaño del estado, podrá advertirse cómo en todos los casos el resultado es del orden de los KB. Bien es cierto que la memoria consumida en la representación de los estados no es la única empleada por el proceso de búsqueda, pero permite hacerse una idea del rendimiento de Aran en este sentido.

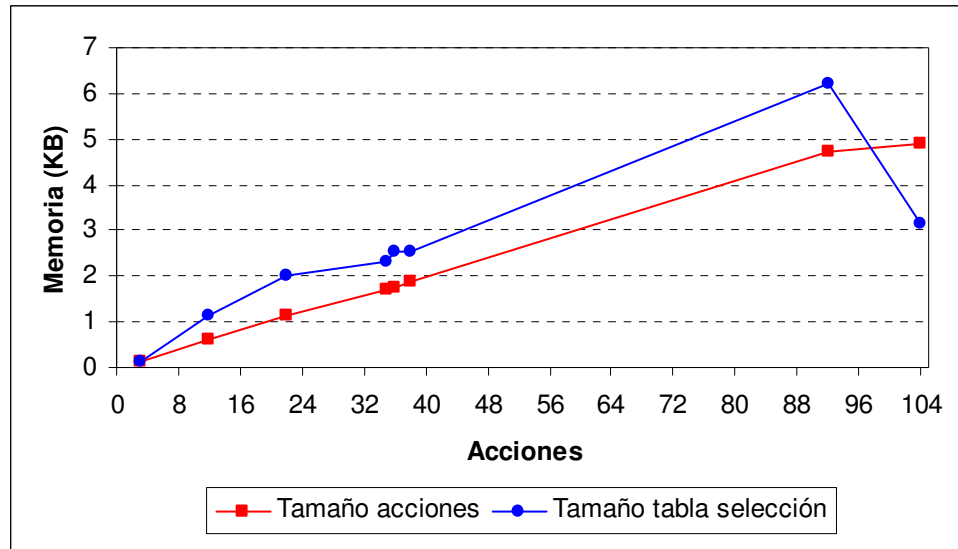


Ilustración 65: Relación entre el número de acciones y la memoria

Se ha aludido ya varias veces al consumo de memoria para justificar los esfuerzos en pro de la reducción del tamaño de los estados. Ahora bien, otro punto importante es el tamaño que ocupan en memoria las estructuras de datos relacionadas con las acciones, esto es, las **acciones** en sí y la **tabla de selección** de las mismas. La Ilustración 65 muestra la relación entre el número de acciones y los dos tamaños señalados. Como puede verse, en los casos evaluados la tendencia general es que la tabla de selección ocupe siempre más que las acciones. Esto en realidad no puede tomarse como norma, ya que existen otros factores de influencia no recogidos en la gráfica, como el número de hechos en el estado. Un ejemplo de ello es la última de las muestras, con 104 acciones correspondientes al problema 10, que tiene un número de hechos bastante inferior al del problema 9 y, también, una tabla de selección de menor tamaño. En cualquier caso, la **memoria empleada por estas estructuras es ínfima**, siendo el consumo máximo en las pruebas realizadas ligeramente superior a los 17 KB en suma (acciones y tabla). Si se tiene en cuenta que esta cantidad se mantiene fija en todo el proceso de búsqueda, puede decirse que se trata de un consumo de memoria despreciable. No debe menospreciarse este hecho. Que estas estructuras ocupen poca memoria implica que son también más pequeñas y veloces.



Comparando ahora los **problemas 4, 5 y 6** en términos de traducción se puede decir que apenas existen hechos reseñables. Entre los 3 **apenas hay diferencias** en tiempo y las que hay pueden achacarse a factores que pueden considerarse aleatorios. En cuanto al número de objetos, variables, etc., existen diferencias, pero nada que no sea esperable, por ser consecuencia directa del planteamiento del problema. Por ejemplo, en el 5 se tiene una meta más debido a la llave de más respecto al 4, mientras que en el 6 se tiene una variable más debido al peso del nuevo martillo. Como se ha dicho, existen más diferencias, pero ninguna destacable. Sí puede aprovecharse esta comparativa **para ver con mayor claridad el efecto de la no representación de predicados estáticos** en el estado. Si se observa, el problema 5 tiene un hecho inicial más que el 4, correspondiente al color del martillo. Ahora bien, el número de hechos representados en el estado no varía, debido a que el predicado empleado para la asociación de colores es estático.

En cuanto a los **problemas 7 y 8**, la única diferencia existente es en relación al tiempo de traducción, menor en el segundo caso. Ello es debido a que las **inicializaciones a 0** no se especifican en la definición del problema, pues las variables valen 0 por defecto. Ello supone un ahorro en procesamiento y, por tanto, en tiempo.

En otro orden de cosas, se procederá ahora a comentar el comportamiento del sistema en términos de **búsqueda** con el fin de valorar lo acertado o no de la decisión de incluir varios algoritmos en Aran. No se pretende, pues, analizar cuál de ellos conviene utilizar en este u otros dominios, sino determinar si existen o no diferencias suficientes entre sus comportamientos que justifiquen esta decisión.

La primera de las cuestiones que se abordará es la **capacidad de los algoritmos para encontrar una solución**. La Ilustración 66 muestra el número de problemas resueltos por cada uno, así como el número de veces que esas soluciones han sido óptimas. Como puede verse, la mayoría de los algoritmos tienen una **buena tasa de solución**, salvo los de escalada (EC y EHC) y la búsqueda en haz de menor anchura (2). El hecho de que estos algoritmos manejen pocas alternativas paralelas hace que sufran de forma más notable los errores de descarte realizados basándose en la función heurística. Téngase en cuenta que el único caso *no resuelto*¹ por los algoritmos A*, WA* (1,3) y Dijkstra es el 10, y no se debe a dicho descarte sino a todo lo contrario y,

¹ En verdad, se detuvo su ejecución pasados unos minutos.

claro, los más de 400 millones de estados que se estiman diferentes para dicho problema.

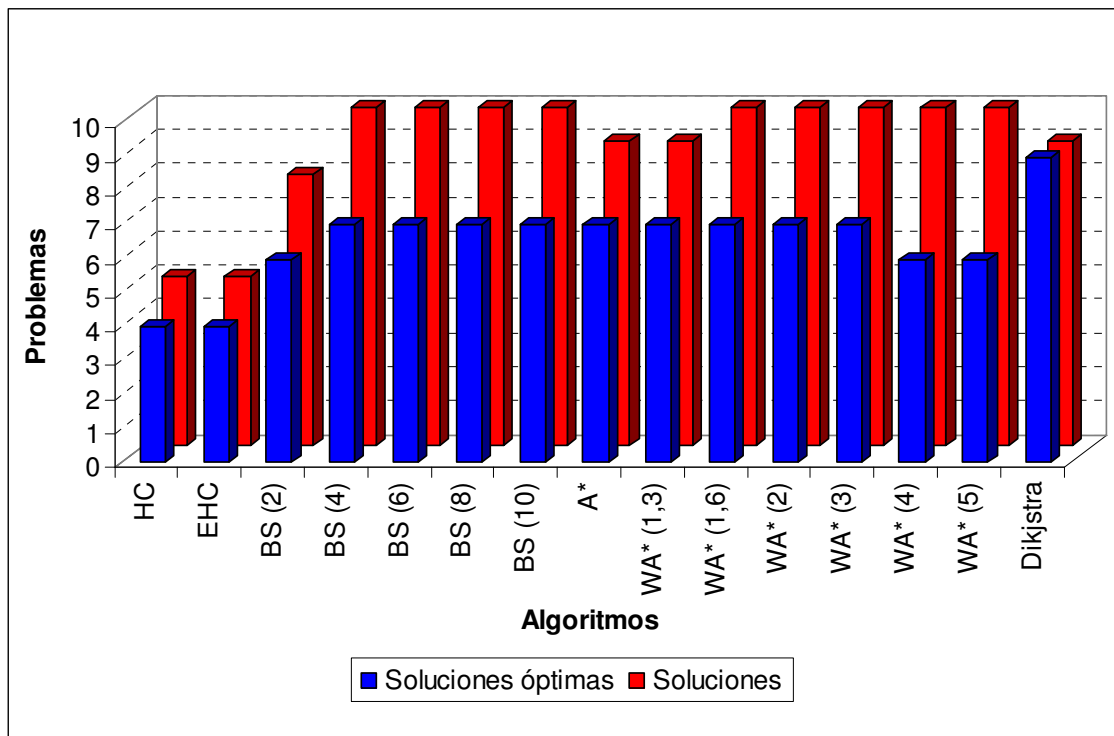


Ilustración 66: Comparativa entre soluciones encontradas y óptimas

Incluir varios algoritmos de búsqueda permite al usuario **elegir en función de las características del problema** y la Ilustración 66 apoya esta inclusión evidenciando que incluso los menos favorecidos en esta estadística, **EC, EHC y BS(2)**, son capaces de obtener soluciones y óptimos, siendo su ámbito de eficiencia el de los problemas **sencillos**. Por otro lado, la oferta de **algoritmos paramétricos** (BS y WA*) se demuestra también como una medida acertada, en tanto en cuanto permite jugar con los valores de sus argumentos para **modificar su rendimiento** en función, también, de las características del problema. No hay más que ver que BS es capaz de resolver todos los problemas en cuanto se le aumenta la anchura del haz por encima de 2 mientras al WA* le sucede lo mismo cuando se aumenta el peso de la heurística. A pesar de ello, la modificación de estos valores puede tener otras consecuencias menos deseables, como evidencia la gráfica de la Ilustración 67 en el caso del WA*.

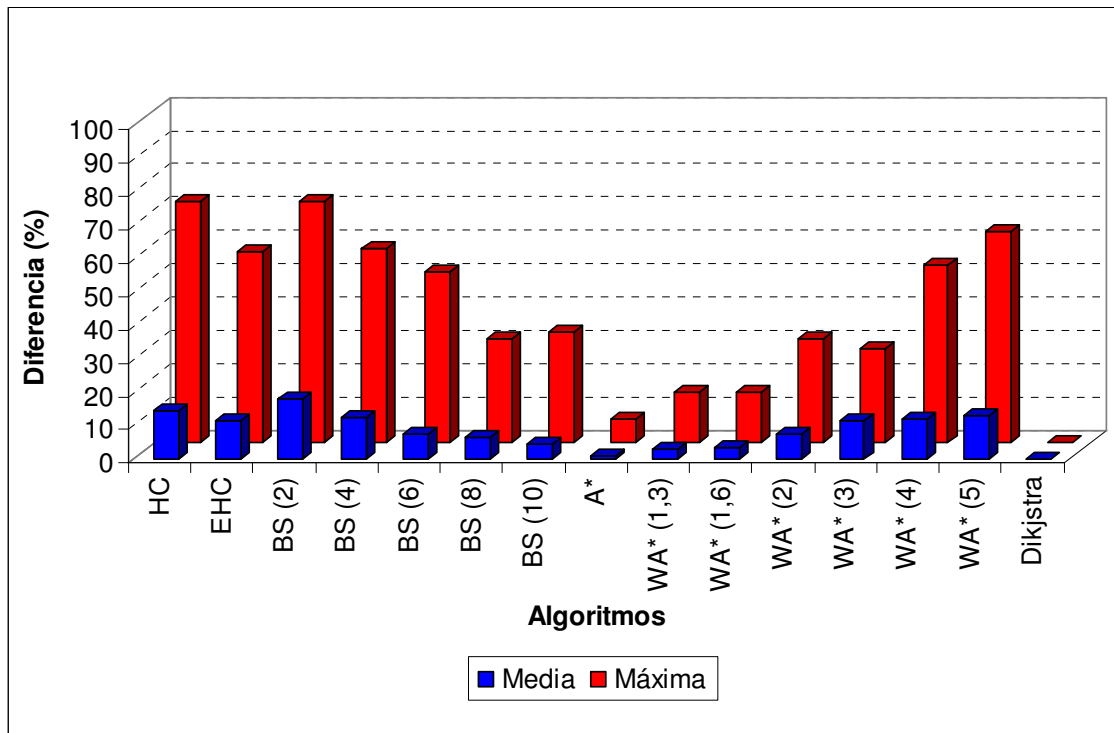


Ilustración 67: Diferencias porcentuales entre soluciones y óptimos

Como puede apreciarse, la **diferencia de la calidad** de la solución devuelta por el **WA*** respecto al óptimo aumenta en la medida en que la heurística tiene más peso. Ocurre justo lo contrario con el parámetro de **BS**, que refina la calidad de la solución cuanto mayor es. En ambos casos la razón es sencilla de explicar. En el **WA*** el mayor peso de la heurística hace que el algoritmo confíe más en ella a la hora de dirigir sus pasos, lo cual suele traducirse en la degradación de la solución, tanto más cuanto menos precisa es la heurística. En el caso del **BS**, el aumento del tamaño del haz le permite al algoritmo confiar menos en la heurística y mantener más alternativas al mismo tiempo, traduciéndose comúnmente en un aumento de la calidad. Un dato preocupante es que, a excepción de los dos algoritmos más precisos, Dijkstra y **A***, el resto tienen un **error máximo** ciertamente elevado, lo cual se traduce en la posibilidad de que, para ciertos problemas, la solución devuelta sea de una calidad bastante cuestionable. Y es que una pérdida de calidad **de entre el 50 y el 70%** (como sucede en varios casos) es **difícilmente aceptable**. Ahora bien, retomando la idea original de proporcionar distintas alternativas, esto viene a corroborar la necesidad de las mismas, pues es tarea del usuario determinar qué le merece más importancia en cada caso: la calidad de la solución o el rendimiento en tiempo y en memoria. Por ejemplo, en juegos con incertidumbre elevada la calidad de la solución pierde relevancia.

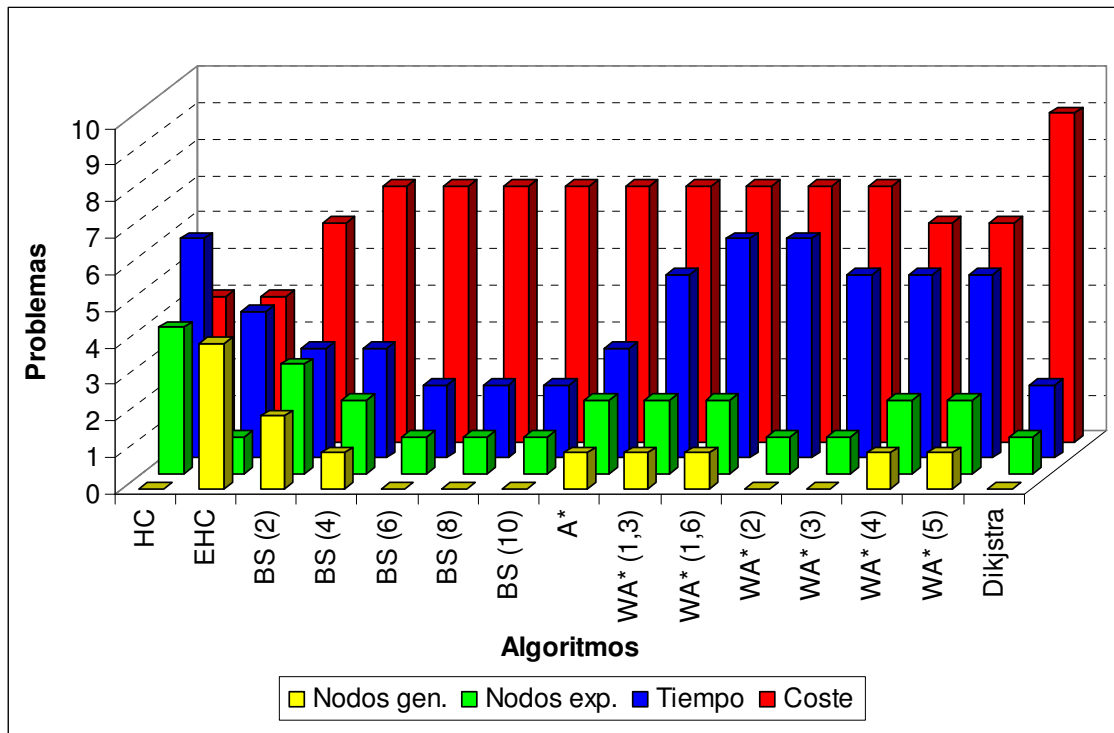


Ilustración 68: Optimalidad de los algoritmos en nodos, tiempo y coste

Otra estadística que confirma que la variedad de algoritmos es necesaria para afrontar un amplio abanico de problemas es la mostrada en la Ilustración 68. En ella se recoge el número de veces que cada algoritmo ha sido el mejor en un cierto campo, pudiendo ser en la minimización del número de nodos generados, del número de nodos explorados, del tiempo empleado en la búsqueda o del coste de la solución. Así, se puede ver cómo las **variantes del BS** con mayor amplitud de haz **hacen frente, en términos de mejor coste, a algoritmos como el A* y el WA*** y, sin embargo, el número de veces que son los más rápidos en responder es menor que en cualquier otra alternativa. Otro dato interesante es que el **WA* con un peso bajo** de la heurística suele tener una **buena relación entre la calidad de la solución y el tiempo** empleado en encontrarla, justo lo contrario de lo que le sucede al algoritmo de **Dijkstra**, sólo recomendable en problemas pequeños. En general, no parece que la **máxima calidad** que ofrece este algoritmo suela compensar los recursos que requiere, pero eso, nuevamente, es una decisión que debe tomar el usuario. Otro dato de interés es que el **HC** es el que mayor número de veces sale ganador en **tiempo de respuesta y nodos explorados**, mientras que el **EHC** hace lo propio en el número de nodos **generados**, gracias a las heurísticas de poda que le distinguen del resto (*“added goal deletion”* y *“helpful actions”*).

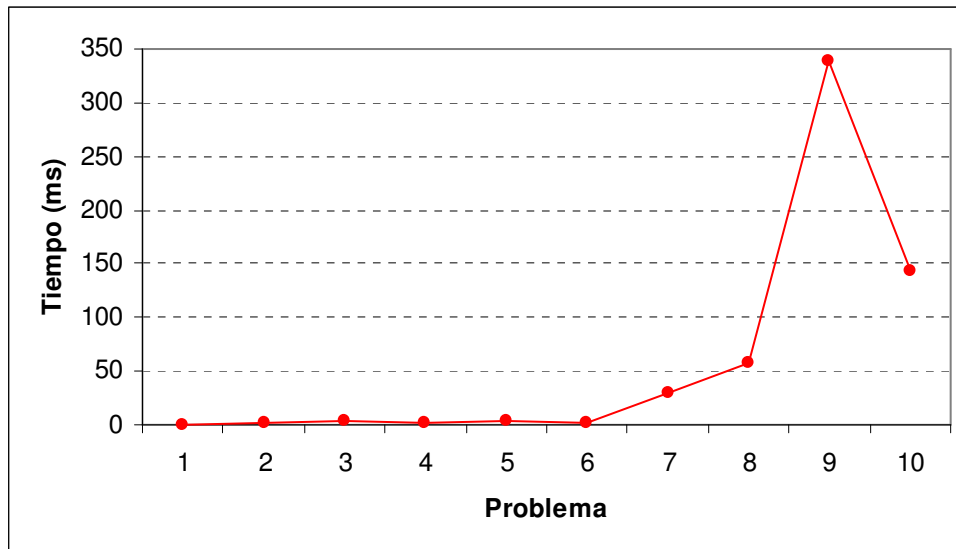


Ilustración 69: Mejor tiempo de búsqueda por problema

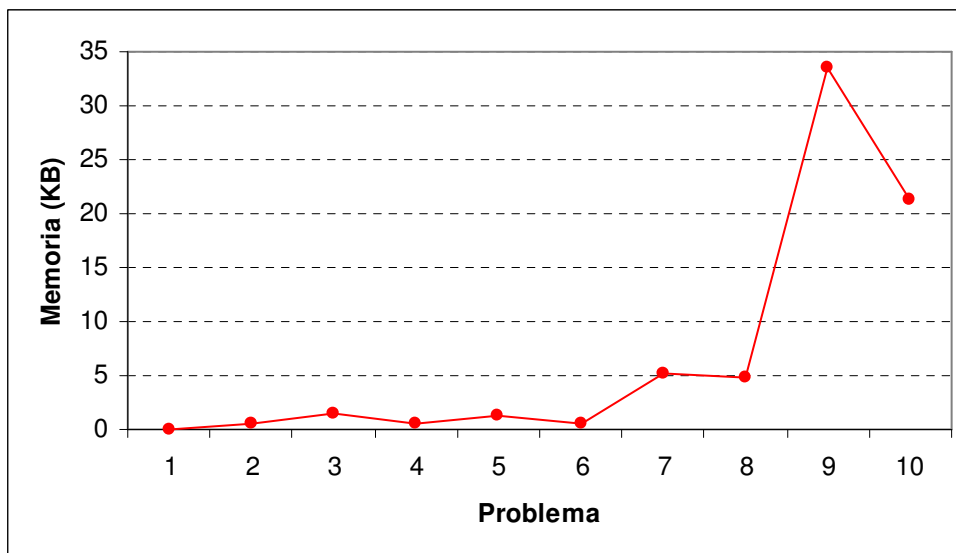


Ilustración 70: Memoria consumida por cada mejor tiempo de búsqueda

Habiendo, pues, justificado la inclusión de varios algoritmos alternativos en Aran, queda preguntarse qué puede obtenerse de esa posibilidad de elección. En este sentido es útil la Ilustración 69, pues muestra una gráfica en la que puede verse el menor consumo de tiempo llevado a cabo para cada problema por alguno de los algoritmos disponibles. Para que pueda verse la **relación** que existe entre ese **consumo de tiempo y el de memoria**, la Ilustración 70 muestra la cantidad de la misma consumida para almacenar los estados por parte del algoritmo que empleó menos tiempo en resolver cada problema. En general, puede verse que existe una **estrecha** relación entre los dos conceptos. Al mismo tiempo, se aprecia cómo las cotas máximas

manejadas en ambos casos son bastante satisfactorias, no superando los **350 ms** en tiempo ni los **35 KB** en memoria de estados.

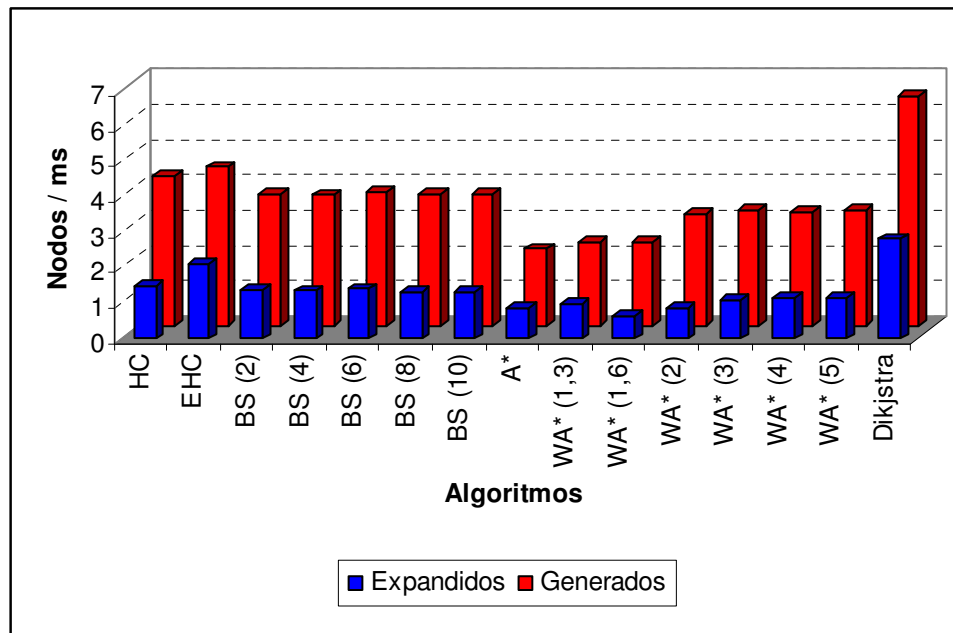


Ilustración 71: Rendimiento de los algoritmos en nodos / ms

Un último dato de interés con respecto a los distintos algoritmos implementados es su **rendimiento en términos de generación y exploración de nodos** por milisegundo. La Ilustración 71 contiene datos al respecto que denotan la mayor capacidad de algoritmos como **HC, EHC y, especialmente, Dijkstra**, en estos términos de rendimiento. En el caso de los algoritmos de escalada, esto se debe a la menor complejidad en el manejo de estructuras internas. En el destacado caso de Dijkstra se debe a la no utilización de heurística alguna. Esto da una idea del **impacto de la heurística** implementada en el rendimiento de los algoritmos que la utilizan, puesto que poco más que esa es la diferencia que separa a este algoritmo de otros, como el A*, cuyo rendimiento es notablemente inferior.

Se concluirá comentando, una vez más, algunos casos concretos de entre los planteados, siempre en términos de búsqueda. Comenzando por las diferencias observadas entre los **problemas 4, 5 y 6**, decir que **se aprecia el impacto en la búsqueda sobre ambas variaciones (5 y 6)**, siendo más acusado en el problema 5 que en el 6. En el 6 el tiempo invertido, en general, es mayor que en resolver el problema 4 debido a que la adición del nuevo elemento a tener en cuenta provoca el aumento del espacio de estados y, por tanto, de las posibilidades a explorar. En el 5 el tiempo



necesario es mayor aún, ya que, además del coste añadido a la adición del elemento, éste es una meta, con lo que aumenta por dos vías la complejidad de resolución del problema.

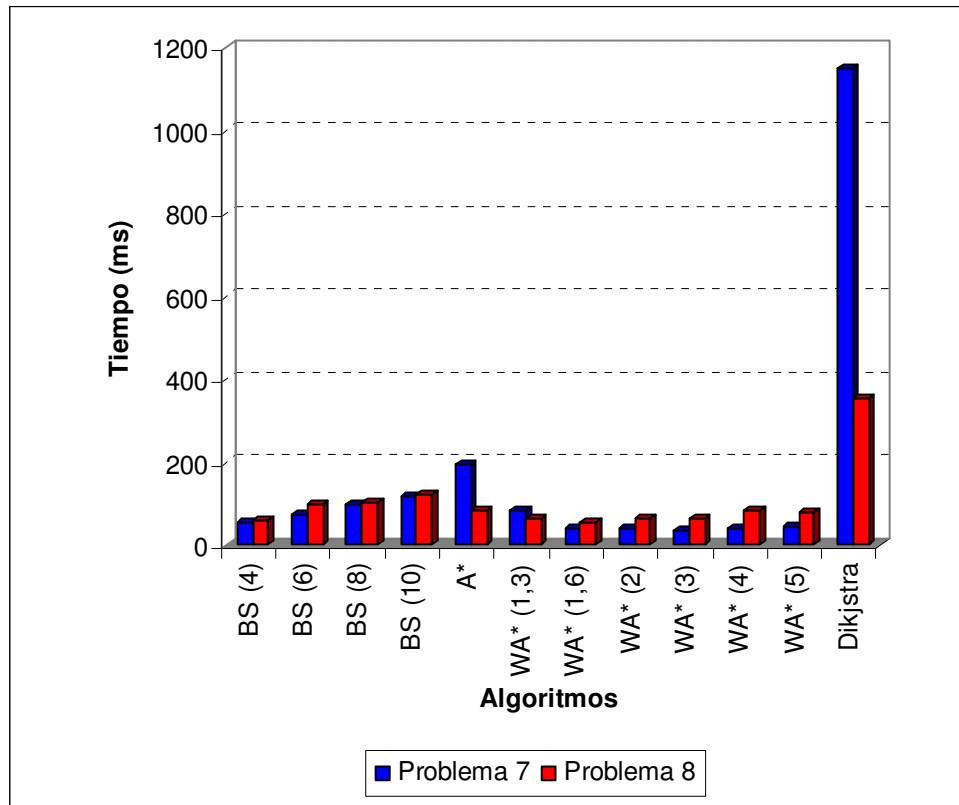


Ilustración 72: Tiempo de búsqueda para los problemas 7 y 8

En cuanto a los **problemas 7 y 8**, se puede ver en la Ilustración 72 cómo la **modificación de los valores de las variables** afecta de forma diferente según el algoritmo empleado. Así, mientras WA* con valores altos del peso heurístico tarda más tiempo en dar una solución, A*, WA* con pesos bajos y, sobre todo, Dijkstra, aprecian reducciones en el mismo. En general, podría decirse que **reducen sus tiempos aquellos algoritmos que tienen depositada menos confianza en la heurística**. Esto se debe a que ésta está originalmente pensada para evaluar basándose en el número de acciones del plan relajado, es decir, asumiendo un coste uniforme para todas ellas. Cuando se varían los valores de las variables el coste deja de ser uniforme y el orden de las acciones del plan relajado cobra importancia a la hora de obtener buenas estimaciones: las mismas acciones ordenadas de forma diferente producen un coste distinto. El hecho de que Dijkstra reduzca su tiempo hasta el 30% del empleado en el problema 7 se debe a que no está influido por la heurística, pues no la emplea, y, sin embargo, la

diferenciación de los objetos del problema mediante los distintos valores de sus variables le sirve para hacer una mejor selección de los nodos a explorar en primer lugar. Con mejor tiempo o no, lo inevitable es que la degradación de la calidad de la heurística tenga un impacto negativo sobre la calidad de los resultados devueltos. En la Ilustración 73 puede verse cómo, frente a las nulas diferencias que existían en el problema 7, las soluciones para el problema 8 difieren un mínimo de un 7% respecto al óptimo. En general, parece que **la heurística empleada tiene dificultades con el manejo numérico**, algo que era de esperar, pues, tal y como se ha explicado en este documento, el coste de las acciones sólo es tenido en cuenta tras, y no durante, la elaboración del plan relajado.

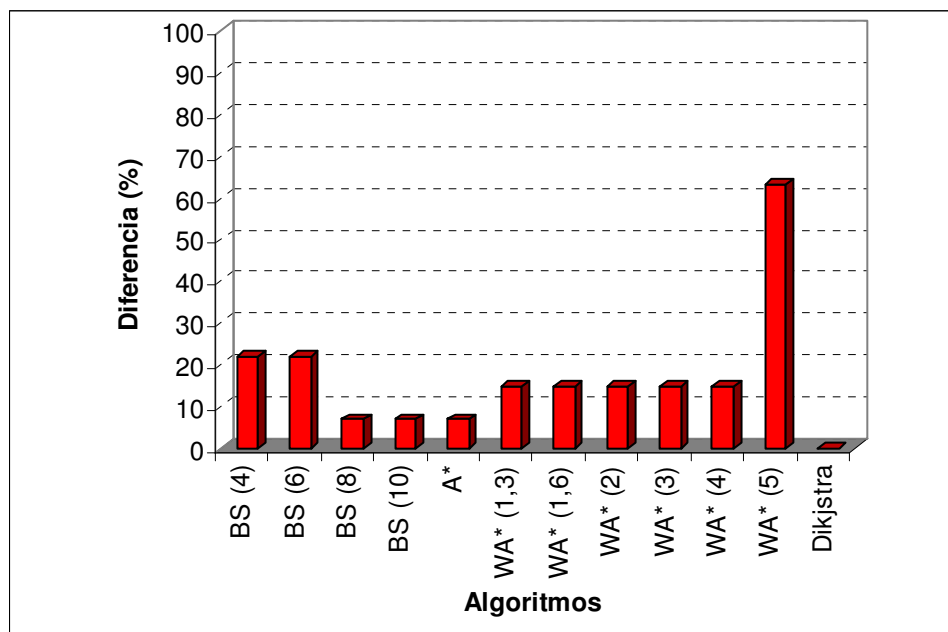


Ilustración 73: Diferencias entre soluciones y óptimo para el problema 8

Respecto a los **problemas 9 y 10**, decir que es en éstos en los que se pone claramente de manifiesto la necesidad del usuario de calibrar los parámetros de búsqueda según sus necesidades. Y es que aquí los algoritmos que venían dando los mejores resultados (**A***, **Dijkstra**) se convierten en inútiles dados los abismales tiempos requeridos para resolver los problemas (**decenas de segundos e incluso minutos**). Esto se hace notar especialmente en el caso del problema 10, cuyo aspecto sencillo al humano esconde un problema de difícil solución en términos de tiempo y memoria. En casos como este, dotar al usuario de alternativas a estos costosos y populares algoritmos



se convierte en un valor añadido para Aran, pues posibilita la resolución del problema, aún a costa de una notable pérdida de calidad, si el usuario está dispuesto a asumirla.

CAPÍTULO 6

CONCLUSIONES

*Tan nocivas como la impaciencia y la precipitación
son la obsesión y las dilaciones*



6 Conclusiones

Extraer los aspectos positivos y negativos de un sistema informático, como es este proyecto, pasa necesariamente por evaluar el grado de cumplimiento que de los objetivos inicialmente marcados se ha conseguido una vez finalizado el mismo. Así pues, a continuación se enumeran de nuevo esos objetivos, adjuntando a los mismos las conclusiones que pueden derivarse de su implementación en Aran.

1. **El idioma del producto resultante debe ser el inglés.** Este objetivo es, seguramente, el que menos dudas puede generar en una valoración objetiva de Aran. Y es que todo, absolutamente todo, en este sistema está en inglés a excepción de este documento. Todo el desarrollo se ha realizado en dicho idioma hasta el punto de emplear las versiones inglesas de las herramientas para evitar que auto generasen contenido en castellano. Incluso los textos utilizados por el sistema fueron inicialmente escritos en inglés y, posteriormente, traducidos al español (abriendo, de hecho, la vía a posibles traducciones a otros idiomas). Así pues, este objetivo se considera cumplido al 100%, obviando, claro está, las deficiencias que de este idioma pueda tener el autor del proyecto y que puedan haber sido plasmadas en la elaboración del mismo.
2. **El sistema de planificación debe ser funcional.** Aran ha sido construido de forma completa y se encuentra totalmente operativo y libre de errores en la medida en que las pruebas realizadas así lo han permitido. Por tanto, se considera un proyecto plenamente funcional y preparado para ser utilizado sin modificación alguna. Un usuario de este sistema únicamente debe ejecutar el sencillo instalador creado al efecto y en sólo unos *clics* puede estar utilizándolo desde Visual Studio o cualquier otro IDE compatible con el sistema de integración implementado. Este objetivo se considera, por consiguiente, cumplido también en su totalidad.
3. **El producto resultante debe facilitar su continuación por terceros.** Varios factores han influido en que, a pesar de los esfuerzos, este objetivo no se considere cumplido al nivel deseado. Por un lado, el diseño del sistema es lo suficientemente complejo como para dificultar su entendimiento sin ayuda de algún tipo de documentación que ayude a comprender los puntos menos claros. Si incluso de esta memoria se han excluido multitud de detalles para evitar un tamaño excesivo,



piénsese un instante en qué situación quedaría alguien que, no entendiendo el castellano, fuera incapaz de leerla. Es obvio que en este sentido hay un hueco aún por cubrir, especialmente en habla no hispana. Por otro lado, aunque el código está profusamente documentado, es también cierto que no se ha puesto el mismo énfasis en unas partes del mismo que en otras. Los elementos más visibles, públicos, están mejor explicados que los internos. Si, por la razón que fuera, alguien necesitase modificar puntos peor documentados, podría encontrarse con mayores dificultades. A ello hay que añadir el hecho de que no a todo el código generado se le puede decir de calidad. Nuevamente, existen partes del sistema mejorables en este sentido. Mención especial requiere el traductor, cuyo código es consecuencia directa de una mala captación de requisitos iniciales, en buena parte debido a la inexperiencia en relación al desarrollo de sistemas de este tipo. Por ejemplo, el hecho de ser un traductor cuyo *parser* puede ser implementado de formas diversas, no necesariamente textuales, resultó ser uno de los mayores escollos en el desarrollo y uno de los principales artífices de la baja calidad del código de ese módulo. Por otro lado, el empleo, quizá demasiado intenso, de genéricos en este mismo módulo y la ausencia de alias para éstos en C# 2.0 es también culpable de una legibilidad mejorable. Por último, la redacción de la documentación XML a lo largo de los distintos ficheros fuente se realizó pensando en el acabado que tendría previo paso por un procesador como Sandcastle. Esto, que ha dado un acabado francamente bueno a la documentación HTML generada con esa herramienta (llena de útiles hipervínculos), ha dificultado al tiempo la lectura de la misma a nivel de fichero fuente. Incluso, en ocasiones complica la lectura de la documentación generada por los sistemas de autocompleción de código, como el IntelliSense de Visual Studio. Sin duda, ésta ha sido una tarea ardua y difícil de calibrar (a mejor HTML, peor IntelliSense y viceversa).

4. **El producto resultante debe facilitar su uso por terceros.** Después de un punto anterior plagado de *peros*, hay que aclarar que este sí se ha cumplido satisfactoriamente. El hecho de haber dividido Aran en varios niveles de abstracción permite que éste sea usado tanto por usuarios expertos como por profanos. El usuario puede elegir a qué nivel trabajar, siendo el comando `aran.exe` el mayor nivel de abstracción del que se provee. Con él, cualquier persona con mínimos conocimientos sobre planificación puede hacer uso del sistema sin preocuparse de



cómo funcionan sus entresijos. No obstante, satisfactorio no es sinónimo de perfecto. Los esfuerzos por evitar al usuario de alto nivel el conocimiento del funcionamiento interno del sistema, unidos a la necesidad de hacer un producto flexible, han derivado en la carencia de elementos de utilidad similares a los provistos por el comando y su interfaz en forma de biblioteca de enlace dinámico, pero a un nivel más bajo. Se aprecia, pues, la necesidad de algún sistema de envoltura que permita, por ejemplo, la creación rápida y sin preocupaciones de algoritmos de búsqueda sobre la base de una cierta configuración común. Lo mismo se puede decir de implementaciones de referencia o de base del subsistema de representación, que puedan servir de ayuda a terceros con el fin de que no requieran partir de cero a la hora de realizar su propia implementación o que, incluso, les permitan no tener que elaborar ninguna implementación propia.

5. **El sistema de planificación debe ser genérico.** Finalmente, Aran es un sistema de planificación genérico, con todo lo que ello conlleva. Dispone de un sistema abstracto de representación que le posibilita ser integrado en una amplia variedad de aplicaciones, sean o no videojuegos, sin obligar al usuario a pasar por ningún lenguaje textual intermedio que pueda dificultar la interacción o penalizar el rendimiento. Este ha sido, sin duda, uno de los puntos más complicados del desarrollo, pues habiendo limitado el mismo a una interfaz textual clásica se habrían eliminado buena parte de los problemas que se han tenido. Ahora bien, ser genérico implica ir más allá de poder implementar nuevos sistemas de representación adaptados al entorno en el que se integra, y es ahí donde este proyecto puede ser mejorado. Sin que esto suponga un incumplimiento de las metas marcadas, lo cierto es que la heurística implementada no es lo suficientemente precisa cuando hay elementos numéricos de por medio. Esto, sin duda, es un escollo de cara a la reutilización del sistema en problemas de índole diversa. Otro escollo unido a lo genérico del sistema se refiere a la cómo están implementados los algoritmos de búsqueda. Y es que el modelo basado en componentes del que hacen gala conlleva un mayor número de llamadas a método y creaciones de objetos efímeros, factores ambos con repercusiones negativas en el rendimiento. En otras palabras, los algoritmos rinden satisfactoriamente, pero podrían hacerlo mejor si no fueran genéricos y/o no siguieran un modelo de composición. En cualquier caso, y dada el alto desacoplamiento del sistema, es factible la implementación de algoritmos *ad-*



hoc sin que ello afecte en absoluto a lo realizado en este proyecto. Por último, Aran es un sistema genérico de planificación, sí, pero eso no debe confundirse con ser un planificador para todo. En este proyecto se ha implementado un tipo de planificador concreto con el cual podrán resolverse, únicamente, problemas que caigan dentro de su ámbito de aplicación.

6. **El sistema de planificación debe ser eficiente.** Las pruebas que se le han hecho al sistema arrojan buenos resultados en términos de eficiencia. Si es poco común hacer uso en videojuegos de sistemas de planificación, menos lo es que se empleen para resolver problemas tan complejos como algunos de los planteados en el apartado correspondiente. Por tanto, entendiendo que los problemas a resolver tendrán una dificultad media, se estima apropiado el rendimiento de Aran tanto en tiempo y memoria como en la calidad de las soluciones aportadas. Quizá aquí el principal limitador en el uso del sistema sea el tiempo que requiere su ejecución. Piénsese que, comúnmente, un videojuego se actualiza 60 veces por segundo, lo cual significa una actualización cada menos de 17 ms. Si se tiene en cuenta que las mediciones de tiempo realizadas en las pruebas eran aisladas, y que en cada actualización de un juego la IA es sólo una parte del trabajo a realizar, se llega rápidamente a la conclusión de que Aran no puede ser ejecutado en tan escaso margen temporal. Ni siquiera duplicando el tiempo entre actualizaciones (hay juegos que se actualizan “sólo” 30 veces por segundo) parece que este uso sea factible. Se hace, pues, necesario distribuir el coste de ejecución del sistema entre varias actualizaciones, utilizando para ello cualquiera de las vías disponibles como, por ejemplo, la ejecución paralela (recuérdese que la Xbox 360 es una máquina multinúcleo). Otra vía de reducción del impacto en el rendimiento del videojuego es la simplificación de los problemas a resolver (por ejemplo, representando secuencias de acciones como una sola). También pueden usarse técnicas que permitan decidir cuándo debe realmente utilizarse el planificador, intentando minimizar el número de ejecuciones. Otro caso típico es la ejecución de planes parciales.
7. **El sistema de planificación debe ser flexible.** Aran es flexible, de eso no hay duda. Su diseño desacoplado permite moldear su uso a las necesidades del usuario. Así, éste puede decidir utilizar todo o parte del sistema y, si lo necesita, complementarlo con nuevos desarrollos propios que interactúen con él siguiendo las interfaces de las



que dispone. Todo ello haciendo uso del producto generado, sin necesidad de modificar su código fuente. Esto, que puede parecer una característica inherente a todo proyecto informático, es, por desgracia, algo no tan extendido como debiera.

8. **El sistema de planificación debe ser multiplataforma.** Este punto es otro que erróneamente suele creerse inherente a todo desarrollo en .NET. Tal y como se ha explicado a lo largo de este documento, Aran es multiplataforma, concretamente compatible con PC y Xbox 360, ya sea para su uso en aplicaciones o videojuegos, pero lo es porque se han tomado las medidas necesarias para ello. Se trata de un punto que tiende a trivializarse pero que, en verdad, conlleva también su esfuerzo. Nuevamente, si el desarrollo del sistema se hubiera centrado únicamente en una plataforma, habría sido más sencillo de realizar. Con todo, haber tenido en cuenta esta característica desde el comienzo ha hecho que sea prácticamente directa la conversión entre ambas plataformas. Pero, se insiste, no es algo que surja por el mero hecho de emplear tecnología .NET. Surge de un diseño pensado para ser multiplataforma, y Aran lo tiene. De hecho, tal y como se ha comentado en este documento, realizar una versión del sistema para el .NET Compact Framework estándar (el que emplean dispositivos como los PocketPC) sería casi directa.
9. **El producto resultante debe seguir los estándares de .NET.** Otra vez, se está ante un objetivo aparentemente sencillo de cumplir, que se considera cubierto satisfactoriamente pero que tampoco es fruto de la casualidad. Seguir los estándares de .NET conlleva, en primer lugar, conocerlos, algo que tampoco está tan extendido como se podría pensar. Aún conocidos los estándares, siempre existe la posibilidad de incumplirlos involuntariamente. La incorporación de una herramienta como FxCop al proceso de desarrollo ha facilitado la coherencia en este sentido. Aún así, se han tomado ciertas libertades al respecto de determinadas recomendaciones que no afectan a código público y con las cuales se discrepa (como el uso de prefijos de campo). El producto final no sólo cumple la CLS sino que, además, informa de ello a sus usuarios por medio de los metadatos correspondientes. Se le puede considerar, en este sentido, un buen habitante del mundo .NET. Otros estándares, como la posibilidad de *serializar* los objetos o el uso de XML también han sido seguidos con acierto.

CAPÍTULO 7

LÍNEAS FUTURAS

*La obra humana más bella
es la de ser útil al prójimo*



7 Líneas futuras

Aran es un proyecto abierto y con margen de mejora en muchos sentidos, así como con amplias posibilidades de ampliación por medio de extensiones de funcionalidad o utilidades complementarias. A continuación se listan algunas de las posibles formas de darle continuidad.

- **Reducción del tamaño del estado.** Actualmente, el traductor de Aran aplica diversas optimizaciones destinadas a reducir el tamaño del estado, como pueden ser la eliminación de hechos estáticos del mismo o la ubicación de ciertas variables, consideradas globales, fuera de él. Pero aún puede irse más allá. Una mejora en este sentido sería implementar un sistema de eliminación de hechos y variables no utilizados, algo que ahora no se hace. Con ello se conseguiría reducir aún más el tamaño del estado pero habría que tener presente esta optimización en el proceso de reasignación de identificadores de *instancia* que se lleva a cabo ahora mismo. Puesto que esta y otras optimizaciones pueden suponer una pérdida de rendimiento que el usuario puede querer evitar, esta mejora podría añadirse como opcional al proceso de traducción.
- **Llamada a métodos externos.** Una funcionalidad que se barajó durante el proceso de construcción del sistema pero que, finalmente, fue descartada, fue la posibilidad de permitir al usuario llamar a métodos propios dentro de la especificación de un dominio o un problema. Estos métodos cumplirían lo que [Orkin, 2004] denomina condiciones y efectos contextuales. El actual sistema de extensión funcional serviría de punto de partida para esta nueva característica. El principal reto aquí sería establecer el mecanismo de traducción de los elementos presentes en el problema (objetos, hechos, variables...) a un formato entendible por la función receptora, pues debe recordarse que dichos elementos han sido codificados como paso previo al proceso de búsqueda. En este sentido, el codificador y el decodificador de *instancias* devuelto por el subsistema de traducción serían de gran utilidad. Si, de forma transparente o no, el método llamado pudiera utilizar este *codec*, el proceso de traducción entre formatos tendría buena parte de sus problemas resueltos de antemano. El descubrimiento de métodos con los que enlazar se podría hacer de diversas formas como, por ejemplo, empleando el API de introspección de .NET (*Reflection*).



- **Parser PDDL.** La funcionalidad del lenguaje definido en Aran es un subconjunto de la de PDDL. No debería ser complicado desarrollar un *parser* para dicho lenguaje. Para ello sería necesario implementar las interfaces del espacio de nombres `Aran.Representation`. Doblemente positiva sería esta ampliación si, además, el *parser* implementado funcionase en la Xbox 360 y no sólo en PC. Si las herramientas de generación utilizadas produjeran código C# compatible con el .NET Compact Framework esto sería directo. Implementar este *parser* permitiría probar en Aran distintos dominios empleados por la comunidad investigadora internacional, pudiendo incluso comparar los resultados obtenidos con los de otros sistemas de planificación o, simplemente, permitiendo utilizar el sistema para probar prototipos de innovaciones en el campo de la planificación.
- **Traductor de PDDL a XML.** Una alternativa a la implementación de un *parser* PDDL en .NET es la realización de un traductor de este lenguaje al formato XML admitido por Aran. Es común que en entornos de investigación se disponga ya de un *parser* PDDL implementado en el lenguaje con el que más se trabaje (C/C++, LISP, etc.). Esta vía permite reutilizar el trabajo existente, pues el cometido de este traductor sería convertir ficheros PDDL a XML entendibles por Aran, lo cual es independiente del lenguaje en el que se implemente el traductor.
- **Mejora del rendimiento del *parser* XML.** Se ha venido insistiendo en este documento acerca de la ineficiencia con la que, conscientemente, ha sido implementado el actual *parser* XML. Se trata de una implementación funcional pero mejorable, basada en la *serialización* XML estándar de .NET. Esta implementación puede ser reemplazada por otra que haga uso de rutinas propias que trabajen de forma directa con el contenido de los ficheros y, por tanto, sean más eficientes.
- **Traductor inseguro.** La actual implementación del traductor realiza numerosas comprobaciones sobre la entrada de datos que recibe. En otras palabras, no confía en que la implementación del subsistema de representación sea correcta y lleva a cabo verificaciones que le aseguren que no hay errores en los datos. Esto implica una pérdida de eficiencia que puede ahorrarse en entornos en los que se tenga por seguro que la entrada que recibe el traductor será siempre correcta. Un caso típico es, precisamente, el de un videojuego en el cual la entrada del traductor es generada



automáticamente a partir del estado actual del mundo del juego. Si el módulo del videojuego encargado de plasmar el estado del mundo está correctamente implementado (algo que se presupone en la versión final de un juego correctamente depurado) entonces las comprobaciones del actual traductor son innecesarias. Un hipotético *FastTranslator* sería muy útil para casos como este. Por otro lado, ya se ha comentado que la actual implementación del traductor es poco legible. Sería una buena oportunidad para partir de cero y diseñar un traductor con idéntica funcionalidad pero con una elaboración más cuidada. Aquí ya no habría problemas de captación de requisitos, pues éstos serían extraídos directamente de la actual implementación.

- **Optimización de rendimiento en tiempo.** El rendimiento en tiempo es un factor fundamental en el desarrollo de videojuegos. En .NET, como en todas las plataformas de desarrollo, se precisan conocimientos avanzados del funcionamiento a bajo nivel para obtener los mejores resultados en este campo. La actual implementación de Aran podría ser revisada en busca de puntos flacos en este sentido. Sirva como ejemplo de lo comentado una optimización aplicada al algoritmo Best First que permitió reducir en minuto y medio el tiempo empleado por el algoritmo de Dijkstra para resolver el problema 9, lo cual supone más de un 30% de ahorro. La optimización consistió simplemente en evitar la construcción prematura de nuevos nodos dentro del bucle principal. Con ella, se evitó la generación de multitud de objetos efímeros, e innecesarios, reduciendo las intervenciones del recolector de basura y aumentando con ello el rendimiento de forma claramente apreciable.
- **Algoritmos de tiempo real.** Ya se ha expuesto que los requerimientos temporales de Aran dificultan su ejecución en entornos de tiempo real. Sin embargo, esto se debe verdaderamente a que todos los algoritmos implementados actualmente y ejecutados en las pruebas se dejaron trabajar hasta la consecución de un plan completo. Existen numerosos casos en los que la ejecución de planes parciales es una solución alternativa. En lugar de obtener un plan completo, se obtienen sólo las N primeras acciones. Ejecutadas estas acciones, se vuelve a requerir un nuevo plan partiendo del actual estado. Algunos de los algoritmos ya implementados permiten este tipo de ejecución, como los de tipo Best First, siempre y cuando se les pase una



condición de parada basada en tiempo (que, dicho sea de paso, se ha implementado para este proyecto). Ahora bien, aparte de la utilización de estos algoritmos al estilo *anytime*, también podrían incorporarse al catálogo otros preparados específicamente para su uso en tiempo real, como el RTA* [Korf, 1990]. Bastaría con llevar a cabo una implementación que se basase en el mismo modelo de componentes utilizado por los actuales algoritmos, de tal modo que pudiera reaprovechar los componentes existentes, tales como el selector de acciones o la función heurística.

- **Algoritmos de fuerza bruta.** Todos los algoritmos implementados actualmente siguen algún tipo de mecanismo que les permita bien descartar ciertas vías bien retrasar su evaluación. Sin embargo, podría ser interesante incluir en el sistema algoritmos de fuerza bruta, que se limitasen a probar todas las combinaciones posibles sin realizar ningún tipo de optimización ni emplear complejas estructuras en su ejecución. En el análisis de resultados pudo observarse cómo los algoritmos más simples, como Hill Climbing, tenían un mayor rendimiento en número de nodos generados y explorados. Para casos sencillos en los que se sabe de antemano que la respuesta no debería tardar demasiado en ser encontrada, este tipo de algoritmos puede ser una buena alternativa a los actuales, pues no realizan operaciones de optimización de las cuales apenas van a obtener beneficios.
- **Paralelización de tareas.** En el transcurso de esta memoria se ha venido haciendo constante hincapié en las capacidades multiproceso de la Xbox 360 (y de cualquier PC moderno) y, sin embargo, éstas no han sido utilizadas en el desarrollo de Aran. Una de las razones para no haberlo hecho es que, normalmente, en videojuegos se suelen emplear pocos hilos de ejecución, pues complican la sincronización de los distintos elementos que componen el juego. Sin embargo, esto no es óbice para que una posible vía de trabajo futuro sobre Aran sea la implementación de características de paralelización que aprovechen la potencia de las máquinas modernas. Por ejemplo, el tratamiento de los distintos nodos hijos dentro de ciertos algoritmos es una tarea altamente paralelizable. Otro caso claro es el procesamiento de la tabla de selección de acciones cada vez que se quiere conocer cuáles son esos hijos.
- **Condiciones de parada.** Actualmente se manejan dos condiciones de parada con las cuales poder detener la ejecución de los algoritmos: por longitud de plan y por



tiempo. Sería interesante disponer de otras condiciones de uso común como, por ejemplo, una que se dispare cuando se haya alcanzado una cierta cota máxima de memoria consumida. Esta condición, trivial a simple vista, puede no ser tan fácil de implementar si se tiene en cuenta que .NET es un sistema con gestión automática de memoria.

- **Encadenamiento hacia atrás.** El funcionamiento actual de Aran se basa en el encadenamiento hacia delante. No es una elección aleatoria, sino que se debe a que este tipo de encadenamiento permite la elaboración de planes parciales. Aún así, puede ser interesante implementar el encadenamiento hacia atrás. Ello conllevaría la definición de otro formato alternativo para la tabla de selección, así como la implementación de componentes reutilizables que hiciesen uso de ella (por ejemplo, un selector de acciones). También se requeriría de un traductor (o una variación del actual) capaz de producir este nuevo formato de tabla.
- **Otras heurísticas.** En el apartado de resultados pudo verse cómo la heurística actual penaliza el rendimiento de forma notable. En ocasiones puede no necesitarse el uso de una heurística tan compleja, pudiéndose sustituir por alguna otra más sencilla, directa, y eficiente. Por ejemplo, podría implementarse una heurística basada en el número de metas resueltas. En el otro lado se situarían implementaciones más complejas aún que la heurística actual. Por ejemplo, alguna que contemple el coste de las acciones desde el primer momento, y no sólo en la fase final del cálculo.
- **Aplicación práctica.** Aran es un sistema de planificación orientado a videojuegos que, sin embargo, aún no ha sido empleado en juego alguno. Un ejercicio interesante que pondría a prueba su utilidad sería desarrollar un juego en XNA que hiciese uso de él. Otra alternativa sería modificar algún juego ya existente de entre los muchos de código abierto que pueden encontrarse en Internet.
- **Editor gráfico.** No hay mejor forma de extender el uso de una herramienta que facilitando al máximo su uso. Un editor gráfico de dominios y problemas sería ideal para un proyecto de estas características. El modelo empleado en F.E.A.R. [Orkin, 2006] es un buen ejemplo a seguir. En él se da la posibilidad de definir operadores y reutilizarlos como componentes para agruparlos en conjuntos de operadores que definen el comportamiento de los personajes controlados por el juego.

CAPÍTULO 8

REFERENCIAS

Sólo sé que no sé nada



8 Referencias

[aDeSe, 2006] aDeSe, “Estudio de Hábitos y Usos de los Videojuegos 2006. Fase Ómnibus.”, http://www.adese.es/pdf/InformeOMNIBUS_2006.pdf, 2006.

[aDeSe, 2007] aDeSe, “Resultados del Consumo de Videojuegos en 2007”, http://www.adese.es/pdf/DATOS_INDUSTRIA_07.pdf, 2007.

[Barrapunto, 2007] Barrapunto, “¿Qué ofrece el software libre para la programación de videojuegos?”, <http://preguntas.barrapunto.com/preguntas/07/12/13/1055202.shtml>, 13/12/2007.

[BBC, 2006] BBC News, “Xbox outlines 'YouTube for games’”, <http://news.bbc.co.uk/2/hi/technology/4789809.stm>, 14/8/2006.

[Bruce & Veloso, 2002] James Bruce & Manuela Veloso, “Real-Time Randomized Path Planning for Robot Navigation”, <http://www.cs.cmu.edu/~mmv/papers/02iros-rrt.pdf>, 2002.

[Carmack, 2007] John Carmack, “A direct response”, <http://slashdot.org/comments.pl?sid=302231&cid=20671657>, 19/9/2007.

[DeMaria & Wilson, 2002] Rusel DeMaria & Johnny L. Wilson, “High score!: La Historia Ilustrada de los Videojuegos”, McGraw Hill/Osborne, 2002.

[Edelkamp & Helmert, 1999] Stefan Edelkamp & Malte Helmert, “Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length”, http://www.informatik.uni-freiburg.de/~helmert/publications/ECP99_StateEncoding.ps.gz, 1999.

[Gamasutra, 2007a] Gamasutra, “Carmack Gives 360 Next-Gen Edge, Considers DS”, http://www.gamasutra.com/php-bin/news_index.php?story=12351, 10/1/2007.

[Gamasutra, 2007b] Gamasutra, “AIIDE: AILive Showcases Wiimote Recognition, Combat Tools”, http://www.gamasutra.com/php-bin/news_index.php?story=14414, 21/6/2007.



[**GamesIndustry, 2008**] GamesIndustry.biz, “GDC: XNA passes 800,000 mark”,
http://www.gamesindustry.biz/content_page.php?aid=33352, 20/02/2008.

[**GameSpy, 2006**] GameSpy, “F.E.A.R.'s AI Demystified”,
<http://www.gamespy.com/pc/fear/698080p1.html>, 23/03/2006.

[**Ghallab et al., 2004**] Malik Ghallab, Dana Nau & Paolo Traverso, “Automated Planning: Theory and Practice”, Morgan Kaufmann, 2004.

[**González Seco, 2001**] José Antonio González Seco, “El lenguaje de programación C#”, <http://www.josanguapo.com/librosharp2.zip>, 2001.

[**Hoffmann & Nebel, 2001**] Jörg Hoffmann & Bernhard Nebel, “The FF Planning System: Fast Plan Generation Through Heuristic Search”,
<http://www.jair.org/media/855/live-855-1976-jair.pdf>, 2001.

[**Holland, 1975**] John H. Holland, “Adaptation in Natural and Artificial Systems”, 1975.

[**JadEngine, 2007**] Jad Engine Blog, “MDX dies, SlimDX borns, XNA 2.0 appears. And what's to happen in Jade?”,
<http://kartones.net/blogs/jadengine/archive/2007/08/16/mdx-dies-slimdx-borns-xna-2-0-appears-and-what-s-to-happen-in-jade.aspx>, 16/8/2007.

[**Korf, 1990**] Richard E. Korf, “Real-Time Heuristic Search”,
<http://www.cs.ualberta.ca/~bulitko/F06/papers/LRTA.pdf>, 1990.

[**Koza, 1990**] John R. Koza, “Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems”, <http://www.genetic-programming.com/jkpdf/tr1314.pdf>, 1990.

[**La Vanguardia, 2007**] La Vanguardia, “El Parlamento Europeo abre sus puertas a un videojuego español”, <http://www.lavanguardia.es/lv24h/20071212/53418108053.html>, 13/12/2007.

[**Laird & van Lent, 2001**] John E. Laird & Michael van Lent, “Human-level AI's Killer Application: Interactive Computer Games”, <http://www.kddresearch.org/Groups/AI-CBR/Papers/Interactive-computer-games-AIMag22-02-003.pdf>, 2001.



[**McDowel *et al.*, 2006**] Perry McDowell, Rudolph Darken, Joe Sullivan & Erik Johnson, “Delta3D: A Complete Open Source Game and Simulation Engine for Building Military Training Systems”,
<http://www.scs.org/pubs/jdms/vol3num3/JDMSIITSECvol3no3McDowell143-154.pdf>, 2006.

[**Microsoft, 2006a**] Microsoft Press, “Xbox 360 Ushers in Exclusive New Worlds of Entertainment for Most-Anticipated Franchises Ever at X06”,
<http://www.microsoft.com/presspass/press/2006/sep06/09-27X06ShowcasePR.msp>, 27/9/2006.

[**Microsoft, 2006b**] Microsoft Press, “Microsoft Launches CodePlex, a New Collaborative Development Portal”,
<http://www.microsoft.com/presspass/press/2006/jun06/06-27CodePlexPR.msp>, 27/5/2006.

[**Microsoft, 2007**] Microsoft Press, “Global Entertainment Phenomenon ‘Halo 3’ Records More Than \$300 Million in First-Week Sales Worldwide”,
<http://www.microsoft.com/presspass/press/2007/oct07/10-04Halo3FirstWeekPR.msp>, 4/10/2007.

[**MSDN, 2005**] .NET Framework Developer Center, “Design Guidelines for Developing Class Libraries”, [http://msdn2.microsoft.com/en-us/library/ms229042\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms229042(VS.80).aspx), 2005.

[**MSDN, 2006a**] .NET Framework Developer Center, “ECMA C# and Common Language Infrastructure Standards”, <http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx>, 2006.

[**MSDN, 2006b**] XNA Team Blog, “XNA Game Studio Express Has Been Released!”, <http://blogs.msdn.com/xna/archive/2006/12/11/xna-game-studio-express-has-been-released.aspx>, 11/12/2006.

[**MSDN, 2006c**] .NET Compact Framework Team, “Managed Code Performance on Xbox 360 for XNA: Part 2 - GC and Tools”,
<http://blogs.msdn.com/netcfteam/archive/2006/12/22/managed-code-performance-on-xbox-360-for-xna-part-2-gc-and-tools.aspx>, 22/12/2006.



[MSDN, 2006d] .NET Compact Framework Team, “Managed Code Performance on Xbox 360 for XNA: Part 1 - Intro and CPU”,

<http://blogs.msdn.com/netcftteam/archive/2006/12/22/managed-code-performance-on-xbox-360-for-the-xna-framework-1-0.aspx>, 22/12/2006.

[MSDN, 2007a] XNA Team Blog, “Announcing the XNA Game Studio Express Update”, <http://blogs.msdn.com/xna/archive/2007/03/08/announcing-the-xna-game-studio-express-update.aspx>, 8/3/2007.

[MSDN, 2007b] XNA Team Blog, “XNA Game Studio 2.0 Released”, <http://blogs.msdn.com/xna/archive/2007/12/13/xna-game-studio-2-0-released.aspx>, 13/12/2007.

[MSDN, 2008a] XNA Team Blog, “Announcing: XNA Game Studio 3.0 and Zune”, <http://blogs.msdn.com/xna/archive/2008/02/20/announcing-xna-game-studio-3-0-and-zune.aspx>, 20/02/2008.

[MSDN, 2008b] XNA Team Blog, “Announcing: Xbox LIVE community games”, <http://blogs.msdn.com/xna/archive/2008/02/20/announcing-xbox-live-community-games.aspx>, 20/02/2008.

[MSDN, 2008c] XNA Team Blog, “XNA Game Studio Connect Update”, <http://blogs.msdn.com/xna/archive/2008/03/10/xna-game-studio-connect-update.aspx>, 10/03/2008.

[Muñoz-Avila & Fisher, 2004] Héctor Muñoz-Avila & Todd Fisher, “Strategic Planning for Unreal Tournament© Bots”, <http://www.cse.lehigh.edu/~munoz/Publications/AAAIWS04Munozh.PDF>, 2004.

[Orkin, 2004] Jeff Orkin, “Symbolic Representation of Game World State: Toward Real-Time Planning in Games”, <http://web.media.mit.edu/~jorkin/WS404OrkinJ.pdf>, 2004.

[Orkin, 2005] Jeff Orkin, “Agent Architecture Considerations for Real-Time Planning in Games”, <http://web.media.mit.edu/~jorkin/aiide05OrkinJ.pdf>, 2005.



- [Orkin, 2006] Jeff Orkin, “Three States and a Plan: The AI of F.E.A.R.”,
http://www.cs.ualberta.ca/~bulitko/F06/papers/gdc2006_orkin_jeff_fear.pdf, 2006.
- [Rabin *et al.*, 2004] Steve Rabin *et al.*, ”AI Game Programming Wisdom 2”, Charles Riber Media, 2004.
- [Reuters, 2007a] Reuters, “Nintendo becomes Japan's 2nd most valuable company”,
<http://www.reuters.com/article/technology-media-telco-SP/idUST1482820070925>,
25/9/2007.
- [Reuters, 2007b] Reuters, “Acuerdo Vivendi-Activision reúne a 'Guitar Hero' y
'Warcraft’”, <http://lta.reuters.com/article/internetNews/idLTAN0237227820071202>,
2/12/2007.
- [Stentz, 1994] Anthony Stentz, ”Optimal and Efficient Path Planning for Partially-
Known Environments”,
http://www.frc.ri.cmu.edu/projects/mars/publications/original_dstar_icra94.pdf, 1994.
- [Stentz, 1995] Anthony Stentz, ”The Focussed D* Algorithm for Real-Time
Replanning”,
http://www.frc.ri.cmu.edu/projects/mars/publications/focussed_dstar_ijcai95.pdf, 1995.
- [Sweeney, 2006] Tim Sweeney, “The Next Mainstream Programming Language: A
Game Developer’s Perspective”, <http://morpheus.cs.ucdavis.edu/papers/sweeny.pdf>,
2006.
- [The Inquirer, 2007] The Inquirer, “There's barely any difference between an Xbox
360 and a PS3”, [http://www.theinquirer.net/gb/inquirer/news/2007/11/07/xbox-360-ps3-](http://www.theinquirer.net/gb/inquirer/news/2007/11/07/xbox-360-ps3-perform-same)
perform-same, 8/11/2007.
- [Turing, 1950] Alan M. Turing, “Computing Machinery and Intelligence”,
<http://www.cs.umbc.edu/471/papers/turing.pdf>, 1950.

ANEXO A

GRAMÁTICA DEL LENGUAJE

Existe un lenguaje que va más allá de las palabras



9 Anexo A: Gramática del lenguaje

En este anexo se define la gramática del lenguaje de texto empleado para comunicarse con Aran. Se explicará en primer lugar la notación empleada para expresar las reglas que la componen para pasar a continuación a definir cada una de ellas.

9.1 Notación

En la definición de las reglas gramaticales se empleará la notación EBNF¹, teniendo en cuenta las siguientes indicaciones:

- Las reglas son del tipo `ELEMENTO_SINTÁCTICO ::= expansión`.
- Los elementos sintácticos expansibles se denotarán en MAYÚSCULAS y **negrita**.
- Los literales se denotarán '`entre comillas simples`' y en *cursiva*.
- No se especificará regla para el elemento sintáctico `NOMBRE`, que hace referencia a una cadena de caracteres² no vacía.
- No se especificará regla para el elemento sintáctico `NÚMERO`, que hace referencia a un número de 32 bits en coma flotante de precisión simple³.
- Un elemento sintáctico entre corchetes (`[]`) puede aparecer una vez o ninguna.
- Un elemento sintáctico seguido de un asterisco (`*`) puede aparecer cero o más veces.
- El elemento sintáctico `FICHERO_ENTRADA` se corresponde con el contenido de un fichero de entrada.
- El elemento sintáctico `FICHERO_SALIDA` se corresponde con el contenido de un fichero de salida.

9.2 Reglas

```
FICHERO_ENTRADA ::=  
    '<definition>'  
    [DOMINIO]  
    [PROBLEMA]  
    '</definition>'
```

```
DOMINIO ::=  
    '<domain name=" ' NOMBRE ' ">'  
    [TIPOS]  
    [PREDICADOS]
```

¹ Siglas inglesas de “Forma de Backus-Naur Extendida”

² Más información en <http://www.w3.org/TR/xmlschema-2/#string>

³ Más información en <http://www.w3.org/TR/xmlschema-2/#float>



```
[FUNCIONES]
[OPERADORES]
'</domain>'

TIPOS ::= '<types>'
        TIPO*
        '</types>'

TIPO ::=
        '<type name="" NOMBRE [PADRE] '/>'

PADRE ::=
        '" parent="" NOMBRE "'

PREDICADOS ::=
        '<predicates>'
        PREDICADO*
        '</predicates>'

PREDICADO ::=
        '<predicate name="" NOMBRE ">'
        PARÁMETRO*
        '</predicate>'

PARÁMETRO ::=
        '<parameter name="" NOMBRE ' type="" NOMBRE '"/>'

FUNCIONES ::=
        '<functions>'
        FUNCIÓN*
        '</functions>'

FUNCIÓN ::=
        '<function name="" NOMBRE ">'
        PARÁMETRO*
        '</function>'

OPERADORES ::=
        '<operators>'
        OPERADOR*
        '</operators>'

OPERADOR ::=
        '<operator name="" NOMBRE ">'
        [PARÁMETROS]
        [HECHOS]
        [VARIABLES]
        [PRECONDICIONES]
        [PRECONDICIONES_FUNCIONALES]
        [ELIMINACIONES]
        [ADICIONES]
        [EFECTOS_FUNCIONALES]
        '</operator>'

PARÁMETROS ::=
        '<parameters>'
        DECLARACIÓN_OBJETO*
        '</parameters>'

DECLARACIÓN_OBJETO ::=
        '<object name="" NOMBRE ' type="" NOMBRE '"/>'
```



```
HECHOS ::=
    '<facts>'
    DECLARACIÓN_HECHO*
    '</facts>'

DECLARACIÓN_HECHO ::=
    '<fact name="" NOMBRE ' " predicate="" NOMBRE ' ">'
    UTILIZACIÓN_OBJETO*
    '</fact>'

UTILIZACIÓN_OBJETO ::=
    '<object name="" NOMBRE ' "/>'

VARIABLES ::=
    '<variables>'
    DECLARACIÓN_VARIABLE*
    '</variables>'

DECLARACIÓN_VARIABLE ::=
    '<variable name="" NOMBRE ' " function="" NOMBRE ' ">'
    UTILIZACIÓN_OBJETO*
    '</variable>'

PRECONDICIONES ::=
    '<preconditions>'
    UTILIZACIÓN_HECHO*
    '</preconditions>'

UTILIZACIÓN_HECHO ::=
    '<fact name="" NOMBRE ' "/>'

PRECONDICIONES_FUNCIONALES ::=
    '<functionalPreconditions>'
    OPERACIÓN*
    '</functionalPreconditions>'

OPERACIÓN ::=
    '<operation symbol="" OPERADOR ' ">'
    '<operand>'
    CONTENIDO_OPERANDO
    '</operand>'
    '<operand>'
    CONTENIDO_OPERANDO
    '</operand>'
    '</operation>'

OPERADOR ::=
    '=' | '+=' | '-=' | '*=' | '/=' | '+' | '-' | '*' | '/' |
    'lt' | 'lt=' | '==' | '!=' | 'gt=' | 'gt' | 'and' | 'or'

CONTENIDO_OPERANDO ::=
    UTILIZACIÓN_VARIABLE | CONSTANTE | OPERACIÓN

UTILIZACIÓN_VARIABLE ::=
    '<variable name="" NOMBRE ' "/>'

CONSTANTE ::=
    '<constant value="" NÚMERO ' "/>'

ELIMINACIONES ::=
```



```
'<deletions>'
    UTILIZACIÓN_HECHO*
'</deletions>'

ADICIONES ::=
    '<additions>'
        UTILIZACIÓN_HECHO*
    '</additions>'

EFECTOS_FUNCIONALES ::=
    '<functionalEffects>'
        OPERACIÓN*
    '</functionalEffects>'

PROBLEMA ::=
    '<problem name="" NOMBRE "" domain="" NOMBRE "">'
        [HECHOS]
        [VARIABLES]
        [HECHOS_INICIALES]
        [INICIALIZACIONES_FUNCIONALES]
        [HECHOS_META]
        [METAS_FUNCIONALES]
        [MÉTRICA]
    '</problem>'

HECHOS_INICIALES ::=
    '<initialFacts>'
        UTILIZACIÓN_HECHO*
    '</initialFacts>'

INICIALIZACIONES_FUNCIONALES ::=
    '<functionalInitializations>'
        OPERACIÓN*
    '</functionalInitializations>'

HECHOS_META ::=
    '<goalFacts>'
        UTILIZACIÓN_HECHO*
    '</goalFacts>'

METAS_FUNCIONALES ::=
    '<functionalGoals>'
        OPERACIÓN*
    '</functionalGoals>'

MÉTRICA ::=
    '<metric type="" TIPO_MÉTRICA "">'
        '<target>'
            CONTENIDO_OPERANDO
        '</target>'
    '</metric>'

TIPO_MÉTRICA ::=
    'max' | 'min'

FICHERO_SALIDA ::=
    '<definition>'
        '<plan>'
            ACCIÓN*
        '</plan>'
    '</definition>'
```



```
ACCIÓN ::=  
  '<action operator=" ' NOMBRE ' "> '  
    UTILIZACIÓN_OBJETO*  
  '</action> '
```

ANEXO B

**EJEMPLOS DE UTILIZACIÓN DEL
LENGUAJE**

El ejemplo es la lección que todos los hombres pueden leer



10 Anexo B: Ejemplos de utilización del lenguaje

Este anexo contiene dos ejemplos de utilización del lenguaje de texto empleado para comunicarse con Aran. Concretamente, se trata de la definición XML del dominio empleado en el apartado de resultados y del ejemplo utilizado en dicho apartado para ilustrar los principales elementos del dominio. Hay que resaltar el hecho de que estos ejemplos no hacen uso del lenguaje en toda su extensión, el cual puede consultarse en el “Anexo A: Gramática del lenguaje”. Sin más que añadir, a continuación se muestran los ejemplos mencionados en el orden en que han sido citados.

10.1 Ejemplo de dominio

```
<definition>
  <domain name="GuideTheGuy">
    <types>
      <type name="Node"/>
      <type name="UnblockableNode" parent="Node"/>
      <type name="HammerNode" parent="UnblockableNode"/>
      <type name="KeyNode" parent="UnblockableNode"/>
      <type name="BlockableNode" parent="Node"/>
      <type name="Color"/>
      <type name="Item"/>
      <type name="Hammer" parent="Item"/>
      <type name="Key" parent="Item"/>
    </types>
    <predicates>
      <predicate name="Connected">
        <parameter type="Node" name="source"/>
        <parameter type="Node" name="destination"/>
      </predicate>
      <predicate name="Teleconnected">
        <parameter type="UnblockableNode" name="source"/>
        <parameter type="UnblockableNode" name="destination"/>
      </predicate>
      <predicate name="Blocked">
        <parameter type="BlockableNode" name="node"/>
      </predicate>
      <predicate name="Unblocked">
        <parameter type="BlockableNode" name="node"/>
      </predicate>
      <predicate name="NodeColor">
        <parameter type="BlockableNode" name="node"/>
        <parameter type="Color" name="color"/>
      </predicate>
      <predicate name="HammerColor">
        <parameter type="Hammer" name="hammer"/>
        <parameter type="Color" name="color"/>
      </predicate>
      <predicate name="At">
        <parameter type="Item" name="item"/>
        <parameter type="UnblockableNode" name="node"/>
      </predicate>
      <predicate name="GuyAt">
        <parameter type="Node" name="node"/>
      </predicate>
    </predicates>
  </domain>
</definition>
```



```
</predicate>
<predicate name="Has">
  <parameter type="Item" name="item"/>
</predicate>
</predicates>
<functions>
  <function name="Time"/>
  <function name="Weight">
    <parameter type="Hammer" name="hammer"/>
  </function>
  <function name="GuyWeight"/>
  <function name="UnblockTime">
    <parameter type="BlockableNode" name="node"/>
  </function>
</functions>
<operators>
  <operator name="GoTo">
    <parameters>
      <object type="Node" name="source"/>
      <object type="UnblockableNode" name="destination"/>
    </parameters>
    <facts>
      <fact name="nodesConected" predicate="Connected">
        <object name="source"/>
        <object name="destination"/>
      </fact>
      <fact name="guyAtSource" predicate="GuyAt">
        <object name="source"/>
      </fact>
      <fact name="guyAtDestination" predicate="GuyAt">
        <object name="destination"/>
      </fact>
    </facts>
    <variables>
      <variable name="time" function="Time"/>
      <variable name="guyWeight" function="GuyWeight"/>
    </variables>
    <preconditions>
      <fact name="nodesConected"/>
      <fact name="guyAtSource"/>
    </preconditions>
    <deletions>
      <fact name="guyAtSource"/>
    </deletions>
    <additions>
      <fact name="guyAtDestination"/>
    </additions>
    <functionalEffects>
      <operation symbol="+=">
        <operand><variable name="time"/></operand>
        <operand><variable name="guyWeight"/></operand>
      </operation>
    </functionalEffects>
  </operator>
  <operator name="GoToUnblocked">
    <parameters>
      <object type="Node" name="source"/>
      <object type="BlockableNode" name="destination"/>
    </parameters>
    <facts>
      <fact name="nodesConected" predicate="Connected">
```



```
<object name="source"/>
<object name="destination"/>
</fact>
<fact name="guyAtSource" predicate="GuyAt">
  <object name="source"/>
</fact>
<fact name="guyAtDestination" predicate="GuyAt">
  <object name="destination"/>
</fact>
<fact name="destinationUnblocked" predicate="Unblocked">
  <object name="destination"/>
</fact>
</facts>
<variables>
  <variable name="time" function="Time"/>
  <variable name="guyWeight" function="GuyWeight"/>
</variables>
<preconditions>
  <fact name="nodesConected"/>
  <fact name="guyAtSource"/>
  <fact name="destinationUnblocked"/>
</preconditions>
<deletions>
  <fact name="guyAtSource"/>
</deletions>
<additions>
  <fact name="guyAtDestination"/>
</additions>
<functionalEffects>
  <operation symbol="+=">
    <operand><variable name="time"/></operand>
    <operand><variable name="guyWeight"/></operand>
  </operation>
</functionalEffects>
</operator>
<operator name="Teleport">
  <parameters>
    <object type="UnblockableNode" name="source"/>
    <object type="UnblockableNode" name="destination"/>
  </parameters>
  <facts>
    <fact name="nodesTeleconected" predicate="Teleconnected">
      <object name="source"/>
      <object name="destination"/>
    </fact>
    <fact name="guyAtSource" predicate="GuyAt">
      <object name="source"/>
    </fact>
    <fact name="guyAtDestination" predicate="GuyAt">
      <object name="destination"/>
    </fact>
  </facts>
  <variables>
    <variable name="time" function="Time"/>
  </variables>
  <preconditions>
    <fact name="nodesTeleconected"/>
    <fact name="guyAtSource"/>
  </preconditions>
  <deletions>
    <fact name="guyAtSource"/>
  </deletions>
</operator>
```



```
</deletions>
<additions>
  <fact name="guyAtDestination"/>
</additions>
<functionalEffects>
  <operation symbol="+=">
    <operand><variable name="time"/></operand>
    <operand><constant value="1"/></operand>
  </operation>
</functionalEffects>
</operator>
<operator name="Unblock">
  <parameters>
    <object type="Node" name="source"/>
    <object type="BlockableNode" name="destination"/>
    <object type="Hammer" name="hammer"/>
    <object type="Color" name="color"/>
  </parameters>
  <facts>
    <fact name="nodesConected" predicate="Connected">
      <object name="source"/>
      <object name="destination"/>
    </fact>
    <fact name="guyAtSource" predicate="GuyAt">
      <object name="source"/>
    </fact>
    <fact name="destinationBlocked" predicate="Blocked">
      <object name="destination"/>
    </fact>
    <fact name="hasHammer" predicate="Has">
      <object name="hammer"/>
    </fact>
    <fact name="destinationColoured" predicate="NodeColor">
      <object name="destination"/>
      <object name="color"/>
    </fact>
    <fact name="hammerColoured" predicate="HammerColor">
      <object name="hammer"/>
      <object name="color"/>
    </fact>
    <fact name="destinationUnblocked" predicate="Unblocked">
      <object name="destination"/>
    </fact>
  </facts>
  <variables>
    <variable name="time" function="Time"/>
    <variable name="guyWeight" function="GuyWeight"/>
    <variable name="hammerWeight" function="Weight">
      <object name="hammer"/>
    </variable>
    <variable name="destinationUnblockTime"
      function="UnblockTime">
      <object name="destination"/>
    </variable>
  </variables>
  <preconditions>
    <fact name="nodesConected"/>
    <fact name="guyAtSource"/>
    <fact name="destinationBlocked"/>
    <fact name="hasHammer"/>
    <fact name="destinationColoured"/>
```



```
<fact name="hammerColoured"/>
</preconditions>
<deletions>
  <fact name="destinationBlocked"/>
  <fact name="hasHammer"/>
</deletions>
<additions>
  <fact name="destinationUnblocked"/>
</additions>
<functionalEffects>
  <operation symbol="+=">
    <operand><variable name="time"/></operand>
    <operand>
      <variable name="destinationUnblockTime"/>
    </operand>
  </operation>
  <operation symbol="-=">
    <operand><variable name="guyWeight"/></operand>
    <operand><variable name="hammerWeight"/></operand>
  </operation>
</functionalEffects>
</operator>
<operator name="GetHammer">
  <parameters>
    <object type="Hammer" name="hammer"/>
    <object type="HammerNode" name="node"/>
  </parameters>
  <facts>
    <fact name="hammerAtNode" predicate="At">
      <object name="hammer"/>
      <object name="node"/>
    </fact>
    <fact name="guyAtNode" predicate="GuyAt">
      <object name="node"/>
    </fact>
    <fact name="hasHammer" predicate="Has">
      <object name="hammer"/>
    </fact>
  </facts>
  <variables>
    <variable name="time" function="Time"/>
    <variable name="guyWeight" function="GuyWeight"/>
    <variable name="hammerWeight" function="Weight">
      <object name="hammer"/>
    </variable>
  </variables>
  <preconditions>
    <fact name="hammerAtNode"/>
    <fact name="guyAtNode"/>
  </preconditions>
  <deletions>
    <fact name="hammerAtNode"/>
  </deletions>
  <additions>
    <fact name="hasHammer"/>
  </additions>
  <functionalEffects>
    <operation symbol="+=">
      <operand><variable name="time"/></operand>
      <operand><constant value="1"/></operand>
    </operation>
```



```
<operation symbol="+=">
  <operand><variable name="guyWeight"/></operand>
  <operand><variable name="hammerWeight"/></operand>
</operation>
</functionalEffects>
</operator>
<operator name="GetKey">
  <parameters>
    <object type="Key" name="key"/>
    <object type="KeyNode" name="node"/>
  </parameters>
  <facts>
    <fact name="keyAtNode" predicate="At">
      <object name="key"/>
      <object name="node"/>
    </fact>
    <fact name="guyAtNode" predicate="GuyAt">
      <object name="node"/>
    </fact>
    <fact name="hasKey" predicate="Has">
      <object name="key"/>
    </fact>
  </facts>
  <variables>
    <variable name="time" function="Time"/>
  </variables>
  <preconditions>
    <fact name="keyAtNode"/>
    <fact name="guyAtNode"/>
  </preconditions>
  <deletions>
    <fact name="keyAtNode"/>
  </deletions>
  <additions>
    <fact name="hasKey"/>
  </additions>
  <functionalEffects>
    <operation symbol="+=">
      <operand><variable name="time"/></operand>
      <operand><constant value="1"/></operand>
    </operation>
  </functionalEffects>
</operator>
</operators>
</domain>
</definition>
```

10.2 Ejemplo de problema

```
<definition>
  <problem name="Example" domain="GuideTheGuy">
    <objects>
      <object type="KeyNode" name="node1-1"/>
      <object type="UnblockableNode" name="node1-2"/>
      <object type="BlockableNode" name="node1-3"/>
      <object type="KeyNode" name="node1-4"/>
      <object type="UnblockableNode" name="node1-5"/>
      <object type="BlockableNode" name="node2-1"/>
      <object type="UnblockableNode" name="node2-5"/>
      <object type="HammerNode" name="node3-1"/>
    </objects>
  </problem>
</definition>
```




```
<object type="UnblockableNode" name="node3-2"/>
<object type="UnblockableNode" name="node3-3"/>
<object type="HammerNode" name="node3-4"/>
<object type="UnblockableNode" name="node3-5"/>
<object type="Color" name="blue"/>
<object type="Color" name="red"/>
<object type="Key" name="key1"/>
<object type="Key" name="key2"/>
<object type="Hammer" name="hammer1"/>
<object type="Hammer" name="hammer2"/>
</objects>
<facts>
  <fact name="way1-1#1-2" predicate="Connected">
    <object name="node1-1"/>
    <object name="node1-2"/>
  </fact>
  <fact name="way1-2#1-1" predicate="Connected">
    <object name="node1-2"/>
    <object name="node1-1"/>
  </fact>
  <fact name="way1-2#1-3" predicate="Connected">
    <object name="node1-2"/>
    <object name="node1-3"/>
  </fact>
  <fact name="way1-3#1-2" predicate="Connected">
    <object name="node1-3"/>
    <object name="node1-2"/>
  </fact>
  <fact name="way1-3#1-4" predicate="Connected">
    <object name="node1-3"/>
    <object name="node1-4"/>
  </fact>
  <fact name="way1-4#1-3" predicate="Connected">
    <object name="node1-4"/>
    <object name="node1-3"/>
  </fact>
  <fact name="way1-4#1-5" predicate="Connected">
    <object name="node1-4"/>
    <object name="node1-5"/>
  </fact>
  <fact name="way1-5#1-4" predicate="Connected">
    <object name="node1-5"/>
    <object name="node1-4"/>
  </fact>
  <fact name="way1-1#2-1" predicate="Connected">
    <object name="node1-1"/>
    <object name="node2-1"/>
  </fact>
  <fact name="way2-1#1-1" predicate="Connected">
    <object name="node2-1"/>
    <object name="node1-1"/>
  </fact>
  <fact name="way1-5#2-5" predicate="Connected">
    <object name="node1-5"/>
    <object name="node2-5"/>
  </fact>
  <fact name="way2-5#1-5" predicate="Connected">
    <object name="node2-5"/>
    <object name="node1-5"/>
  </fact>
  <fact name="way2-1#3-1" predicate="Connected">
```



```
<object name="node2-1"/>
<object name="node3-1"/>
</fact>
<fact name="way3-1#2-1" predicate="Connected">
  <object name="node3-1"/>
  <object name="node2-1"/>
</fact>
<fact name="way2-5#3-5" predicate="Connected">
  <object name="node2-5"/>
  <object name="node3-5"/>
</fact>
<fact name="way3-5#2-5" predicate="Connected">
  <object name="node3-5"/>
  <object name="node2-5"/>
</fact>
<fact name="way3-1#3-2" predicate="Connected">
  <object name="node3-1"/>
  <object name="node3-2"/>
</fact>
<fact name="way3-2#3-1" predicate="Connected">
  <object name="node3-2"/>
  <object name="node3-1"/>
</fact>
<fact name="way3-2#3-3" predicate="Connected">
  <object name="node3-2"/>
  <object name="node3-3"/>
</fact>
<fact name="way3-3#3-2" predicate="Connected">
  <object name="node3-3"/>
  <object name="node3-2"/>
</fact>
<fact name="way3-3#3-4" predicate="Connected">
  <object name="node3-3"/>
  <object name="node3-4"/>
</fact>
<fact name="way3-4#3-3" predicate="Connected">
  <object name="node3-4"/>
  <object name="node3-3"/>
</fact>
<fact name="way3-4#3-5" predicate="Connected">
  <object name="node3-4"/>
  <object name="node3-5"/>
</fact>
<fact name="way3-5#3-4" predicate="Connected">
  <object name="node3-5"/>
  <object name="node3-4"/>
</fact>
<fact name="portal1-2#3-5" predicate="Teleconnected">
  <object name="node1-2"/>
  <object name="node3-5"/>
</fact>
<fact name="portal3-5#1-2" predicate="Teleconnected">
  <object name="node3-5"/>
  <object name="node1-2"/>
</fact>
<fact name="node1-3Color" predicate="NodeColor">
  <object name="node1-3"/>
  <object name="blue"/>
</fact>
<fact name="node1-3Blocked" predicate="Blocked">
  <object name="node1-3"/>
```



```
</fact>
<fact name="node2-1Color" predicate="NodeColor">
  <object name="node2-1"/>
  <object name="red"/>
</fact>
<fact name="node2-1Blocked" predicate="Blocked">
  <object name="node2-1"/>
</fact>
<fact name="hammer1Color" predicate="HammerColor">
  <object name="hammer1"/>
  <object name="blue"/>
</fact>
<fact name="hammer1Location" predicate="At">
  <object name="hammer1"/>
  <object name="node3-1"/>
</fact>
<fact name="hammer2Color" predicate="HammerColor">
  <object name="hammer2"/>
  <object name="red"/>
</fact>
<fact name="hammer2Location" predicate="At">
  <object name="hammer2"/>
  <object name="node3-4"/>
</fact>
<fact name="key1Location" predicate="At">
  <object name="key1"/>
  <object name="node1-1"/>
</fact>
<fact name="hasKey1" predicate="Has">
  <object name="key1"/>
</fact>
<fact name="key2Location" predicate="At">
  <object name="key2"/>
  <object name="node1-4"/>
</fact>
<fact name="hasKey2" predicate="Has">
  <object name="key2"/>
</fact>
<fact name="guyLocation" predicate="GuyAt">
  <object name="node3-3"/>
</fact>
</facts>
<initialFacts>
  <fact name="way1-1#2-1"/>
  <fact name="way2-1#1-1"/>
  <fact name="way1-1#1-2"/>
  <fact name="way1-2#1-1"/>
  <fact name="way1-2#1-3"/>
  <fact name="way1-3#1-2"/>
  <fact name="way1-3#1-4"/>
  <fact name="way1-4#1-3"/>
  <fact name="way1-4#1-5"/>
  <fact name="way1-5#1-4"/>
  <fact name="way1-5#2-5"/>
  <fact name="way2-5#1-5"/>
  <fact name="way2-1#3-1"/>
  <fact name="way3-1#2-1"/>
  <fact name="way2-5#3-5"/>
  <fact name="way3-5#2-5"/>
  <fact name="way3-1#3-2"/>
  <fact name="way3-2#3-1"/>
</initialFacts>
```



```
<fact name="way3-2#3-3"/>
<fact name="way3-3#3-2"/>
<fact name="way3-3#3-4"/>
<fact name="way3-4#3-3"/>
<fact name="way3-4#3-5"/>
<fact name="way3-5#3-4"/>
<fact name="portal1-2#3-5"/>
<fact name="portal3-5#1-2"/>
<fact name="node1-3Color"/>
<fact name="node1-3Blocked"/>
<fact name="node2-1Color"/>
<fact name="node2-1Blocked"/>
<fact name="hammer1Color"/>
<fact name="hammer1Location"/>
<fact name="hammer2Color"/>
<fact name="hammer2Location"/>
<fact name="key1Location"/>
<fact name="key2Location"/>
<fact name="guyLocation"/>
</initialFacts>
<goalFacts>
  <fact name="hasKey1"/>
  <fact name="hasKey2"/>
</goalFacts>
<variables>
  <variable name="node1-3UnblockTime" function="UnblockTime">
    <object name="node1-3"/>
  </variable>
  <variable name="node2-1UnblockTime" function="UnblockTime">
    <object name="node2-1"/>
  </variable>
  <variable name="time" function="Time"/>
  <variable name="guyWeight" function="GuyWeight"/>
</variables>
<functionalInitializations>
  <operation symbol="=">
    <operand><variable name="node1-3UnblockTime"/></operand>
    <operand><constant value="1"/></operand>
  </operation>
  <operation symbol="=">
    <operand><variable name="node2-1UnblockTime"/></operand>
    <operand><constant value="1"/></operand>
  </operation>
  <operation symbol="=">
    <operand><variable name="guyWeight"/></operand>
    <operand><constant value="1"/></operand>
  </operation>
</functionalInitializations>
<metric type="min">
  <target><variable name="time"/></target>
</metric>
</problem>
</definition>
```

ANEXO C
MANUAL DE USUARIO

*La belleza es muy superior al genio
No necesita explicación*



11 Anexo C: Manual de usuario

En este anexo se procederá a explicar someramente las distintas **formas que un usuario tiene para utilizar Aran, desde el nivel de abstracción más alto al más bajo**. Se comenzará, pues, comentando el uso del comando `aran.exe` para pasar seguidamente a indicar cómo emplear la misma funcionalidad desde el código de una aplicación o videojuego. Se concluirá apuntando la forma en la que acceder a los componentes que forman el sistema para utilizarlos sin ningún tipo de adorno que pueda impedir expresar su potencial de forma directa. En todos estos casos, las descripciones que se harán tendrán el objetivo de iniciar a un usuario, de forma rápida pero poco profunda, en el uso de Aran, al estilo de las populares **guías de inicio rápido** (*quick start guides*). Para mayor detalle, habrá de consultarse la documentación generada al efecto para el comando y las bibliotecas.

11.1 Utilización del comando desde la consola del sistema

La forma más sencilla de utilizar Aran es a través de su comando, **`aran.exe`**, ejecutándolo sin ningún argumento. El comando buscará así en el directorio actual los ficheros **`domain.xml`** y **`problem.xml`**, ejecutará el planificador y guardará el plan resultante, de haberlo, en el fichero **`plan.xml`**. Por pantalla informará acerca de si ha podido o no encontrar un plan, así como de los posibles errores o advertencias que puedan darse.

```
C:\>aran
Aran, interfaz en línea de comandos del planificador XML
Versión 1.0
Copyright © 2008 Abel García Plaza

Este programa no ofrece ABSOLUTAMENTE NINGUNA GARANTÍA. Es
software libre, y se le invita a redistribuirlo bajo ciertas
condiciones. Use la opción /exportLicense para más detalle.

¡Plan encontrado!
```

Ilustración 74: Ejecución por defecto del comando `aran.exe`

Si se ejecuta desde un *script*, puede saberse el estado con el que finalizó a través de su **código de retorno**. Este código valdrá 0 si todo ha ido correctamente o tomará un valor negativo en caso contrario. Pueden consultarse los valores exactos para este código haciendo uso de la opción de ayuda del propio comando (`aran /?`). Nótese, en



cualquier caso, que la forma de invocar el comando depende del sistema operativo y la implementación del .NET Framework sobre la que se esté trabajando¹. Los ejemplos aquí mostrados se corresponden con el SO Windows y la implementación del .NET Framework de Microsoft. Nótese también que la ejecución de este comando tiene como prerequisite tener instalado el .NET Framework 2.0, o equivalente, en el sistema.

Con esta configuración **por defecto**, la planificación se lleva a cabo empleando el **algoritmo** Enforced Hill Climbing en primer lugar y el A* en caso de que el primero no encuentre solución. Por otro lado, el comando predefine el **tipo raíz** `object`, padre de todos los que se declaren en el fichero del dominio. También predefine el predicado de **desigualdad del tipo raíz**, `notEqual`, que recibe dos argumentos de tipo `object`, y declara en el problema un hecho de dicho predicado por cada par de objetos de nombre diferente. En la práctica, esto quiere decir que, por ejemplo, puede utilizarse el tipo `object` en el parámetro de un predicado cuando se quiera poder pasar como argumento de dicho parámetro cualquier tipo de objeto. También quiere decir que se puede utilizar el predicado `notEqual` para establecer condiciones de desigualdad entre dos objetos cualesquiera.

La configuración por defecto puede ser alterada mediante el uso de las distintas **opciones** del comando. A continuación se listan las más comunes, que no las únicas. Para todas ellas se muestra entre paréntesis su forma abreviada, si la tienen.

- **/help** (/?) muestra la ayuda.
- **/domain** (/d) permite especificar la ruta del fichero de definición del dominio.
- **/problem** (/p) permite especificar la ruta del fichero de definición del problema.
- **/out** (/o) permite especificar la ruta de salida del fichero de definición del plan.
- **/overwrite** permite sobrescribir el fichero de definición del plan si éste ya existe.
- **/algorithm** (/a) permite especificar el algoritmo a utilizar durante la búsqueda del plan. Si se utiliza más de una vez, los algoritmos indicados se ejecutarán en secuencia hasta que alguno de ellos dé con una solución. Este parámetro tiene las siguientes subopciones:
 - **/beamWidth** (/b) permite indicar la anchura del haz para el algoritmo Beam Search. Por defecto se toma una anchura de 2.

¹ Por ejemplo, en sistemas Unix no puede omitirse la extensión `.exe` del nombre del fichero.



- **/heuristicWeight** (/w) permite especificar el peso de la heurística para el algoritmo Weighted A*. Por defecto se toma un peso de 2.
- **/maxPlanLength** permite especificar cuántas acciones, como máximo, puede tener el plan devuelto. Sirve para devolver planes parciales. A este respecto, nótese que la implementación realizada de algoritmos como Hill Climbing o Beam Search no garantiza la devolución del mejor plan parcial.
- **/maxTime** permite especificar el tiempo máximo que puede dedicársele a la búsqueda del plan. También sirve para devolver planes parciales.

La Ilustración 75 muestra algunos **ejemplos** de uso del comando.

```
C:\>aran /?  
C:\>aran /d dominio.xml /p problema.xml /maxPlanLength 15  
C:\>aran /a /a HillClimbing BeamSearch /b 10 /maxTime 2000  
C:\>aran /overwrite /a WeightedAStar /w 1.5
```

Ilustración 75: Ejemplos de uso del comando **aran.exe**

El primero de los ejemplos se limitaría a mostrar la ayuda. El segundo, ejecutaría el planificador tomando como dominio el definido en el fichero `dominio.xml` y como problema el escrito en el fichero `problema.xml`. En ese caso, el plan devuelto nunca tendría más de 15 acciones y podría ser parcial. El tercer ejemplo modifica la secuencia de algoritmos por defecto de tal modo que se ejecute, en primer lugar, el Hill Climbing y, en caso de no encontrar solución, el Beam Search con un ancho de haz de 10. El tiempo total disponible para la ejecución de ambos algoritmos sería de 2 segundos (se indica en milisegundos), pudiendo devolverse un plan parcial en caso de que en ese tiempo no se hubiera encontrado aún una solución completa. Finalmente, el último ejemplo especifica que se emplee únicamente el algoritmo Weighted A* con un peso de la heurística de 1,5. Al mismo tiempo, solicita la sobrescritura del fichero de salida del plan, si éste ya existiera.

11.2 Utilización de la funcionalidad del comando desde una aplicación o videojuego

Para utilizar la misma funcionalidad de la que dispone el comando desde el código de una aplicación o videojuego, lo primero que debe hacerse es incluir una **referencia** a la biblioteca `Aran.Planning.Xml.dll` en el proyecto en que se quiera usar (ver Ilustración 47).



Una vez referenciada la biblioteca, debe importarse su espacio de nombres (ver Ilustración 43). Esto, en C#, se hace mediante la directiva `using`. Hecho esto, debe crearse un objeto de tipo **XmlPlanner** e invocar a su método **Plan**. Si todo sucede sin problemas, el plan se escribirá en el fichero de salida. En caso contrario, se lanzará una excepción de tipo **XmlPlanNotFoundException** que contendrá los mensajes de error.

```
using System;

using Aran.Planning.Xml;

namespace Test {

    public static class Program {

        public static void Main() {
            try {
                XmlPlanner planner = new XmlPlanner();
                planner.Plan();
            } catch (XmlPlanNotFoundException e) {
                foreach (XmlPlannerMessage message in e.Messages) {
                    Console.WriteLine(message.Text);
                }
            }
        }
    }
}
```

Ilustración 76: Ejemplo de uso de la biblioteca `Aran.Planning.Xml.dll`

La Ilustración 77 muestra un ejemplo de cómo utilizar la biblioteca en su forma más simple. Como sucediera con el comando, si no se modifican las propiedades del objeto `planner`, se toman los valores de la **configuración por defecto**. Las propiedades de la clase `XmlPlanner` se corresponden con las opciones del comando. La Tabla 18 muestra los nombres de aquellas que se corresponden con las opciones comentadas en el apartado anterior.



Opción	Propiedad
/domain	DomainPath
/problem	ProblemPath
/out	OutputPath
/overwrite	OverwriteOutput
/algorithm	SearchAlgorithms
/maxPlanLength	MaxPlanLength
/maxTime	MaxTime

Tabla 18: Correspondencia entre las opciones del comando y la biblioteca

No todas las opciones del comando están reflejadas directamente en la clase `XmlPlanner`. Algunas derivan la configuración a clases auxiliares. Por ejemplo, el establecimiento de la secuencia de algoritmos de búsqueda se hace a través de un objeto de la clase `XmlPlannerSearchAlgorithmSequence`, accesible desde la propiedad `SearchAlgorithms` de la clase `XmlPlanner` (ver Ilustración 43).

```
planner.DomainPath = "dominio.xml";  
planner.ProblemPath = "problema.xml";  
planner.MaxTime = 2000;  
planner.SearchAlgorithms.ReplaceAllBy(  
    XmlPlannerSearchAlgorithmId.HillClimbing);  
planner.SearchAlgorithms.Add(  
    XmlPlannerSearchAlgorithmId.WeightedAStar);  
planner.SearchAlgorithms[1].HeuristicWeight = 1.3f;
```

Ilustración 77: Ejemplo de parametrización de `Aran.Planning.Xml.dll`

La Ilustración 77 muestra un **ejemplo** de modificación de la configuración por defecto. En él, se indica que los ficheros de definición del dominio y problema son `dominio.xml` y `problema.xml`. También se asigna un tiempo máximo de 2 segundos para buscar una solución. Por último, se establece que en dicha búsqueda se utilice primeramente el algoritmo Hill Climbing y, en caso de no encontrar una solución, hacer uso del Weighted A* con un peso de 1,3 para la heurística.

11.3 Utilización directa de los componentes del sistema desde una aplicación o videojuego

Esta última forma de utilizar Aran es, con mucha diferencia, la más complicada de todas pero, al tiempo e igualmente distanciada de las anteriores, la más flexible.



Hacer uso directo de los componentes del sistema significa un mayor control sobre todo el proceso. **Aquí no existe un ente que cumpla el papel del planificador**, sino que se deben utilizar los componentes que lo formarían. En las anteriores vías de utilización simplemente había que indicar una serie de parámetros y pedir que tuviera lugar la ejecución. El planificador, siempre con entrada/salida basada en ficheros XML, se ocupaba de realizar la gestión de los mismos, traducir el contenido de dominio y problema, crear los algoritmos de búsqueda, establecer su secuencia de ejecución, llevar a cabo la búsqueda de la solución y, en caso de encontrarla, traducirla y escribirla en el fichero de salida. Aquí nada de eso se hace automáticamente. **Todo se realiza manualmente**, pues es la única forma de garantizar la total libertad del desarrollador para elegir el modo en que quiere utilizar los componentes del sistema. Él determinará qué tipo de representación del conocimiento necesita, qué algoritmos (incluso podría implementar uno propio), qué heurística (porque también podría implementar una propia), qué formato de salida, etc. Todo. Los componentes de Aran están a su entera disposición para hacer únicamente su cometido directo, aquél del que se sirve el usuario para lograr sus fines.

El proceso de **utilización de Aran a bajo nivel**, que es de lo que aquí se está tratando, consistiría, pues, en los siguientes pasos:

1. Creación del sistema de representación.
2. Creación del traductor.
3. Ejecución del traductor.
4. Creación del algoritmo de búsqueda.
5. Ejecución del algoritmo de búsqueda.
6. Extracción del plan.

A continuación se expondrá un ejemplo de utilización que cubre todas estas fases y que, por simplicidad, se ha decidido ir suministrando de forma fragmentada, sin mostrar el tratamiento de las posibles excepciones. Téngase en cuenta, por tanto, que las porciones de código mostradas no tienen sentido por sí mismas y requieren a todas las demás o, al menos, a las que les preceden. Por otro lado, obsérvese que, en ocasiones, algunas interfaces no se corresponden exactamente con las descripciones gráficas expuestas en el apartado de diseño detallado. Recuérdese que aquellas descripciones fueron simplificadas para mayor claridad. Aquí se utilizarán las interfaces completas,



afectando principalmente al número de parámetros recibido por ciertos métodos o constructores o a la *instanciación* de los tipos genéricos.

11.3.1 Creación del sistema de representación

Como se ha explicado en este documento, el subsistema de representación es un conjunto de clases que implementan las interfaces recogidas en el espacio de nombres `Aran.Representation`. Esto quiere decir que la creación del subsistema de representación puede hacerse por dos vías. La sencilla es **crear un objeto de tipo `IKnowledgeProvider`** a partir de una implementación existente. Más complicada pero, a su vez, adaptable a las necesidades del usuario, es la vía de la implementación de un `IKnowledgeProvider` propio, con todo lo que ello conlleva, pues este tipo no es sino la raíz de una jerarquía de interfaces que también deben ser implementadas, salvo que existan implementaciones reutilizables de las mismas. En el ejemplo se utilizará la implementación del subsistema de **representación basado en XML**, para lo cual se requerirá importar (y referenciar la biblioteca que lo contiene) el espacio de nombres `Aran.Representation.Xml`.

```
XmlParser parser = new XmlParser();  
Stream domainFile = File.OpenRead("dominio.xml");  
Stream problemFile = File.OpenRead("problema.xml");  
parser.DomainStream = domainFile;  
parser.ProblemStream = problemFile;
```

Ilustración 78: Ejemplo de creación de un sistema de representación

La Ilustración 78 ejemplifica la creación del sistema de representación del conocimiento basado en texto XML. Nótese que las propiedades del sistema, un *parser* en este caso, no son ahora cadenas de caracteres que indiquen la ruta de los ficheros. Aquí se trabaja a un nivel inferior de abstracción, por lo que **se trabaja con** flujos de datos (*streams*), y no con ficheros. En este caso el flujo proviene de ficheros, pero el origen podría ser cualquier otro, como por ejemplo un buffer en memoria. Para hacer uso de las clases `Stream` y `File` es necesario importar el espacio de nombres `System.IO`.

11.3.2 Creación del traductor

Una vez se ha creado el sistema de representación, se puede proceder a hacer lo propio con el traductor, del que el sistema de representación es parte. Como tal, debe ser



asociado al traductor durante la inicialización de este último. Este paso requiere la importación del espacio de nombres `Aran.Translation`, en el que se agrupan las clases del traductor.

```
Translator<string, string, string, string, string, string, string, string>  
    translator = new Translator<string, string, string, string,  
        string, string, string>(parser);
```

Ilustración 79: Ejemplo de creación del traductor

En la Ilustración 79 puede apreciarse el código necesario para realizar la construcción. A pesar de su longitud, únicamente se **está llamando al constructor de la clase `Translator`, pasándole el `parser`** creado en el paso anterior para que se asocie a él. Lo que complica la lectura del código es la especificación de los valores que deben tomar los parámetros genéricos del traductor. En este caso, y puesto que todos los identificadores de la representación XML son de tipo `string`, todos los parámetros toman ese tipo como valor, pero debe tenerse presente que estos valores serán distintos dependiendo del sistema de representación que se esté utilizando. Estos mismos valores se propagarán a lo largo del ejemplo.

Opcionalmente, pueden modificarse algunas **propiedades del traductor** antes de ser utilizado (ver Ilustración 40). Y es que debe tenerse presente que el traductor es genérico y, como tal, desconoce a priori el tipo de los identificadores que manejará en tiempo de ejecución. Por tanto, no puede establecer un valor por defecto para propiedades como el tipo raíz. Esto quiere decir que, al contrario de lo que sucede con las formas de uso tratadas con anterioridad, aquí la configuración por defecto ni tiene predefinido un **tipo raíz** ni, claro, **predicado de desigualdad** alguno.

```
translator.RootTypeEnabled = true;  
translator.RootTypeId = "object";  
translator.InequalityPredicatesEnabled = true;  
translator.InequalityPredicates.Add("notEqual", "object");
```

Ilustración 80: Ejemplo de configuración del traductor

La Ilustración 80 muestra cómo configurar el traductor para obtener una configuración igual a la que establece por defecto el planificador XML.



11.3.3 Ejecución del traductor

La ejecución del traductor es bien sencilla. Simplemente debe invocarse su método **Translate** y recoger su resultado. Dicho resultado requiere de la especificación de argumentos para los parámetros genéricos de la clase que lo representa, **TranslationResult**. Tales argumentos han de ser los mismos que se especificaron al crear el traductor, en este caso todos `strings`.

```
TranslationResult<string, string, string, string, string, string,  
string> translation = translator.Translate();
```

Ilustración 81: Ejemplo de ejecución del traductor

En caso de fallo, la ejecución del traductor desembocará en el lanzamiento de una excepción. Dicha excepción puede ser de tipo **RepresentationException** si el error proviene del sistema de representación, o de tipo **GenerationException**, si ha tenido lugar durante la generación del resultado de la traducción. No se incluirá aquí código de ejemplo sobre el tratamiento de las mismas. Sólo decir que cada cual aporta un tipo de información diferente. `RepresentationException` presenta el formato estándar de toda excepción de .NET, pero `GenerationException` contiene una lista de mensajes provenientes del traductor que indican qué ha sucedido exactamente (ver Ilustración 41).

Algunos de los tipos devueltos en el resultado de la traducción pertenecen a los espacios de nombres `Aran.Core` y `Aran.Functional`, que deben ser importados.

11.3.4 Creación del algoritmo de búsqueda

Cuando se trabaja a este nivel con Aran, no existe ningún módulo encargado de la gestión de los algoritmos de búsqueda. Éstos deben ser creados desde cero y por supuesto, no forman parte de ningún tipo de secuencia. Si así se precisara, el usuario debería escribir su propio código para inicializar los algoritmos y ejecutar uno tras otro hasta dar con la solución. Aquí sólo se hablará acerca de **cómo crear un algoritmo de búsqueda**, tomando el Weighted A* como ejemplo. Conviene insistir en que todo lo demás queda en manos del usuario.

Como puede verse en la Ilustración 37, los algoritmos de búsqueda en Aran siguen un diseño orientado a componentes. **Crear un algoritmo conlleva hacer lo**



propio, en primer lugar, con los componentes que formarán parte de él. En el caso del Weighted A* se requiere un proveedor de estado inicial, una función de validación de meta, un generador de sucesores, una función de coste, una heurística y un comparador de igualdad entre estados. Para ello, se requerirán los espacios de nombres `Aran.Searching` y `Aran.Searching.Specialized`.

El único **proveedor de estado inicial** implementado en Aran es uno que se limita a devolver siempre el mismo estado que se le pasó en el constructor, aunque el usuario es libre de realizar cualquier otra implementación según sus necesidades. La Ilustración 82 muestra cómo crear el mencionado proveedor.

```
IInitialStateProvider<State> initialStateProvider = new  
    ConstantInitialStateProvider<State>(translation.InitialState);
```

Ilustración 82: Ejemplo de creación del proveedor del estado inicial

Con la **función de validación de meta** sucede algo similar. En Aran sólo se proporciona una implementación, quedando a criterio del usuario la necesidad o no de implementar cualquier otro componente con el mismo cometido. Esta implementación, cuya creación se expone en la Ilustración 83, da la posibilidad de asignar una **condición de ruptura**, tal y como se explicó en este documento. En la mencionada ilustración se emplea una condición de ruptura basada tiempo. Así, si tras un segundo desde el momento en que se crea la condición aún no se ha encontrado solución, ésta corresponderá al primer estado que se compruebe.

```
DateTime oneSecond = DateTime.Now.AddMilliseconds(1000);  
ICondition breakCondition = new DateTimeCondition(oneSecond);  
IGoalStateVerifier<State> goalVerifier = new GoalStateVerifier(  
    breakCondition, translation.GoalFacts,  
    translation.FunctionalGoal);
```

Ilustración 83: Ejemplo de creación de la función de validación de meta

También se proporciona una única implementación del **generador de sucesores**. Ésta, como puede verse en la Ilustración 84, requiere de un selector de acciones, por lo que ha de ser creado previamente mediante la, también única, implementación de la que dispone Aran.



```
ActionSelector actionSelector = new ConjunctionActionSelector(  
    translation.OperatorCount, translation.PreconditionCounts,  
    translation.ActionCount, translation.ActionCounts,  
    translation.ActionActivationTable);  
IStateSuccessorsProvider<State> successorsGenerator =  
    new StateSuccessorsProvider(translation.Actions, actionSelector);
```

Ilustración 84: Ejemplo de creación del generador de sucesores

Como se indicó en su momento, única la **función de coste** implementada consiste en un adaptador de la función métrica del problema. En la Ilustración 85 se muestra cómo crearla.

```
IStateCostProvider<State> costCalculator =  
    new FunctionalStateCostProvider(translation.Metric);
```

Ilustración 85: Ejemplo de creación de la función de coste

Si el generador de sucesores requería de un selector de acciones, a la **función heurística** le sucede lo propio con el planificador relajado. La Ilustración 86 muestra cómo hacerlo.

```
RelaxedPlanner relaxedPlanner = new RelaxedPlanner(  
    translation.FactCount, translation.Actions,  
    translation.OperatorCount, translation.ActionCounts,  
    translation.ActionActivationTable, translation.GoalCount,  
    translation.GoalFacts);  
IStateHeuristicCostProvider<State> heuristicCalculator =  
    new RelaxedPlanHeuristic(relaxedPlanner, costCalculator);
```

Ilustración 86: Ejemplo de creación de la función heurística

En cuanto a la implementación de la **función de asignación de padre** para los estados de Aran, su construcción se lleva a cabo tal y como se ejemplifica en la Ilustración 87.

```
IStateParentSetter<State> parentSetter =  
    new StateParentalManager();
```

Ilustración 87: Ejemplo de creación de la función de asignación de padre

La forma en la que se haga la **comparación entre estados** es muy importante. Una mala comparación puede provocar bucles infinitos. La Ilustración 88 recoge el modo de crear un comparador que evite estas situaciones, pues sólo considerará parte del estado algunas variables, concretamente las que influyan en operaciones condicionales. Aunque esta forma puede utilizarse siempre, se recomienda utilizar



`EqualityComparer<State>.Default` cuando todas las variables influyan en operaciones condicionales y un `FactsOnlyStateEqualityComparer` cuando no lo haga ninguna. Así se mejora la eficiencia, pues se ahorran operaciones innecesarias.

```
IEqualityComparer<State> equalityComparer =  
    new SomeVariablesStateEqualityComparer(  
        translation.ConditionalVariables);
```

Ilustración 88: Ejemplo de creación del comparador de igualdad entre estados

Una vez se tienen todos los componentes necesarios, puede procederse a la creación del **algoritmo de búsqueda**. En la Ilustración 89 se ejemplifica este proceso con la creación del algoritmo **Weighted A***, con un peso de 1,5 para la heurística.

```
ISearchAlgorithm<State> searchAlgorithm =  
    new WeightedAStarAlgorithm<State>(initialStateProvider,  
        goalVerifier, successorsGenerator, costCalculator,  
        heuristicCalculator, parentSetter, 1.5f, equalityComparer);
```

Ilustración 89: Ejemplo de creación del algoritmo de búsqueda

11.3.5 Ejecución del algoritmo de búsqueda

Comparado con el complejo proceso de construcción, la ejecución del algoritmo de búsqueda es realmente simple. Tal y como puede verse en la Ilustración 90, basta con ejecutar el método **Search** del mismo, que devolverá el estado final resultante de la búsqueda, no el plan necesario para alcanzarlo.

```
State finalState = searchAlgorithm.Search();
```

Ilustración 90: Ejemplo de ejecución del algoritmo de búsqueda

En caso de que el algoritmo no encontrase solución, lanzaría una excepción del tipo **ResultNotFoundException**. Si el algoritmo no ha encontrado la solución por causas propias, entonces la excepción no enlazará con ninguna otra. En caso contrario, habrá de consultarse su propiedad `InnerException` para conocer la causa real de no haber encontrado solución. En condiciones normales, esto es, entendiendo que la implementación está libre de errores, el único caso en el que puede darse esta situación es cuando la búsqueda consuma toda la memoria disponible. Entonces, la excepción contenida como `InnerException` será del tipo **OutOfMemoryException**.



11.3.6 Extracción del plan

Puesto que la ejecución del algoritmo de búsqueda no devuelve el plan necesario para alcanzar un estado meta, sino el estado en sí, se hace necesario un proceso posterior de extracción de dicho plan. Para ello, **los estados disponen de un enlace a su predecesor**, su padre, a través del cual puede conseguirse la cadena completa de estados por los que pasa la ejecución del plan. Puesto que la cadena va del estado final al inicial, se hace necesario invertirla. Con la secuencia invertida, bastará entonces con **extraer el identificador de la acción que se aplicó para alcanzar cada uno de los estados y, a partir de él, obtener la acción a la que se refiere**. Para ello debe utilizarse el decodificador resultante de la traducción.

En la Ilustración 91, se crea un array de idéntica longitud al plan, en el que guardar cada una de las acciones decodificadas. La longitud se extrae del último estado de la cadena, el devuelto por el algoritmo de búsqueda. Tras ello, se recorre la secuencia de estados en orden inverso, obteniendo el identificador de la acción aplicada para alcanzar el estado en cuestión y decodificándola para obtener el objeto de tipo `IInstance` que la representa. Dicho objeto es guardado en la posición correspondiente del array que representa el plan y permite conocer tanto el identificador del operador al que pertenece la acción como sus argumentos (`IInstance` es idéntico al tipo `IIdentifiedInstance` de la Ilustración 33, pero sin la propiedad `Id`).

```
int actionId;
State currentState = finalState;
IInstance<string, string>[] plan = new int[finalState.PlanLength];
for (int i = (planLength - 1); i >= 0; i--) {
    actionId = currentState.AppliedAction.Id;
    plan[i] = actionDecoder.GetInstance(actionId);
    currentState = currentState.Parent;
}
```

Ilustración 91: Ejemplo de extracción del plan

*Cuando llegamos a la meta,
creemos que el camino ha sido el bueno*

