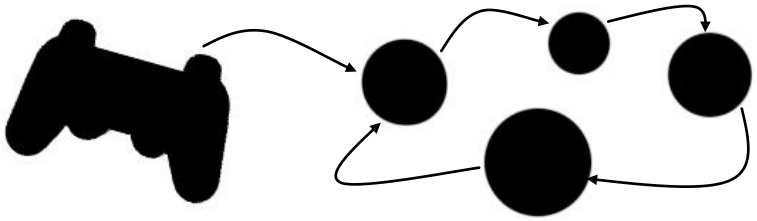
UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

API de gestión de Inteligencia Artificial basada en las Máquinas de Estados Finitos en C#



Autor : **José Alapont Luján**

Director : **Dr. D. Ramón Mollá Vayá**

Septiembre de 2014

Agradecimientos

Quisiera manifestar mi más profundo y sincero agradecimiento a todas las personas que han contribuido de alguna forma a la realización del presente trabajo.

Especialmente, al Dr. D. Ramón Mollá Vayá, director de este proyecto, por el continuo seguimiento, por la orientación que me ha ofrecido y por su vocación, ofreciéndome ayuda incluso en los momentos en los que resultaba muy difícil hacerlo.

También quiero expresar mi agradecimiento al Dr. D. Vicente Julián Inglada, director del posgrado, por su capacidad y admirable predisposición para ayudar en cualquier momento y sobre cualquier cuestión.

También quiero dar las gracias a mis compañeros de proyecto y amigos, Steve Rossius y Javier López Punzano, por hacer más llevaderos estos meses de trabajo y contribuir con sus ideas.

Por último, agradecer a mi familia y amigos su comprensión, ánimo y fuerza necesaria para alcanzar los objetivos.

A todos ellos, muchas gracias.

Abstract

En el presente trabajo se expone y discute el estado del arte relativo al modelo computacional basado en las máquinas de estados finitos como técnica de Inteligencia Artificial en videojuegos y se ofrece el diseño, especificación y posterior prueba de una API basada en este paradigma, en lenguaje de programación C#, utilizando el motor de videojuegos Unity3D y el entorno de desarrollo MonoDevelop integrado.

Este proyecto constituye en realidad una herramienta cuya creación se deduce de otro proyecto mayor, basado en el desarrollo de un videojuego y con dos personas más, que fue ideado como tesis conjunta. Esta propuesta ambiciosa se pospuso por su extensión, y se estudió el conjunto de utilidades que podían llegar a desarrollarse y sirvieran de utilidad en un futuro, de forma que surgieron tres módulos que conformarían las tres tesinas pertinentes.

Finalmente, se ha llevado a cabo uno de esos tres módulos comentados, relativo a la Inteligencia Artificial a emplear en un posible juego, y para probar su funcionamiento, se ha llevado a cabo una serie de demostraciones.

Por otra parte, se ofrece una guía de usuario con una sección de ejemplos prácticos con el fin de orientar al público potencial de la aplicación.

Tabla de contenidos

CAPÍTULO 1. Introducción

En este capítulo se expone una introducción al concepto de Inteligencia Artificial orientado a los videojuegos, además de su influencia en éstos, la motivación para la realización de este proyecto y los objetivos a alcanzar.

1.1 Introducción

El área de la informática y las nuevas tecnologías está experimentando un constante cambio en la actualidad debido a los continuos progresos en la investigación que se han ido sucediendo especialmente desde mediados de siglo XX.

Los avances en la programación informática, con la inclusión de algoritmos basados en la resolución de problemas complejos, así como en la investigación basada en la psicología cognitiva y la lógica matemática, son la antesala de la aparición de los llamados Sistemas Inteligentes, que basan su capacidad de proceso y metodología en la Inteligencia Artificial (IA).

Existe una serie de puntos de vista que explican y validan el concepto de IA. Según *Stuart Russell* y *Peter Norvig* [1] dicho concepto se explica en dos dimensiones: la primera entiende la Inteligencia Artificial como una serie de procesos pensantes o basados en el razonamiento y la segunda como comportamientos que emulan la capacidad de acción humana.

Los autores dividían a su vez estas dos dimensiones en cuatro tipos de Inteligencia Artificial. Con definiciones que los dotan de sentido, se pueden observar en la siguiente tabla:

Dimensión 1: RACIONALIDAD	Dimensión 2: COMPORTAMIENTO
Sistemas que actúan racionalmente	**Sistemas que actúan como humanos**
(Nilsson, 1998), "La IA (…) se entiende como el comportamiento inteligente en artefactos" (Poole et al, 1998), "La inteligencia computacional es el estudio del diseño de agentes inteligentes"	(Rich y Knight, 1994), "La IA es el estudio de cómo lograr que las computadoras realicen tareas que, por el momento, los humanos hacen mejor" (Definida también así por Stuart en 1996) Nebendah (1988) y Delgado (1998) La definían como el campo de estudio que se enfoca en la explicación y emulación de la conducta inteligente (…).
Sistemas que piensan racionalmente	**Sistemas que piensan como humanos**
(Charniak y McDermott, 1985), "El estudio de las facultades mentales a través del uso de modelos computacionales"	(Haugeland, 1985) "(…) intento de que las máquinas piensen, que tengan cerebro, literalmente"

Tabla 1. Puntos de vista de la IA

La dimensión basada en racionalidad (uso de la razón) implica una combinación de matemáticas e ingeniería, mientras que la basada en el comportamiento

(humanas) requiere de ciencia empírica, confirmación experimental y suposición de hipótesis.

Así pues, a modo de resumen, podría definirse la Inteligencia Artificial como un conjunto de técnicas y/o métodos de proceso que pueden llegar a transmitir comportamiento inteligente una vez adheridas a una entidad computacional.

En la actualidad, parece más frecuente el hecho de continuar con el estudio teórico de los contenidos y/o metodología ya investigada que se basa en la Inteligencia Artificial, que en la idea de proponer nuevas teorías.

En los últimos años, se han ido puliendo numerosas áreas de investigación pertenecientes a la Inteligencia Artificial como el reconocimiento del habla y reconocimiento manuscrito, fundamentalmente basadas en los **modelos ocultos de Markov** (HMMs) [2], permitiendo el avance hacia a la industria, los sistemas basados en **planificación** de los años 70, programas que resolvían pequeños problemas y mediante los cuales ha sido posible progresar actualmente en la organización temporal en fábricas o tareas espaciales, entre otras cosas, y de forma similar se ha avanzado en las áreas de **Robótica**, **Visión por Computador**, **Aprendizaje** y **representación del conocimiento**.

La ingeniería informática también ha sido capaz de crear máquinas con una velocidad de cómputo y capacidad de memoria impensable hace unos años. Gracias a las áreas de investigación mencionadas y a estos progresos que la industria del hardware ha sido capaz de lograr, el grado de aplicación de la Inteligencia Artificial ha ido creciendo considerablemente.

Una de las industrias en las que ha tenido una influencia importante es en la de los videojuegos. Si bien es cierto que el concepto de Inteligencia Artificial en informática no es estrictamente el mismo que el empleado en los videojuegos, sí que se utilizan metodologías que son directamente técnicas extraídas del primer campo.

En los videojuegos, el término queda relativizado, y se interpreta la Inteligencia Artificial como todo aquello que dota de comportamiento a un personaje y que no necesariamente se centra en la mejor solución:

> *"Academic AI is usually concerned with making rational decisions, that is, the best or most correct (given that there might not be a best) situation. In contrast, game AI focuses on acting "human", with much less dependence on total rationality."*
>
> *[3] (Brian Schwab, AI Game Engine Programming, p. 10)*

Pese a la utilización flexible del término expuesto, se utilizan técnicas estrictas de Inteligencia Artificial ampliamente conocidas y extendidas en el mundo de la

informática. Quizá la más utilizada sea la técnica basada en Máquinas de Estados Finitos (Finite State Machines, FSMs) [4], y esto se debe su facilidad de uso y comprensión y su carácter tan general.

"Within game AI, it is generally recognized that finite-state machines (FSMs) are the most common software pattern"

(Steve Rabin, Introduction To Game Development, p. 530)

Esta tesina se basa en el paradigma de las máquinas de estados finitos, partiendo de una idea sencilla, se pretende extender el modelo básico y ofrecer una API de control sobre la capa de Inteligencia Artificial de un videojuego o por extensión, de cualquier proyecto de desarrollo de software que requiera su uso.

1.2 Motivación

Según lo comentado anteriormente, las técnicas de Inteligencia Artificial se han ido aplicando en diferentes áreas de la informática, y por tanto, han llegado a diferentes sectores de la industria.

Esta memoria de trabajo se ha enfocado desde el punto de vista de los videojuegos, aunque la implementación y posterior uso de la técnica puedan ser de utilidad en cualquier proyecto de desarrollo.

Para este fin, las máquinas de estados finitos constituyen un modelo de implementación muy extendido en proyectos Software que requieren mecanismos de control sencillos sobre la capa de Inteligencia Artificial.

Aunque el conocimiento sobre estas máquinas de estados sea amplio y el paradigma de autómatas esté presente en numerosos proyectos, los usos reales de las variantes (conocidas a nivel teórico) o extensiones de una máquina con carácter determinista son prácticamente inexistentes en la industria del desarrollo de software y más concretamente, en los videojuegos.

Siguiendo con lo expuesto por *Steve Rabin*, los mecanismos ad-hoc, los cuales parecen generar bastante controversia debido a su estructura pobre en términos de IA, basados en máquinas de estados y que se utilizan en este tipo de proyectos son numerosos, o al menos, el uso de mecanismos que implícitamente se comportan como tal: el comportamiento de una entidad software se basa en situaciones concretas que se dan en un escenario y tiempo determinados en el que se producen cambios. La escena (para una unidad software) es definitivamente modelable mediante una máquina de estados sensible a eventos o cambios que podrían alterar o no el comportamiento de la entidad a la que afectan.

Este carácter estructural de la técnica permite tomar control sobre cualquier situación que dispare eventos; dota de autonomía a una entidad abstracta dentro de un escenario sujeto a un constante cambio. Así pues, en definitiva, es una

potencial forma de modelado de un comportamiento o, yendo más allá, una manera interesante de estructurar la Inteligencia Artificial de un videojuego.

"Yet, it would be a mistake not to initially investigate whether FSMs could solve a portion of your AI needs"

(Steve Rabin, Introduction to Game Development p. 536)

En resumen, se describe la implementación de una herramienta de gestión de Inteligencia Artificial basada en el paradigma de las máquinas de estados finitos que permita otorgar comportamientos con relativa facilidad y comodidad a entidades software que así lo requieran. Los motivos que han impulsado su desarrollo son los siguientes:

- **La introducción al mundo de los videojuegos**, como objetivo personal.
- **Los buenos y rápidos resultados que otorga el uso de la técnica a nivel básico**. Como se ha comentado, la utilización sencilla del paradigma cumple en muchas ocasiones con el objetivo de dotar sensación de inteligencia.
- **La popularidad de las Máquinas de Estados Finitos como técnica utilizada en los videojuegos**. Pese a sus defectos, facilita mecanismos para estructurar adecuadamente la Inteligencia Artificial de un videojuego.
- **La inexistencia de una API utilizable para este cometido**. Hasta el momento, personalmente no he encontrado nada que pueda ser utilizado como una herramienta adicional y público en un proyecto de desarrollo.
- **Las posibilidades y potencial que ofrece su uso**. La utilización básica de la herramienta, así como de las extensiones del paradigma.

1.3 Objetivos

Teniendo en cuenta lo expuesto como motivación, considero necesario e interesante por todo ello la realización de este proyecto.

El propósito de esta API es realizar una revisión del paradigma de las máquinas de estados finitos y expandirlo, de forma que sea posible utilizar la técnica en cualquier proyecto de desarrollo de software que así lo requiera brindando la posibilidad de dar más y variados enfoques en el plano de la Inteligencia Artificial.

Como se ha comentado, no se parte de cero en la utilización de las FSM como técnica de IA. El uso de las máquinas de estados finitos en el desarrollo de videojuegos está bastante extendido, aunque su uso queda bastante limitado en lo funcional y por tanto considero necesaria una revisión y expansión de la fórmula para otorgar más facilidades y puntos de vista al diseño e implementación de comportamientos.

Se listan a continuación los objetivos que se persiguen desde el inicio de la realización del trabajo.

1.3.1 Objetivos generales

- Realizar un análisis profundo del uso de las máquinas de estados finitos como técnica de IA en videojuegos.

- Criticar el estado del arte de la técnica y aportar modificaciones que distingan significativamente al proyecto.

- Conformar una API que proporcione utilidades y variantes para el control de la capa de IA en un proyecto de desarrollo de software, enfocando su creación desde el punto de vista de los videojuegos.

- Continuar con la formación a nivel personal y profesional dentro del marco de los videojuegos.

1.3.2 Objetivos específicos

- Conocer las técnicas de IA aplicadas a videojuegos en la actualidad.

- Ampliar los conocimientos entorno al lenguaje de programación C#.

- Aprender en profundidad el desarrollo de un proyecto software utilizando el motor de videojuegos Unity3D.

- En lo relativo a la API creada:

    - Utilizar las Máquinas de Estados Finitos como técnica estructural de la capa de IA en un juego corto o demostración.

    - Facilitar el diseño e inserción de Máquinas de Estados Finitos mediante el uso de un parser de información y el lenguaje de etiquetado XML.

- Aportar variantes de la Máquina clásica que permitan el diseño de Máquinas de Estados Finitos diferentes.

- Dar soporte para la anidación o jerarquía de máquinas independientemente de su tipo o clase.

CAPÍTULO 2. Estado del arte: máquinas de estados finitos

En esta sección introduzco el estado del arte en la que se basa el núcleo de la tesina: las máquinas de estados finitos (*Finite State Machine* - FSM), como una técnica de Inteligencia Artificial muy extendida y utilizada en la actualidad en el mundo de los videojuegos y aplicable al desarrollo de software general.

2.1 Introducción

Un autómata finito o máquina de estados finitos es un modelo computacional capaz de generar una salida a partir de una entrada. Este modelo está compuesto por entidades, tales como un alfabeto comprendido por la máquina, unos estados y una serie de transiciones entre éstos.

Su funcionamiento se basa en el proceso de una serie de caracteres de entrada (alfabeto) por una función de transición. La máquina parte de un estado definido como inicial, y leyendo estos caracteres, alcanza un estado de aceptación que puede constituir la salida.

Desde comienzos del siglo XX se ha utilizado este modelo, que posteriormente, a mediados de siglo, se extendería el paradigma con la aparición del concepto no-determinista y su utilización para fines prácticos crecería con la aparición del sistema operativo Unix, en los años setenta.

2.2 Las FSM como técnica de IA

En esta sección se expone y discute el estado del arte referente a las máquinas de estados finitos: posibles definiciones, tipos de máquinas, ventajas e inconvenientes y sus aplicaciones en el mundo de los videojuegos, pues es el enfoque de uso práctico que se ha tomado como referencia.

2.2.1 En la actualidad

Hoy en día, en lo relativo a la programación de IA, la técnica más utilizada en los videojuegos es la de las máquinas de estados finitos.

Esta metodología organizacional ofrece al diseñador la posibilidad de dividir la problemática principal en subproblemas más pequeños cuyas soluciones resultan más accesibles, de manera que resulta más cómodo llevar a cabo la programación relativa a la IA gracias a la naturaleza flexible y escalable de la técnica.

Una máquina de estados está compuesta por estados, transiciones entre ellos y acciones posibles.

El presente trabajo se basa en lo comentado anteriormente. El núcleo reside en la lectura de información y su posterior volcado a una estructura de datos de tipo grafo que constituirá la propia máquina de estados a consultar; cada objeto recorre

la máquina que tiene asociada. En la fase de parsing, se extrae información de una serie de documentos de texto etiquetado (en este caso de tipo xml) que guardan el diseño de cada una de las máquinas de estados, definiendo los estados, transiciones y acciones en cada uno de ellos, así como las peculiaridades de cada tipo de máquina. Posteriormente, esta información constituye el grafo/máquina a recorrer (API pbFSM). La definición de esta estructura también es posible realizarla desde programa en lugar de utilizar como fuente un fichero de texto externo, aunque resulta más cómoda la primera forma.

No se ha encontrado bibliografía específica de los juegos que hacen uso de la tecnología, pues en la mayoría de los casos, no es información de carácter público. Igualmente, en la siguiente sección se mencionan algunos de los juegos conocidos por hacer uso de las FSM.

2.2.2 Definición

En su libro [5], *Bran Selic*, *Garth Gullekson y Paul T. Ward* entendían una máquina de estados como:

- Un conjunto de eventos de entrada
- Un conjunto de eventos de salida
- Un conjunto de estados
- Una función que mapea estados y la entrada a la salida.
- Una función que mapea estados y entradas a estados (función de transición de estado)
- La descripción de un estado inicial

Una máquina de estados finitos tiene un número limitado de estados posibles (una máquina de estados infinita puede ser concebida teóricamente pero no a nivel práctico). Puede ser utilizada como herramienta de desarrollo para aproximarse o solucionar problemas y/o como un camino formal para describir una solución de utilidad para desarrolladores y mantenimiento de sistemas.

Por otro lado, según *Samarjit Chakraborty* en [6], una definición formal sería definir una FSM como una 5-tupla $(Q, \Sigma, q_0, \delta, F)$ donde:

- *Q es un conjunto de estados;*
- *Σ es un alfabeto;*
- *$q_0 \in Q$ es el estado inicial;*
- *$\delta: Q \times \Sigma \longrightarrow Q$ es una función de transición.*
- *$F \subseteq Q$ es un conjunto de estados finales o aceptación*

De acuerdo con una adaptación al marco de la IA a nivel práctico, una máquina de estados o FSM (*Finite State Machine*) es una entidad abstracta compuesta por

estados (siendo uno de ellos el estado de partida o inicial), transiciones que relacionan a dichos estados (con un estado origen y uno destino), un vocabulario de entrada y acciones a realizar en cada transición o estado concretos.

Como comentaba anteriormente, es una técnica muy utilizada en el mundo de los videojuegos, siendo ejemplos claros de ello algunos títulos como Quake, *Unreal 2*, o *The Sims*, entre otros [7].

Una forma de modelado de IA en un videojuego, podría ser aquella en la que un NPC sea gestionado por una FSM y que además cuente con otros algoritmos computacionalmente más agresivos en su toma de decisiones, como la búsqueda espacial o en estados (*crash and turn*, *random*, algoritmos A/A*), planificación, basados en la posición (Mapas de influencia, Terreno inteligente, Análisis de terreno, etcétera). Es decir, es una técnica que puede combinarse con otras muchas para sustentar la inteligencia artificial de un personaje.

En principio, una FSM puede tener tantos estados como requiera (si un personaje constituye una entidad muy compleja), aunque esto puede desencadenar una explosión de estados debido a la complejidad. A priori, sólo se puede estar en un estado a la vez (luego hay variantes del paradigma que permiten camuflar esto) y siempre es necesario un estado inicial.

En cuanto a las transiciones, simplemente unen estados de manera unidireccional, es decir, si un estado A puede transitar a B y viceversa, se tratan como transiciones diferentes. Para que una transición tenga lugar, deben cumplirse una serie de condiciones que disparen este movimiento o que la activen (Por ejemplo, un personaje estará en el estado "Atacando" hasta que le quede el 40% de su energía, momento en el que mandará el evento "Escapar" y activará la transición correspondiente (a través de la transición "Escape") al nuevo estado "Huyendo"). Este último caso es muy sencillo, pero podría complicarse si fuera necesario.

Una FSM debe definir una serie de acciones a realizar, ya sea para un estado o para una transición concretos. En cualquier caso, las hay de cuatro tipos: de entrada, acciones que se ejecutan al entrar a un estado; de salida, acciones que se ejecutan al salir de un estado y liberan recursos; de estado, acciones que se ejecutan en el propio estado regularmente; y de transición, acciones que se ejecutan después de la salida de un estado y antes de la entrada en otro, es decir, en la propia transición.

Las bases de esta metodología son su carácter organizacional y su capacidad de gestión (como se ha comentado, es una técnica de carácter muy general y simple que puede utilizarse para dar solución a comportamientos sencillos y más sofisticados), de manera que puede ser usada en cualquier ámbito como técnica de control y consulta a una estructura de datos auxiliar. Esto es una ventaja por su aplicabilidad, pero en el caso de los videojuegos, es una técnica dependiente del

tipo de juego o personaje (requiere programación ad-hoc), y nos otorga capacidad de control simplificando el proceso de creación/asociación del comportamiento.

2.2.3 Pros y contras

El uso de las máquinas de estados finitos puede tener muchas ventajas pero también inconvenientes que se deben conocer.

En lo referente a las ventajas que ofrece, destaca la facilidad de uso, pues simplemente recibe eventos y responde a ellos, su velocidad, ya que responde a meras comprobaciones y actúa en consecuencia con un coste asequible, además de su capacidad de expansión y adaptabilidad a problemas.

Además, hace factible la combinación con otras técnicas de IA dado su implícito carácter organizativo (se distribuye en estados que pueden gozar de autonomía) y son sencillas para modelar comportamientos que no requieran de una sofisticación elevada o bajos en cuanto a complejidad.

Por otro lado, también aparecen inconvenientes, pues su número limitado de estados puede llegar a hacerlas predecibles (aunque con la ampliación del paradigma esto puede ser camuflado), se basan en reglas que son meras comprobaciones o cambios simples del entorno en el que se sitúan, además de que su uso frente a un problema muy complejo elevaría exponencialmente el número de estados disponibles haciéndola incontrolable o inútil frente a situaciones demasiado exigentes en términos de complejidad.

A continuación se muestra una tabla-resumen de los beneficios y limitaciones que puede ofrecer el uso del paradigma como técnica de IA.

Ventajas	Inconvenientes
• Facilidad de uso • Rapidez • Útil para comportamientos sencillos • Flexibilidad • Adaptabilidad • Posibilita combinación de técnicas • Programación sencilla	• Comportamientos predecibles • Basadas en reglas simples • Complejidad eleva nº de estados • No válida para problemas complejos

Tabla 2. Ventajas e inconvenientes de uso

2.2.4 Tipos de FSM conocidos

No existe una única máquina de estados. Dependiendo de las peculiaridades de un personaje concreto (en un videojuego) o problema (una FSM debe poder aplicarse a cualquier división de problemas dado su carácter organizacional), se pueden utilizar máquinas de estados diferentes y lograr diferentes resultados (con comportamientos más o menos sofisticados).

A continuación se exponen los diferentes tipos de máquinas de estados finitos contempladas en [3] *(Brian Schwab 2004)* y he basado el estudio y posterior desarrollo en las ideas que propone como extensiones del paradigma clásico.

- **Máquina Clásica**

Se muestra a continuación un grafo referente a una máquina de estados finitos clásica:

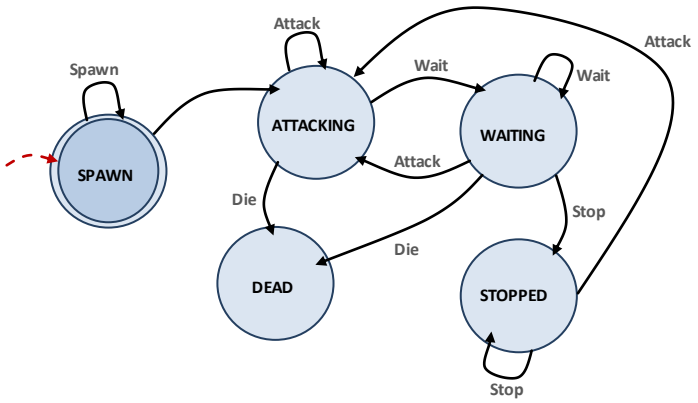


Ilustración 1. Máquina de estados finitos clásica determinista

Esta FSM está compuesta por varios estados que determinan por programa la acción de un NPC (*Non-Playable Character*) enemigo. Se corresponde con una FSM básica o clásica, en donde la división de estados supone la división del comportamiento.

El estado SPAWN, relativo a la aparición del monstruo y por tanto estado inicial de la máquina, ATTACKING, relativo al ataque, WAITING, estado relativo a la espera de un objetivo, STOPPED, estado visitado cuando un objetivo muere o cuando necesitamos energía, y el estado DEAD, relativo a la muerte del NPC.

Por otro lado están las transiciones entre ellos, en este caso, bastante intuitivas. SPAWN conecta con ATTACKING mediante la transición ATTACK. Aparecen otras como DIE, STOP y WAIT, que relacionan otros estados, además de los propios bucles.

El diseño relativo a esta máquina de estados (en este caso el parsing de su xml correspondiente) se especifica en secciones posteriores, así como la API que instancia una nueva máquina/grafo basándose en dicho diseño.

Desde el diseño (xml) se especifica qué acción corresponde a qué estado/transición. En este caso, y como se ha introducido antes, cada estado tiene tres acciones asociadas (se pueden incluir hasta tres), una de entrada,

otra de estado y otra de salida, y cada transición tiene una única posible acción.

Los tipos detallados a continuación son en realidad variantes relativamente sencillas de esta máquina, añadiendo matices que las hacen diferentes. De esta forma, se producen cambios (algunos sutiles) en los comportamientos que se deducen de ellas.

- **Máquina Jerárquica**

Esta variante de la FSM clásica ofrece unas posibilidades de gestión interesantes (entendiendo esto como capacidad de control del diseñador/programador sobre la capa de IA) cuando, por ejemplo, uno de los estados definidos inicialmente es demasiado complejo para ser autónomo en su gestión.

Si un enemigo requiere de una IA muy sofisticada y por tanto es necesario que conozca muchos estilos de combate, podría variar su estilo de ataque si en el control de su estado ATTACKING determina la lógica necesaria para ello. Eso sí, el uso de un solo estado para este propósito podría limitar el carácter modular y manejable de la FSM. Quizá lo más interesante sería utilizar otra submáquina de estados que se encargara del control de ATTACKING sobre el control general, es decir, dotar de autonomía propia a uno de los estados principales pero para ser el mánager de otra FSM, entendiendo estado mánager como nodo que activa la máquina subordinada, porque es en realidad la submáquina la que controla el comportamiento y no dicho "superestado".

Así pues, el estado ATTACKING, podría contener una submáquina con subestados como PHYSICAL, si se trata del combate físico, o MAGICAL, si desempeña el rol de un mago, por ejemplo.

Esta técnica/metodología de diseño de FSM resulta una opción más que interesante para añadir complejidad a un sistema FSM básico; se mantiene la naturaleza modular y simplista del sistema inicial, proporcionando otra máquina en un nivel inferior que se activará únicamente cuando el "superestado" gestor/mánager pase a ser el estado actual.

Cuando la máquina entre en estado ocioso (llegue algún evento que no conoce, por ejemplo), se retornará el control a la máquina inmediatamente superior.

Otro punto de vista posible para el modelado de este tipo de sistemas es plantear la submáquina como una serie de subestados que pertenecen a un superestado, no como una FSM al uso. Es mínimamente diferente y en la práctica el funcionamiento es similar. Acorde a las exigencias del diseñador, se puede tomar un enfoque u otro. En secciones posteriores se explicará el carácter jerárquico de las máquinas de estados relativas a la API que se ha

creado, las cuales están basadas en el primer enfoque, que otorga un carácter independiente a cada máquina, encapsulando la lógica/problemática de cada una de ellas.

A continuación se muestra una ilustración que clarifica ambos enfoques:

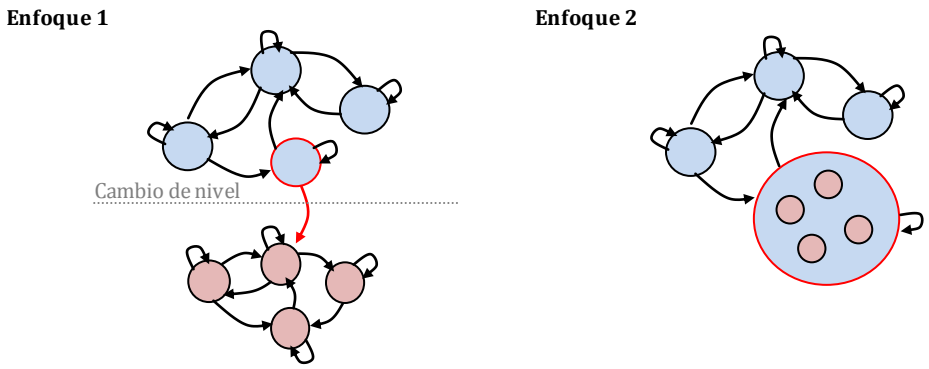


Ilustración 2. Enfoques de la máquina jerárquica

En el primer enfoque, hay dos máquinas independientes y el estado que activa la submáquina tiene únicamente un puntero a ella. En el segundo, el superestado tiene incrustada la submáquina (o subestados) requiriendo un mayor conocimiento entre objetos, lo cual (bajo mi perspectiva) no es deseable.

Para más información al respecto, consultar el libro [3], capítulo 15, página 268.

- **Máquina Probabilista**

Este es un tipo de FSM indeterminista que otorga un carácter menos previsible a la entidad a la que está sujeta; forma parte de las FSM *Fuzzy* o borrosas. Si la máquina es lo suficientemente compleja, el personaje dispondrá de un comportamiento impredecible aunque éste esté basado en la percepción-reacción gracias a una activación de sus transiciones que hace uso de la probabilidad.

Un personaje gobernado por este tipo de máquina, podrá tener comportamientos diferentes hacia una misma situación. Es decir, si por ejemplo tenemos un personaje policía que detecta en la partida a un ladrón, es posible definir un comportamiento que determine que en el 50% de las veces que detecte a un enemigo intente detenerlo (transite al estado Reducir enemigo) y el otro 50% vaya a buscar refuerzos (transite al estado Dar alerta). Esta solución es quizá demasiado simple y puede diseñarse de manera más sofisticada, pero incluso con este diseño, el jugador no podrá predecir la reacción del personaje.

Es una variante muy interesante pues precisamente es esta indeterminación la que caracteriza un comportamiento realista. Con una máquina de estados determinista, el jugador puede llegar a darse cuenta del comportamiento que tiene un NPC y sentirse aburrido tras varias partidas, dependiendo del caso.

- **Máquina basada en pilas**

Es otra variante interesante de las FSM clásicas que permiten otorgar memoria a, por ejemplo, un enemigo. Básicamente, consiste en el apilado de un estado "inacabado" en un sistema de pilas cuando se dispara la activación de otro que la interrumpe (y así sucesivamente si fuera necesario).

De esta manera, cuando un soldado está en un estado de exploración en una zona concreta y se ve sorprendido por un tiroteo, correría a cubrirse y/o a atacar (este nuevo estado de cobertura/ataque interrumpiría al de patrulla), en caso de salir ileso de éste podría volver a su tarea inicial (desapilado del estado cobertura/ataque) con la posibilidad de recuperar la posición que tenía en ese momento previo al tiroteo (si almacenamos el estado del estado, valga la redundancia) o bien alcanzar otro estado en caso de estar herido y apilar el de ataques:

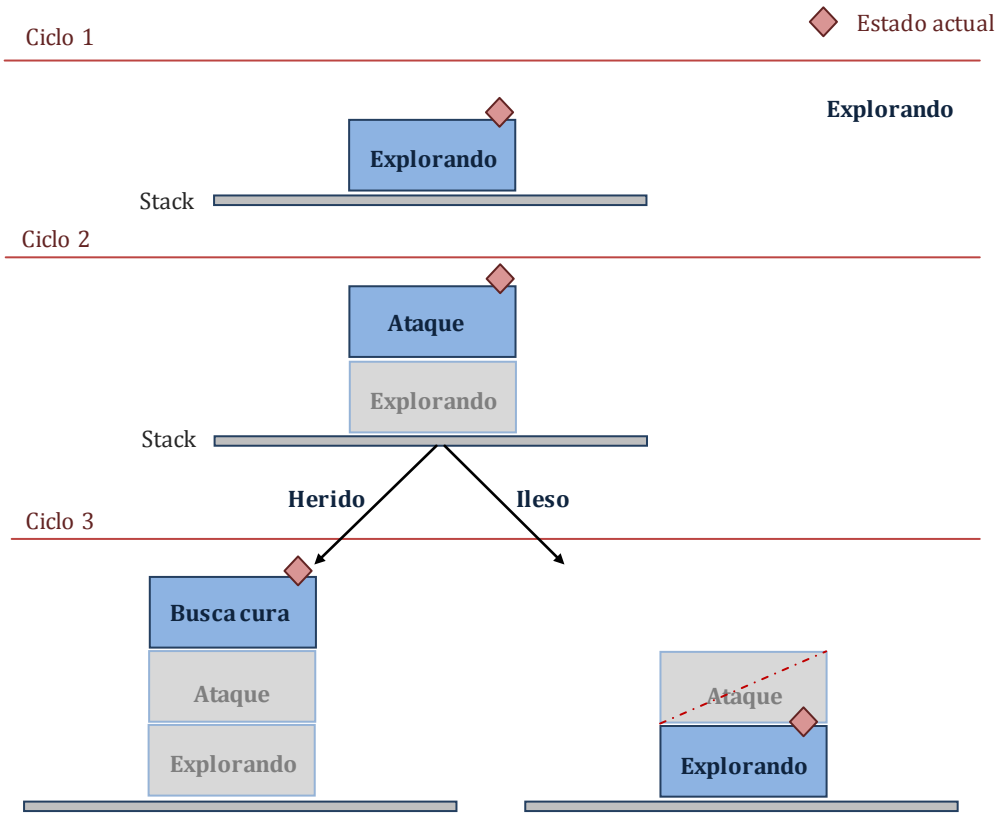


Ilustración 3. Funcionamiento de la máquina basada en pilas

Internamente, el estado que interrumpe al primero se eliminaría de la pila y se trataría el inmediatamente anterior. Hay que tener cuidado, eso sí, con dicho desapilado de estados, pues quizá la capa de control de IA deba decidir si volver al estado anterior es lógico o si por lo contrario, debido a las nuevas acciones, es necesario realizar otra tarea que se adecue más (de ahí la división de casos en la ilustración anterior).

- **Máquina Inercial**

Este tipo de extensión de una FSM surge a raíz de un problema bastante frecuente en esta técnica, conocido como la oscilación entre estados.

Si las condiciones que disparan el cambio de estado son demasiado simples (para un juego que exija algo más) podrían darse cambios constantes entre estados provocando un efecto nervioso en el personaje que domina la FSM y por lo tanto un comportamiento emergente incorrecto que no ha sido programado explícitamente.

Un ejemplo muy sencillo se podría dar en un *shooter*, cuando un centinela descubre al protagonista. Imaginemos que la FSM que impera sobre el centinela dispara el estado "ataque" al detectar al protagonista, pero que debido a unas condiciones de disparo quizá algo simples, dispara el estado "descansar" cuando deja de verlo. Esto provocaría un cambio continuo de estado en el centinela si el jugador decidiera mostrarse y esconderse, y se observaría el efecto nervioso de "apuntar-descansar-apuntar-descansar…" en el enemigo.

Una manera de solucionar este efecto extraño y poco inteligente es introducir inercia en la FSM, es decir, disminuir esa tendencia a cambiar de estado de manera instantánea cuando se produce un cambio.

Esto no quiere decir que no se vaya a ejecutar el estado correcto, sino que se intenta retardar su ejecución para disminuir este nerviosismo. Son varias las alternativas que se pueden manejar: dar un peso especial a determinados estados anteriores al nuevo (de esta manera el centinela no dejaría de intentar atacar, aunque sería interesante incluir un estado de búsqueda del enemigo, en este caso); utilizando un timeout (obligando al centinela a estar en un estado concreto pese al disparo de otro estado)… en definitiva, dando más prioridad a la inercia/tendencia inicial que al disparo de un estado para regular la reactividad del NPC.

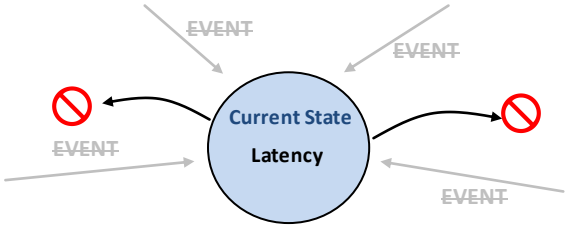


Ilustración 4. Funcionamiento de la máquina inercial

- **Máquina Concurrente**

Esta FSM funciona muy bien para determinadas situaciones o aspectos en los que es necesario realizar más de una acción cada vez (o al menos dar esa sensación de concurrencia).

Podríamos definir una máquina de estados para el comportamiento facial de un personaje, es decir, en este caso no hay una única FSM para una entidad, sino que decidimos controlar por partes a un personaje. Habría varios estados como Mover boca, Pestañear, Mover Nariz, etc… y los estados de descanso relativos a éstos, como Parar boca, Fijar ojos, Parar nariz, etc…

Dado el carácter concurrente de la máquina, podríamos tener varios estados activos siempre que no fueran excluyentes entre sí (esto es fácilmente controlable en el diseño de la máquina), de forma que un personaje pudiera mover la boca y a la vez pestañear. Sería absurdo que un personaje activara los estados Mover boca y Parar boca a la vez.

- **Otras**

  - **Basada en mensajes o eventos**

    Es posible que una FSM contenga transiciones entre estados que se invocan con poca frecuencia, que la ejecución de la lógica que determina estas transiciones resulte computacionalmente costosa y/o que se usa un sistema de polling para determinar las transiciones a llevar a cabo.

    En estos casos es interesante cambiar el punto de vista y sortear este coste computacional con el disparo de eventos o mensajes.

    Para ello es necesario que los estados de la FSM sean potencialmente activables tanto por una transición (como hasta ahora) como por mensajes o eventos. Es necesario, por tanto, un cambio en el diseño clásico de las FSM para que soporten esta opción.

De esta manera, la FSM ya no tiene por qué seguir un flujo de programa lineal, y puede activar un estado (a priori inalcanzable) desde la posición actual en el grafo.

Finalmente no se ha añadido porque requería bastantes cambios en el diseño de la API inicial, y su utilidad, bajo mi punto de vista, es discutible. Si bien es verdad que permite romper con el funcionamiento normal y ofrecer alternativas interesantes (que también llevan a confusión), su principal objetivo es evitar el *polling*, y explícitamente lo consigue, pero sigue apareciendo internamente.

- **Fuzzy FSM (otras)**

  - *Basadas en modelos de Markov*

    Con transiciones completamente probabilistas. Útiles cuando se necesita definir una toma de riesgos o realizar una tendencia de comportamiento fallida.

    Las entidades presentan un comportamiento indeterminista que puede modelarse definiendo una probabilidad con una función de confianza.

    Finalmente no añadida, porque salvo error u omisión, lo que propone es modelable con la máquina probabilista.

  - *Basadas en logica difusa real*

    Son máquinas basadas en reglas determinadas previamente y combinables. Acorde con lo visto en [8], esta metodología necesita de verdades parciales o no absolutas que son complicadas de establecer en una aplicación genérica.

    Su especificación es confusa y no se ha encontrado bibliografía suficiente para hacer frente a su adaptación a las FSM.

  - *Basadas en transiciones priorizadas*

    Se basan en máquinas de estado simples, aunque requieren un estudio de prioridad en cada ciclo de ejecución para determinar las acciones a realizar, ejecutando la lógica del estado con mayor nivel de activación en dicho ciclo.

Considero que el dominio de una FSM con estas características puede hacerse demasiado complejo, además de que la FSM basada en pilas con carácter probabilista podría modelar comportamientos similares. Por todo ello, no se ha añadido finalmente.

2.3 Crítica al estado del arte

La situación actual merece ser replanteada. La mayoría de los proyectos que se llevan a término, como se ha comentado, hacen uso de la técnica expuesta en secciones anteriores, o al menos, apuestan por mecanismos de control que claramente utilizan las bases expuestas por las máquinas de estados finitos.

Las situaciones que un proyecto de desarrollo, como es el caso de un juego, en el que se exponen una serie de actores, todos ellos presentes en un escenario variable y en el que por tanto se dan cambios continuamente que les afectan en su manera de actuar y comportarse, son premisas que invitan a la utilización de estructuras que controlen en todo momento sus estados, siendo éstos meras reacciones a sucesos dentro de un mundo vivo.

Ahora bien, como anticipaba en la motivación del presente trabajo, muchos de esos mecanismos empleados en la actualidad son simples adaptaciones de las ideas que componen la técnica de las máquinas de estados finitos, siendo éstas estructuralmente insuficientes además de ser poco reutilizables.

Por otro lado, la gran mayoría de proyectos hace uso del aspecto determinista de la técnica basada en máquinas de estados, pero no tiene en cuenta las posibles extensiones de este paradigma básico y aparentemente tampoco existe una plataforma de control que pueda ser utilizada por las desarrolladoras para facilitar la utilización del sistema de manera eficiente y aprovechable.

La aparente poca sofisticación como técnica de Inteligencia Artificial parece haber hecho mella en los desarrolladores, que quizá piensen en mecanismos de utilización *ad-hoc,* que cumplen con los requisitos que exige el momento, y no en llevar a cabo herramientas que aporten facilidades para el uso de autómatas de control, cuando en realidad es este tipo de tecnología la que demuestra ofrecer un nivel de dominio y dirección sobre la capa relativa a la Inteligencia Artificial que pocas (o ningunas) metodologías y procedimientos otorgan.

Veo por lo tanto necesaria la creación de una herramienta genérica basada en el paradigma de las máquinas de estados finitos deterministas, además de extender dicho paradigma permitiendo la inclusión de nuevas máquinas, conformando un motor que rompa con la dinámica de creación y utilización *ad-hoc* mencionada de la técnica.

2.4 Propuesta de mejora

Teniendo en cuenta lo expuesto como crítica al estado del arte, en la que se expone la importancia del paradigma de las máquinas de estados finitos en el mundo del desarrollo de software y por extensión en los videojuegos, en donde es muy lógico y útil tener cierto control sobre los eventos que se suceden en un entorno cambiante, realizo una propuesta de mejora que tenga en cuenta estas premisas.

Considero necesaria la utilización de una herramienta de control, que base su funcionamiento en el paradigma de la técnica expuesta y que por tanto oferte las opciones idóneas para su uso.

Por otro lado, esta herramienta sería de uso público; desconozco de ninguna API gratuita que esté basada en las máquinas de estados o sea un banco de herramientas que conformen un motor de Inteligencia Artificial en el que se exponga la técnica con extensiones.

Algunos de los motores crecientes en uso, como es Unity3D [10] proponen el uso de máquinas de estados para el control de animaciones [17], pero no hay mecanismos de control de utilización para la capa de Inteligencia Artificial.

En resumen, se propone una herramienta de uso como la siguiente:

- Que su núcleo esté fundamentado en el paradigma de las máquinas de estados finitos.

- Que utilice las extensiones del paradigma clásico determinista encontradas en la bibliografía.

- Que sea integrable en algún motor de uso creciente, con la idea de fomentar su utilización por parte de las desarrolladoras independientes.

- Que utilice un lenguaje de programación estándar o de uso común, al menos, en el ámbito de los videojuegos.

- Que sea sencillo y dote al usuario de mecanismos de inserción rápida de máquinas, de forma que le sea posible añadir comportamientos modelables mediante dicha técnica.

CAPÍTULO 3. Análisis

En las secciones siguientes se ofrece la fase de análisis del problema y el diseño finalmente elegido. Se detalla posteriormente la especificación de la API creada que se basa en el paradigma de las máquinas de estados finitos como técnica de IA.

3.1 Análisis del problema

En lo que respecta a la herramienta, se ha realizado un estudio previo para afrontar con garantías la fase de diseño. Los siguientes puntos detallan la fase de análisis.

3.1.1 Propuestas y decisiones para el diseño

A continuación se muestran el análisis realizado en forma de propuestas y decisiones que fueron cambiando en base a lo necesitado.

Primeras propuestas

- La idea principal ha sido otorgar al usuario la posibilidad de **crear sus máquinas de estados finitos y cargarlas de manera externa**.

  Para ello se ha implementado una utilidad adicional a la aplicación básica que es capaz de leer un documento xml estructurado siguiendo una plantilla concreta y volcar su información a la estructura de datos propuesta en la API.

  Se estudió la posibilidad de realizar un parseo al vuelo de la información propuesta por el usuario pero no se conocía con claridad el proceso a seguir para ello, así que se decidió volcar dicha información a otra estructura intermedia que la sintetizara como un árbol de nodos *xmlDocument*, que hiciera posible el acceso a cada uno de estos nodos cómodamente para entonces ir rellenando progresivamente la estructura de datos de la API.

  Aunque hay un proceso intermediario de nodos *xmlDocument*, la aplicación no se resiente computacionalmente en exceso, de forma que ha sido posible tomar la decisión de llevarlo a término por esta vía.

- **Se ha decidido añadir las máquinas de estados finitos encontradas y propuestas en la bibliografía que más utilidad práctica tenían.** Aunque se mencionan otras máquinas, su desarrollo no se ha contemplado por falta de tiempo y/o de más información para su implementación.

  Las máquinas elegidas fueron las siguientes:

- Máquina determinista
- Máquina indeterminista (al menos una de las propuestas)
    - Probabilista
    - Basadas en cadenas de Markov
    - Transiciones priorizadas
    - Máquinas *Fuzzy*
- Máquina jerárquica
- Máquina Inercial
- Máquina basada en Pilas
- Máquina Concurrente

- **Los requisitos que exigían dichas máquinas debían completarse creando una serie de entidades básicas** que les dieran forma y capacidad de almacenamiento.

  Como en un autómata finito clásico, se ha propuesto crear **estados**, **transiciones** y **acciones**. Todas estas entidades, son adheridas a una estructura de datos que las almacena y gestiona, que adquieren el nombre del tipo de la máquina tal que así: FSM_"TipoFSM". En principio:

    - Los estados almacenan transiciones y contienen también acciones a realizar (estas acciones pueden constituir la lógica directamente de la tarea a realizar o una referencia a ella).

    - Las transiciones parten de un estado y alcanzan otro estado. También podrían tener alguna acción asociada.

    - Las acciones podrían integrarse como código ejecutable o script en el diseño xml mencionado de las máquinas, de forma que se pudieran añadir por diseño y no por programa. Estas acciones constituirían las tareas finales que llevarían a cabo los personajes de un futuro juego o procesos de cualquier otro proyecto software.

- **La interacción Entidad – Máquina de Estados se realizaría en las propias estructuras de datos propuestas como FSM_Tipo**, mencionadas previamente, de forma que fuera posible almacenar un estado actual y cambiar de estado en base a percepciones del entorno. Basando estas ideas en lo propuesto en [9] como ejemplo de implementación.

- **Las entidades sensibles a utilizar una FSM realizarían el recorrido de su máquina, realizando las invocaciones oportunas de gestión y actualización.** Por ejemplo, el personaje_1 podría tener asociada una copia de la FSM_Tipo1**.**

Crítica

Las propuestas anteriores fueron duramente criticadas y se estudiaron los pros y contras que constituían. Se hizo una criba importante de lo propuesto con el objetivo de refinar el proyecto.

- Como se ha comentado, no se lleva finalmente a cabo un parseo al vuelo de la información extraída de los xml que constituyen el diseño de cada máquina de estados. En este sentido, **se mantuvo la propuesta inicial de hacer uso de una estructura intermedia xmlDocument**, pues no constituía ningún impedimento a nivel computacional.

- En lo referente a las máquinas propuestas al inicio, sí que **se decidieron una serie de cambios**:

    - **La máquina determinista sería la base** de las demás máquinas de estados, de manera que éstas fueran extensiones del paradigma clásico (siendo en realidad nuevas máquinas con pequeños matices que las distinguirían).

    - **La máquina indeterminista elegida podría ser la basada en probabilidades de ejecución**, pues con ella ya se alcanzaba el indeterminismo perseguido, además de que modelaría casi todos los casos que requirieran de dicho indeterminismo. Aunque no se descartaba realizar alguna propuesta más, como las basadas en Markov.

    - La máquina de pilas tendría la **capacidad de almacenar las máquinas y estados anteriores**. De forma que fuera perfectamente posible regresar al estado anterior aunque éste formara parte de otra máquina. Además, los estados manejarían una prioridad que determinaría su importancia en la máquina.

      Este hecho permitía dar la opción de que un estado pudiera interrumpir a otro o no, dependiendo de dicha importancia otorgada al inicio.

    - La máquina jerárquica dejaría de ser una máquina como tal y **pasaría a ser una característica más** de la máquina determinista base, de forma que cualquier estado pudiera tener una referencia a otra máquina de nivel inferior.

- Debido a que la concurrencia de máquinas ya era posible asignando a cada entidad sensible a las FSM (un personaje no jugable, por ejemplo) más de una máquina, se descartó en un principio su implementación.

- El principal cambio en las entidades que conformarían cada máquina de estados lo constituiría la concepción de **las acciones desde otro punto de vista**. La lógica de éstas ya no se llevaría a cabo externamente sino en el propio programa (más concretamente en el NPC que tuviera asignada una máquina). Las acciones definidas externamente serían meras referencias a las tareas reales a desempeñar.

  Este fue un cambio importante debido a que **de esta forma se separaba la lógica de las acciones del diseño de la máquina**.

- Por otro lado, **aparecería una nueva entidad llamada FSM_Event**. Las transiciones ahora serían sensibles a eventos que sucedieran en el entorno. Estos eventos serían meras referencias o números de entrada a cada máquina, de forma que éstas devolvieran a su vez otras referencias (las relativas a las acciones).

  Un NPC ahora generaría eventos, y la máquina le devolvería las órdenes necesarias para actuar.

  Para evitar confusiones entre la entidad FSM_Event y los eventos que se dan en la ejecución del programa, **es necesario leer la especificación final descrita en secciones posteriores**.

- **Se desestimó la opción de incrustar la lógica de recorrido en la propia máquina, lo cual constituía un cambio vital en la interacción entidad-FSM**. Esto traía un problema bastante importante no contemplado en un principio: recorrer la estructura en la propia estructura suponía garantizar un único recorrido para todas las entidades adheridas a la misma FSM, obligando a crear otras diferentes.

  Esto, obviamente, fue un **error importante**: no tiene sentido, por ejemplo, que un número determinado de NPCs enemigos hagan exactamente los mismos movimientos en cada ciclo, pues al compartir máquina, comparten comportamiento.

  Por lo tanto, lo visto en [9] se revisaron y adaptaron a la nueva problemática: la generación de llamadas *setState* de esta demo, que son en realidad

disparos de cambios de estados dada una condición, se adecuó a la generación de eventos.

- **También la primera propuesta de que gran parte de la lógica de recorrido se realizara en las propias entidades (NPCs) constituyó otro error importante**. La API perdería fuerza y utilidad teniendo en cuenta que el trabajo consistía en la creación de una herramienta genérica, la cual de esta forma no garantizaba ni independencia ni sentido final.

  Así pues, se decidió separar la estructura de datos de la lógica del juego y del recorrido, llevándolo a cabo descansando sobre otra entidad (*FSM_Machine*) que finalmente constituiría el núcleo de la API junto al mánager de IA (*FSM_Manager*), y que sí garantizaría autonomía propia en la interacción con las FSM.

- Dada la aparición del empleo de muchas referencias (eventos y acciones básicamente) **se hizo necesaria la utilización de una clase estática Tags** que hiciera definir explícitamente al usuario las etiquetas y *tokens* (etiqueta-valor) de cada entidad de una FSM.

  Esto requiere una mayor implicación del diseñador, pues debe especificar dichas etiquetas como un valor constante, de forma que el programador las tenga disponibles para referenciarlas si fuera necesario. Esta clase estática se utilizaría a su vez en la utilidad de parsing para etiquetar muy rápidamente cada entidad.

Propuesta final

La propuesta final fue en realidad una fase más de refinamiento a la crítica anterior.

A continuación se exponen las últimas consideraciones que formarían el diseño final (especificado en secciones posteriores).

- Se tomó la decisión de **aunar en una misma máquina a la máquina Determinista y a la máquina Indeterminista**. Ambas constituirían la Máquina Clásica, que podría activar el indeterminismo o no.

  Esto, además de ser un cambio importante en el diseño de máquinas, también **constituía un avance más que interesante en lo práctico**: posibilitaba la inserción de máquinas híbridas ya que las demás extensiones de la máquina determinista pasaban a ser también extensiones de la indeterminista. **Las posibilidades se multiplicaron**.

- **El carácter indeterminista de la máquina clásica también cambió**. En un primer momento se utilizaron probabilidades de ejecución en las transiciones distribuidas de forma que todas las salidas de un estado constituyeran el 100% de la probabilidad. Esto se desestimó, pues traía consigo una minimización notable de la probabilidad de activación si de un estado partían muchas transiciones activables por diferentes eventos.

  Finalmente se propuso desestimar la normalización de probabilidad y el usuario definiría probabilidades de ejecución de cada transición de manera independiente (tanto entre ellas, como con los eventos que llegaran a la máquina).

  Estos matices solamente fueron detectables tras numerosas pruebas de la herramienta, por lo que vinieron casi al final de la etapa de implementación.

  Para más detalles, **es necesario leer la especificación de secciones posteriores relativas a la máquina clásica-probabilista.**

- La jerarquía de máquinas estuvo en continuo cambio. Los problemas obtenidos para acceder a determinas ramas en la jerarquía hicieron que se replanteara el diseño de esta característica.

  El diseño previo de la jerarquía implicaba que una máquina superior incluyera los eventos que activaban la inferior, dando pie a una mezcla de autómatas y por tanto a un diseño farragoso y poco transparente.

  La primera idea para mejorar esto fue la inclusión de un FSM_Event jerárquico, además de los eventos ya contemplados.

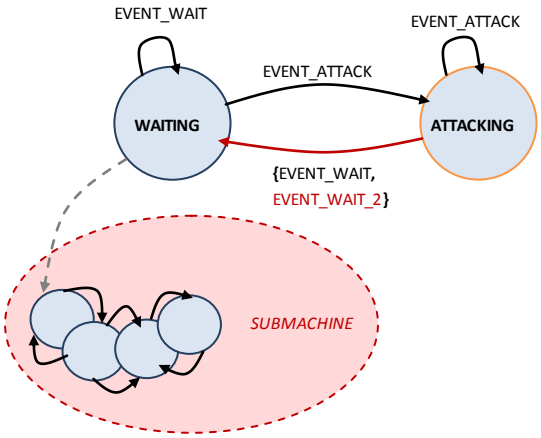
  - **Evento de tipo jerárquico** (*HIERARCHICAL*)

    En un primer momento, se contemplaba este otro tercer tipo que permitía tratar la jerarquía de las FSM. Este tipo es el jerárquico, pensado para cualquier FSM y aplicable a una transición cuyo estado destino tuviera una submáquina asociada. De no ser así, un evento jerárquico hubiera funcionado como un evento básico para la FSM.

    Al principio, cuando la especificación de una FSM incluía una submáquina asociada a un estado era necesario que al menos una de las transiciones que llegaran a dicho estado fuera sensible a un evento de tipo jerárquico. De no ser así, nunca se invocaría a la submáquina.

Esto forzaba a que una máquina conociera los eventos de las submáquinas que tuviera asociadas, de forma que no fueran completamente independientes. Por este hecho, se ha desestimado esta forma de trabajo.

Un ejemplo gráfico de su uso:



Si el estado **ATTACKING** es el estado activo y se da el evento **EVENT_WAIT_2** de tipo ***Hierarchical***, la lógica a seguir por la FSM es la de apilar la máquina actual y sustituir su ejecución por la de la submáquina asociada al estado destino WAITING. Cuando la *subFSM* se encuentre en estado ocioso, devolverá el control a la máquina superior, que se desapilará y seguirá con su gestión tal y como estaba antes de invocar a la submáquina.

Ilustración 5. Uso de eventos de jerarquía (desechado)

Como se deduce en la ilustración, el EVENT_WAIT_2 es un evento que afecta únicamente a la submáquina pero que la máquina superior también debe conocer.

Como se ha comentado anteriormente, **este tipo de evento no está incluido finalmente**, pues aunque exista la anidación de FSM, se considera necesario que en su diseño sean completamente independientes.

Desechando la opción anterior, finalmente **se estudió la idea de incluir por diseño a cada máquina una referencia a un procedimiento de eventos al que fuera sensible**, de forma que éste generara los eventos pertinentes a una máquina. De esta manera se separaba la lógica de cada máquina en la jerarquía y se esclarecía el diseño.

- Los FSM_Event finalmente integrados fueron uno básico (*Basic*) y uno de tipo apilable (*Stackable*), siendo el uso de este último utilizable en la máquina de pilas únicamente, y con el fin de facilitar el apilado de estados.

Una transición que fuera sensible a este tipo de evento permitiría otorgar facilidades para apilar el estado del que partía la transición y además aportaría luz al diseño de la máquina de pilas, el cual era y es un poco más complicado.

- Para evitar confusiones, **se decidió separar conceptualmente lo que significaba una máquina de estados (diseñadas vía xml por el usuario) de los autómatas que constituirían estas máquinas**.
  Así pues, los tipos FSM_"NombreTipo" pasarían a ser FA_"NombreTipo". Y la máquina de estados real a recorrer sería la *FSM_Machine* basada en uno de los FA.

- **El FSM_Manager mencionado brevemente en la crítica, constituiría el motor principal que generaría objetos FSM_Machine basados en los tipos de autómata.**

- **La carga de máquinas también sufrió cambios al final.** En un primer momento se realizaba el parseado de todas las máquinas diseñadas (me refiero a los xml) disponibles desde dentro de la herramienta: existía un bucle que recorría los directorios y cargaba todo lo que hubiera disponible.

  Esto se extrajo de la herramienta y pasó a formar parte de la lógica del programa, de forma que se minimizó y se simplificó: el objeto *FSM_Manager* haría uso de este parser de información y la carga la especificaría el usuario desde el programa cargando las máquinas individualmente.

- **Se llevó a cabo la inclusión del autómata relativo a los estados concurrentes y para solucionar el problema de la concurrencia** que se daba a la hora de decidir si ejecutar o no un estado, **se basó el diseño en los créditos de ejecución vistos en las Redes de Petri**, explicadas en la especificación.

3.2 Análisis de las herramientas

En paralelo al análisis del problema en el que los requisitos mínimos de la herramienta se expusieron y discutieron, se realizó un estudio de qué herramientas eran las más adecuadas para desempeñar esta tarea .

3.2.1 Introducción

El presente proyecto ha sido extraído de otro proyecto que se pensaba llevar a término inicialmente, y por tanto, algunas de las decisiones que afectan al actual trabajo se discutieron con anterioridad. Considero importante hablar del proyecto previo para contextualizar el desarrollo del nuevo.

Un proyecto conjunto

Se tomaron decisiones previas al análisis de la especificación de esta API que han afectado al apartado técnico en la que se basa la misma.

En un primer momento, el presente trabajo iba a formar parte de un proyecto mayor formado por tres personas: Steve Rossius, Javier López Punzano y yo.

El primer proyecto ideado en conjunto consistía en el desarrollo de un videojuego para dispositivos móviles que hacía uso de la Realidad Aumentada [19].

Para ello, se decidió emplear *Unity3D* por ser un motor de videojuegos que está en continuo crecimiento para la comunidad de desarrollo independiente, la cual está llevando a cabo proyectos software ambiciosos y de calidad notable.

Además, se haría uso de una plataforma basada en la Realidad Aumentada llamada *Vuforia* [15], integrable en el motor mencionado, y que proporciona resultados precisos en la interacción con dicha tecnología.

El proyecto debía contar con módulos claramente diferenciables para que cada integrante del grupo pudiera llevar a cabo una defensa de su labor de manera independiente.

Finalmente, la envergadura de un videojuego como el que habíamos ideado, sumada al trabajo de los tres módulos a desempeñar, constituía un proyecto demasiado grande teniendo en cuenta el tiempo con el que se contaba, de forma que se acabó posponiendo la idea de llevar a término dicho juego, incluso en versiones beta, para que cada integrante se centrara en su módulo.

3.2.2 Herramientas para el desarrollo

El abanico de posibilidades para llevar a cabo el desarrollo es bastante amplio. Ahora bien, la idea de integrar la herramienta como utilidad para uno o varios de los motores en auge implicaba estudiar una vía de desarrollo estándar.

Son muchos los motores existentes de videojuegos en la actualidad [18] y la mayoría de ellos son de carácter gratuito siempre que la idea del usuario no sea publicar su producto.

Era necesario tener en cuenta también el lenguaje de programación a emplear, pues se requería un lenguaje presente en muchos proyectos de desarrollo, además de constituir una aplicabilidad multiplataforma.

Llegado este punto, había otra premisa a tener en cuenta. Aunque la motivación principal del trabajo era realizar un aporte significativo al mundo de los videojuegos, sería interesante que esta metodología se extendiera también a otros

trabajos de desarrollo general, que no necesariamente tuvieran que ver con el mundo de los videojuegos.

La cantidad de opciones ya se reducía considerablemente, y teniendo en cuenta la idea de retomar el proyecto inicial de desarrollo de un videojuego en un futuro, parecía evidente la utilización de ***Unity3D***.

Otro hecho que justificaba su uso, era la experiencia obtenida en el posgrado con el empleo de este motor. Ya eran dos trabajos los llevados a término utilizando este *engine* de videojuegos, por lo que la familiarización con esta herramienta estaba muy avanzada. El primer trabajo consistió en la creación de un render para una plataforma de agentes, lo que me permitió adquirir conocimientos sobre el entorno, el uso de escenarios y codificación, y el segundo trabajo fue un tutorial de utilización de la herramienta basado en un juego pequeño, lo cual me sirvió para conocer trucos y posibilidades que ofrecía *Unity*, así como su capacidad para exportar a múltiples plataformas y la integración de otros motores de desarrollo desde el motor como *Visual Studio*, aparte del integrado por defecto, *MonoDevelop*.

A nivel personal, también había aprendido a crear escenarios, y a emplear el motor de máquinas de estados finitos para la animación de *sprites*.

Ilustración 6. Sprites en Unity

Unity permitía el desarrollo en lenguaje *C#*, *Javascript* o *Boo*, y es más, la combinación de todos ellos, además de poseer la capacidad para exportar en múltiples plataformas como *Windows*, *OS X*, *Linux*, *Xbox 360*, *PlayStation 3*, *Wii*, *WiiU*, *iPad*, *iPhone* y *Android*, y por otro lado, gracias al *Plug-In Web* de *Unity*, también tenía la capacidad para el desarrollo de juegos en el navegador, para *Windows* y *Mac*.

Todas estas ventajas, además de utilizar un lenguaje estándar y ampliamente conocido y utilizado en el mundo del desarrollo de aplicaciones, como es el caso de C#, hicieron que personalmente lo planteara como una opción seria.

Otro motor interesante para llevar a cabo el proyecto podría haber sido ***Wave Engine*** [20], igualmente con la capacidad de integrar *Visual Studio*, de exportar a múltiples plataformas y Sistemas Operativos, y todo ello empleando el lenguaje C#.

Se descartó desde un inicio por la inexperiencia con el motor, pues el aprendizaje y familiarización con la interfaz podría haber llevado más tiempo del previsto.

Otro de los motores demandados y utilizados por la comunidad de desarrolladores, como es el caso de ***ShiVa3D*** [16] fue descartado por no cumplir el requisito de uso del lenguaje C#.

3.2.3 Detalles finales

Finalmente, teniendo en cuenta ambos apartados de análisis presentados anteriormente, se ofrecen los detalles técnicos relativos a las herramientas y una API de gestión de IA formada por un conjunto de máquinas de estados que podrán utilizarse independientemente o anidarse en jerarquía, que podrán ser deterministas o probabilistas como base, y luego además Inerciales, basadas en pilas o de estados concurrentes.

Detalles técnicos del proyecto

- *Desarrollo sobre el motor de videojuegos Unity3D v4.3.4f, aunque no exclusivo ni basado en la herramienta.*
- *Entorno de desarrollo MonoDevelop v4.0.1 (Integrado en Unity)*
- *Lenguaje de programación C#.*
- *Uso de lenguaje etiquetado XML para el diseño de FSM desde fichero.*
- *Demostraciones simples creadas con el motor mencionado previamente.*
    - *Uso de primitivas básicas (esferas, planos, etc..)*
    - *Control de un personaje como entrada de teclado.*
    - *Iluminación y uso de texturas básicas*

CAPÍTULO 4. Diseño e Implementación de la API

En los siguientes apartados, se explica al detalle la especificación de la API, el flujo de datos y funcionamiento interno y demás componentes que le dan forma.

4.1 Diseño de los componentes de la API

En esta sección se amplía la información sobre cada uno de los módulos que componen la API, diferenciando entre la estructura de datos que almacena los autómatas relativos a cada FSM y el núcleo que garantiza el recorrido.

4.1.1 Estructura de datos

Esta estructura de datos está compuesta por diferentes objetos: Estados, Transiciones, Eventos, Acciones y los propios autómatas que recogen los objetos anteriores y dan forma a cada máquina. Por otro lado, también hay que tener en cuenta la clase de constantes y métodos estáticos llamada Tags, definida (de manera opcional) por el usuario de la API y útil para tener localizados e identificados por medio de etiquetas con su correspondiente *token* todos los eventos/acciones/estados/transiciones añadidos.

La aplicación genera un autómata/grafo dirigido dependiente de la fase de diseño. Los autómatas que derivan de dicha fase pueden ser de múltiples tipos, es decir, no hay distinción entre clases de autómatas diferentes cuando se realiza la carga. Así pues, de acuerdo con [11], capítulo 17, es posible generar cualquier tipo de grafo vía diseño. Eso sí, **el núcleo no garantiza el recorrido óptimo de cualquier estructura cargada**; considero necesario limitar las posibilidades para garantizar un funcionamiento estándar.

Hay que tener en cuenta una serie de directrices generales para la creación de un autómata cuyo recorrido posterior no suponga un problema añadido (ver la especificación de cada máquina).

En un primer momento se consultó diferente literatura para la construcción de la estructura de datos. Aunque la implementación sigue el esqueleto clásico, la implementación final no tuvo en cuenta ningún guión de desarrollo encontrado en la bibliografía.

A continuación se explica cada módulo relativo a un autómata.

- **Estados**

  Son los nodos en los que la entidad software correspondiente residirá. La clase *State.cs* es la que permite instanciar dichos nodos. Es una entidad básica para la construcción de un autómata que garantice un recorrido.

Los estados son los vértices o nodos de los grafos/autómatas. En un primer momento se contempló la posibilidad de que albergaran ellos mismos la lógica de las acciones que soportan, idea que se desechó con el objetivo de encapsular cada aspecto de la IA en sus respectivas entidades, y evitar la unión innecesaria de dos fases recomendablemente independientes, tan diferentes e importantes como el diseño y la implementación de la lógica, la cual finalmente debe ser implementada por cada entidad software que sea usuaria potencial de una FSM y no por la propia FSM vía diseño.

Los estados deben definir ciertos parámetros explícitamente para que su funcionamiento sea el adecuado. Estos parámetros pueden variar dependiendo de a qué máquina de estados finitos estemos añadiendo un estado.

Dependiendo de qué clase de máquina estemos intentando cargar, será necesario utilizar un constructor u otro para crear un estado, es decir, no es lo mismo un estado para una máquina clásica, que un estado relativo a una inercial, por ejemplo.

A continuación se explica qué es obligatorio definir dependiendo de cada situación. Un estado (Objeto *State*) debe definir, acorde con la fase de análisis:

- **Si es inicial o no**. La FSM debe partir de dicho estado.

- **Un nombre que lo identifique**. Importante en la fase de creación de la máquina de estados finitos que albergue ese estado.

- **Tres acciones**, que pueden ser:
  - Acción de entrada al estado
  - Acción de salida del estado
  - Acción de estado

  Es **obligatorio definir las tres acciones**, aunque luego no haya lógica asociada a ellas.

Por otro lado, si estamos cargando un estado que pertenece a una variante de la máquina clásica, será necesario que defina ciertos parámetros adicionales (añadidos a lo anterior).

Si un estado pertenece a un FA_Inertial también deberá definir:

- **Una latencia**, tiempo de retardo necesario para esta máquina.

Si un estado pertenece a un FA_Stack también deberá definir:

- **Una prioridad**, necesaria para la lógica de esta máquina.

Si un estado pertenece a un FA_Concurrent_States también deberá definir:

- **Unos créditos de ejecución**, necesarios para garantizar la concurrencia en la que se basa dicha máquina.

Por otro lado, si se requiere jerarquía será necesario definir un puntero a otra FSM en el propio estado. Es un parámetro obligatorio siempre que queramos definir una submáquina para gobernar un estado.

Internamente, un objeto *State* tiene un funcionamiento muy simple: alberga una lista con las transiciones que parten de él mismo, incluidos los bucles. De esta forma se conoce todo lo necesario para posteriormente residir y/o transitar a otro estado y así garantizar el recorrido por el grafo.

Se puede decir que es un objeto de base que permite el funcionamiento posterior de la aplicación y que por sí solo no tiene comportamiento alguno.

En las secciones relativas a la especificación de cada máquina, se comenta con mayor profundidad este tipo de aspectos a tener en cuenta.

Esta información se amplía en el Anexo 1.

- **Transiciones**

Son las aristas del grafo, objetos puente entre estados que los conectan. Al igual que los estados, son entidades que no tienen un comportamiento por sí solas pero que garantizan la cohesión y recorrido posterior de un grafo. Una transición es por tanto otra entidad básica para el funcionamiento de una máquina de estados finitos.

Su funcionamiento es también simple: requieren de una serie de parámetros a definir explícitamente.

Una transición necesita conocer los siguientes parámetros explícitamente:

- **Un nombre identificativo**. Es importante en la fase de creación del grafo.

- **Un estado origen**. Es necesario saber de dónde parte una transición.

- **Un estado destino**. Es necesario conocer dónde desemboca una transición.

- **Una acción de transición**. Esta acción debe especificarse obligatoriamente aunque no tenga una lógica asociada a ella finalmente.

- **Una lista de eventos que la activan**. Esta lista puede ser de un elemento o más (Mínimo uno). Es necesario conocer esta lista para garantizar el recorrido por el grafo.

Dependiendo de qué máquina estemos intentando cargar, es necesario definir algún parámetro más.

Si estamos intentando cargar una transición que pertenece a una FSM_Classic con el flag de probabilidad (*flagProbabilistic*) a 'YES' es necesario definir:

- **Una probabilidad de ejecución**. La transición se activará o no dependiendo de esta probabilidad.

Posteriormente, al igual que los estados, en la especificación de cada máquina de estados finitos se explica más a fondo cada uno de los parámetros a definir.

Esta información se amplía en el Anexo 1.

- **Eventos**

Otra de las entidades básicas y necesarias para el recorrido de cada FSM. Tal y como se ha comentado, cada máquina tiene una serie de nodos y enlaces entre ellos (Estados y Transiciones) que son, de alguna manera, sensibles a eventos puntuales que se dan en el contexto en el que se encuentran (entendemos contexto como el entorno en el que se hace uso del API, sea un juego o cualquier otro proyecto software). A partir de ahora, se asumirá que **el contexto hace referencia al entorno de cada demostración de uso de las FSM que se ha creado.**

Así pues, el contexto está formado por enemigos que hacen uso del API, pues requieren de las FSM para controlar su comportamiento, y que por tanto generan eventos (realmente son Tags de eventos) para que éstas determinen sus acciones a realizar.

Estos tags de eventos se disparan cuando se dan una serie de condiciones que especifica el programador o diseñador del software. **Un tag de evento, por tanto, es un parámetro de entrada a una FSM**.

Si se hace uso del parser aportado, es necesario definir los eventos de la siguiente forma:

```
<Transition>
<!-- Más tags -->
   <Events>
      <Event>
         <!-- Información del evento 1-->
         <ID>Nombre Identificador</ID>
```

```
            <Type>Tipo de evento </Type> <!--[BASIC/STACKABLE]-->
        </Event>
        <Event>
            <!-- Información del evento 2-->
            ...
        </Event>
         .
         .
         .
    </Events>
</Transition>
```

Tabla 3. Fragmento de ejemplo xml de eventos

Dentro de una transición hay una serie de etiquetas con información para su creación. Una de esas etiquetas es <Events>, que alberga una lista de eventos, cada uno definido con la etiqueta <Event>. Estos eventos se identifican con una cadena <ID> que puede ser cualquier nombre (Lo recomendable, por pura comodidad es que se identifiquen con la cadena **EVENT_'**NOMBRE', donde nombre variará en función del evento) y con un tipo, que puede ser de dos clases (BASIC/STACKABLE).

- **BASIC**: Evento estándar utilizado en todas las FSM.
- **STACKABLE**: Evento que apila el estado origen si se produce. Utilizado en la FSM de pilas.

Esta información se amplía en el Anexo 1.

- **Acciones**

Las FSM controlan ciertos comportamientos de las entidades a las que gobiernan. Para ello retornan acciones que son en realidad referencias a funcionalidades o tareas de dicha entidad. **Una acción, por tanto, es un parámetro de salida de una FSM**.

Cuando esta entidad (un NPC en un videojuego que requiere de inteligencia artificial, por ejemplo), genera un/unos evento/s que envía a la FSM que tiene asociada, dicha máquina realizará la gestión oportuna de la entrada que le llega y generará una serie de referencias que retornará como salida.

Como se ha comentado previamente, hay varios tipos de acciones según el diseño de una FSM. Aunque las propias acciones entre sí (como objetos) no son diferentes (son simples referencias), dependiendo de la especificación de una máquina pueden ejecutarse en un momento u otro. Es decir, **las únicas diferencias existentes entre los tipos de acciones dependen del diseño de la máquina en la que están incluidas y no de ellas mismas**.

Los cuatro tipos son los siguientes:

Tipo de acción	En el xml	Detalles
de Entrada a un Estado	*<S_inAction>*	*Se ejecuta al entrar en un estado*
de Estado	*<S_Action>*	*Se ejecuta tras entrar en el estado o tras un bucle*
de Salida de un Estado	*<S_outAction>*	*Se ejecuta al salir de un estado*
de Transición	*<T_Action>*	*Se ejecuta al atravesar una transición*

Tabla 4. Tipos de acciones

Las acciones cuyo prefijo en su etiqueta en el xml tengan una S, estarán asociadas a un Estado (Por ejemplo, *S_Action*), y las que tengan una T, a la Transición (Por ejemplo, *T_Action*). Se dan más detalles del uso de acciones en la sección de particularidades generales.

Esta información se amplía en en el Anexo 1.

- **Grafos**

Como se ha comentado, las máquinas de estados finitos definidas son en realidad grafos cuyos vértices (estados) están conectados mediante aristas dirigidas (transiciones).

El grafo relativo a una FSM se construye añadiendo estados y transiciones a un objeto FSM. Así pues, las clases relativas a cada FSM contienen necesariamente una lista de estados y se ha decidido incluir también la lista total de transiciones aunque no se requiera estrictamente. Se guarda el estado inicial (o los estados iniciales) y consultando sus transiciones se puede empezar a controlar el recorrido.

Los objetos FSM definidos son los siguientes:

- *Class FA_Classic*

  Clase relativa a un autómata clásico. Es el autómata básico del que heredan todos las demás. Su comportamiento puede ser determinista o probabilista (indeterminista). Esto permite que todas las posteriores FSM sean híbridas. Posteriormente se explicará su uso.

  Para el caso probabilista, no se normalizan las probabilidades de manera que todas las transiciones que parten de un estado sumen una probabilidad del cien por cien (eso se hacía al principio), el usuario simplemente indica un porcentaje de activación para cada transición entre 0 y 100 (siendo todas las transiciones independientes unas de otras desde el punto de vista de la máquina de estados).

- *Class FA_Inertial : FA_Classic*

  Clase relativa a una FSM inercial. Hereda de la FSM básica (Determinista) y aunque su comportamiento es simple, añade el concepto de latencia de estado: hay un periodo de permanencia en el correspondiente estado actual en el que la máquina no es sensible a cambios.

- *Class FA_Concurrent_States : FA_Classic*

  Clase relativa a una FSM de estados concurrentes. Hereda de la FSM básica (Determinista), y es una variante que elimina el concepto de "estado actual" e introduce la posibilidad de albergar un conjunto de estados activos, dando soporte para la gestión de cada uno de ellos por separado y simultáneamente. La concurrencia está basada en el funcionamiento de las redes de Petri.

- *Class FA_Stack : FA_Classic*

  Clase relativa a una FSM basada en pilas. Hereda de la FSM básica (Determinista) y es una alternativa que añade memoria a las entidades asociadas a la máquina mediante el uso de pilas que se basan en la prioridad de los estados para controlar su gestión. Su comportamiento es determinista.

Esta información se amplía en el Anexo 1.

- **Tags**

La clase *Tags* contiene la información básica sobre la API y se da la oportunidad de ampliar esta información de forma que sea posible tener localizados e identificados todos los objetos e entidades de la API; **su uso es muy recomendable**.

Como base, por defecto, la clase *Tags* cuenta con las etiquetas de los tipos de máquinas añadidas Tags.{Classic; Inertial; Stack_based; Concurrent_States), además del método estático StringToTag que permite asociar una etiqueta a una cadena de entrada, de forma que la aplicación (únicamente en la fase de carga) va asociando tags a cada elemento de la API creado que viene identificado con un string.

De forma que si creamos una FSM_Classic, tendremos que identificarla con el nombre especificado en la clase Tags de la siguiente forma:

*FSM_Classic fsm = new FSM_Classic("CLASSIC", Tags.StringToTag("CLASSIC"), *...*);*

El identificador es una cadena ("CLASSIC"), que pasada al método *StringToTag* retorna una etiqueta que a su vez tiene asociado un número o *token* definido de forma explícita en la clase *Tags* como un entero constante.

Vienen definidos por defecto los Tags siguientes como const int:

- *public const int CLASSIC = 0;*
- *public const int INERTIAL = 1;*
- *public const int CONCURRENT_STATES = 2;*
- *public const int STACK_BASED = 3;*

... y su correspondiente asociación en el método *StringToTag()* de dicha clase:

- if(word.Equals("CLASSIC")) return Tags.CLASSIC;
- if(word.Equals("INERTIAL")) return Tags.INERTIAL;
- if(word.Equals("CONCURRENT_STATES")) return Tags.CONCURRENT_STATES;
- if(word.Equals("STACK_BASED")) return Tags.STACK_BASED;

Si se añaden más Tags (tarea recomendable para estados/transiciones/eventos/acciones), será necesario añadir un *matching* cadena-tag en el método comentado.

Esta información se amplía en el Anexo 1.

4.1.2 Núcleo

Se han explicado varios objetos pertenecientes a la API que son necesarios para su uso. En esta sección se explica el corazón de ésta, el *core* que garantiza el recorrido y la asignación de FSM: las clases FSM_Manager y FSM_Machine, ambas previamente mencionadas.

- **FSM_Manager**

Esta entidad es la encargada de almacenar las FSM disponibles y realizar las tareas oportunas para asignarlas a otras entidades que requieran su uso.

La aplicación requerirá al menos uno que tendrá que ser referenciado por todos y cada uno de las entidades que sean usuarias de FSM. En el ejemplo de una partida, habrá un FSM_Manager que recibirá peticiones de los NPCs para que éste les asigne una FSM u otra, o para atender las nuevas máquinas que otra entidad esté cargándole.

Será por tanto necesario instanciar al menos un objeto FSM_Manager. Hay dos constructores posibles: el primero hace uso de la herramienta FSM_Parser aportada y el segundo es un constructor vacío.

- Opción 1: *public FSM_Manager(xmltest.FSM_Parser parser)*
- Opción 2: *public FSM_Manager()*

Como se ha anticipado en secciones anteriores, ambos constructores internamente inicializan un diccionario del tipo Dictionary<TKey, TValue> donde Tkey es una instancia de una clase que almacena pares Tipo_FSM – ID_FSM (donde tipo es el tipo de autómata e ID es una cadena que identifica a esa FSM concreta) llamada Tuple y TValue es una instancia de un autómata a almacenar (FA_Classic por defecto, aunque luego pueda ser un subtipo).

De esta manera, es muy sencillo identificar por tipo+nombre a una FSM. El FSM_Manager se encargará de realizar una búsqueda de una FSM concreta dados su tipo y su nombre.

Así pues, el FSM_Manager posee determinados métodos de adición de FSM y de asignación/búsqueda de las FSM.

Cuando una entidad necesita una FSM, invoca el método CreateMachine de FSM_Manager pasando como parámetros el objeto que requiere la FSM, un tipo y un identificador de FSM. El FSM_Manager busca en su diccionario la FA solicitada e internamente instancia un objeto FSM_Machine que se basa en el autómata almacenado.

Esto se ha hecho así para evitar que dos entidades solapen sus recorridos en un mismo autómata, es decir, en realidad estamos cargando X autómatas, que delegarán su recorrido a Y objetos FSM_Machine. **Así se garantiza que dos entidades diferentes puedan recorrer el mismo autómata desde dos FSM_Machine diferentes.**

Esta información se amplía en el Anexo 1.

- **FSM_Machine**

Tan importante es la labor que lleva a cabo el FSM_Manager al almacenar autómatas y asignar máquinas a las diferentes entidades, como la misión de recorrerlas adecuadamente según su tipo. Previamente se ha nombrado a la entidad FSM_Machine, un objeto creado por el FSM_Manager que recoge la lógica del autómata que se le asigna y hace que descanse sobre él, garantizando el recorrido y la gestión de la FSM.

Así pues, un autómata se instanciará una sola vez, pero cada entidad que quiera recorrerlo, deberá pedírselo al *FSM_Manager* y éste internamente instanciará una nueva *FSM_Machine* basada en dicho autómata.

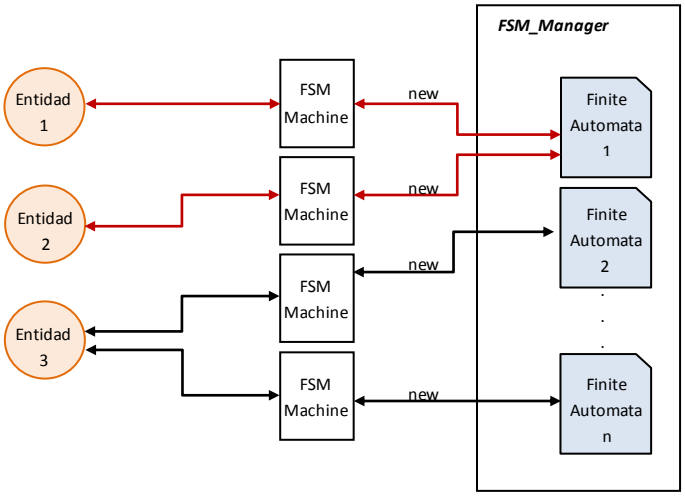


Ilustración 7. Uso de FSM_Machine

Observando las flechas en rojo de la ilustración anterior, vemos que las entidades 1 y 2 referencian al mismo autómata por medio de dos *FSM_Machine* diferentes. Este es la metodología que la API utiliza.

Esta información se amplía en el Anexo 1.

4.2 Diseño de los tipos de máquinas

En este caso, se han incluido una serie de máquinas y se han descartado otras.

Finalmente se han añadido los autómatas Clásico (FA_Classic), Inercial (FA_Inertial), basada en Pilas (FA_Stack) y la de Estados Concurrentes (FA_Concurrent States). Como todos heredan del autómata clásico, es posible que su base sea determinista o probabilista.

Por otro lado, se les ha otorgado el soporte necesario para que puedan tener carácter jerárquico y se deja en manos del diseñador que tengan carácter concurrente (aunque es posible garantizar la concurrencia de máquinas integrando una FSM de Estados Concurrentes que sea a su vez jerárquica).

Se ha creado una demostración que hace uso del API y consta de lo siguiente:

- Una escena simple con esferas que representan enemigos.
- Una esfera más grande que puede moverse con el teclado que representa al personaje del juego.

Se implementan varios escenarios en los que cada entidad utiliza una máquina o varias y se les otorga un comportamiento simple para probar las posibilidades que aporta el API de pbFSM.

4.2.1 Características generales

Para diseñar una FSM y garantizar un funcionamiento correcto, es necesario tener en cuenta una serie de aspectos, además de las particularidades de cada máquina en concreto. A continuación se explican los puntos comunes a todas las máquinas:

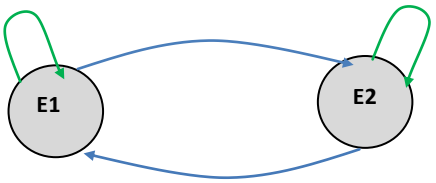
- **Al crear un autómata de los mencionados, el *FlagProbability* determina si ese autómata es probabilista (indeterminista) o determinista**. Esta especificación permite crear autómatas con una base determinista o indeterminista como tipo base y un tipo que puede ser *Classic*, *Inertial*, *Stack_based* o *Concurrent_States*.

  La idea es que haya hasta 8 combinaciones disponibles:

	Deterministic		***Probabilistic***				
Classic		*Inertial*		*Stack*		*Concurrent*	
Det.	***Prob.***	***Det.***	***Prob.***	***Det.***	***Prob.***	***Det.***	***Prob.***
Classic	*Classic*	*Inertial*	*Inertial*	*Stack*	*Stack*	*Concurrent*	*Concurrent*

Tabla 5. Combinaciones posibles

- **En la especificación son necesarios los bucles**. De esta forma se garantiza el buen funcionamiento de la estructura cuando se da un evento que no requiere la activación de otro estado.



Los **bucles** posibilitan la ejecución de un mismo estado después de un ciclo.

Aunque un estado ejecuta por defecto su lógica siempre que no llegue un evento que active la salida de éste, se recomienda introducir budes para evitar situaciones no deseadas con otras máquinas.

Ilustración 8. Bucles

- **Se pueden determinar hasta 4 posibles acciones dado un estado y una transición**. Un estado tiene una acción de entrada, una de salida y una propia del estado; las transiciones soportan una única acción. Si no hay una función asociada a la acción especificada, no ocurrirá nada.

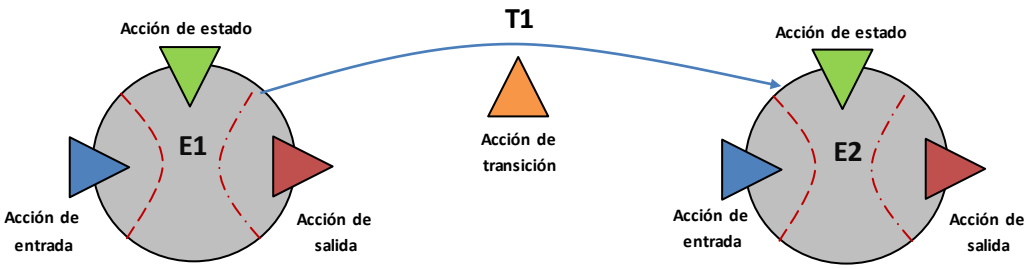


Ilustración 9. Situación de las acciones

  - **La secuencia máxima** por ciclo puede ser:

    Acc. de salida de estado (del estado viejo)
    Acc. de transición
    Acc. de entrada de estado (al nuevo estado)
    Acc. de estado

  - **Podrían haber cero acciones**. Aunque la máquina no ordenaría hacer nada. Es posible asociar de cero a cuatro acciones totales por cada par estado-transición añadido.

  - **Si la transición que ejecuta un estado es a sí mismo (un bucle), se ejecutará únicamente la acción de estado**. Las FSM funcionan de esta manera únicamente por cuestiones de diseño.

- **La máquina puede ser diseñada por programa o desde un xml**. El propio Parser proporcionado posibilita la carga desde un xml, aunque es factible crear las máquinas de cero desde donde se considere oportuno. Es una opción bastante más rápida a nivel computacional (pues no existe la estructura de carga intermedia) pero más costosa en cuanto al diseño/programación.

- **El parser de carga de xml proporcionado no forma parte de la API**. Es una herramienta adicional que se ha creado para hacer más sencilla la inserción de las FSM. No es obligatorio su uso aunque sí recomendable.

Si se utiliza el parser, todas las máquinas deben contener la información necesaria siguiendo un orden concreto que es obligatorio. ***Si no se sigue este orden, la carga no será correcta****. Se proporcionarán plantillas para cada tipo de máquina posible a añadir.*

Nota 1. Plantillas

- **Una transición puede ser activable por un evento o por más**. De esta manera se posibilita otra forma de indeterminismo en una FSM, pues es posible llegar a los mismos comportamientos por la misma vía y dados eventos diferentes.

- **Cada máquina define un procedimiento de eventos al que es sensible**. El objeto que requiera de una FSM deberá implementar dicho procedimiento de eventos. Por ejemplo, una máquina es sensible al procedimiento ***EventsMethod*** porque así lo ha definido por diseño, de forma que la entidad que requiera de esa FSM tendrá que implementar un método llamado "***EventsMethod***" que genere eventos que influyan sobre la máquina. Información ampliada en el Anexo 1.

- **Las máquinas soportan jerarquía**. Es decir, es posible especificar una submáquina dentro de un estado. La API pasará el control a la submáquina si así se requiere.

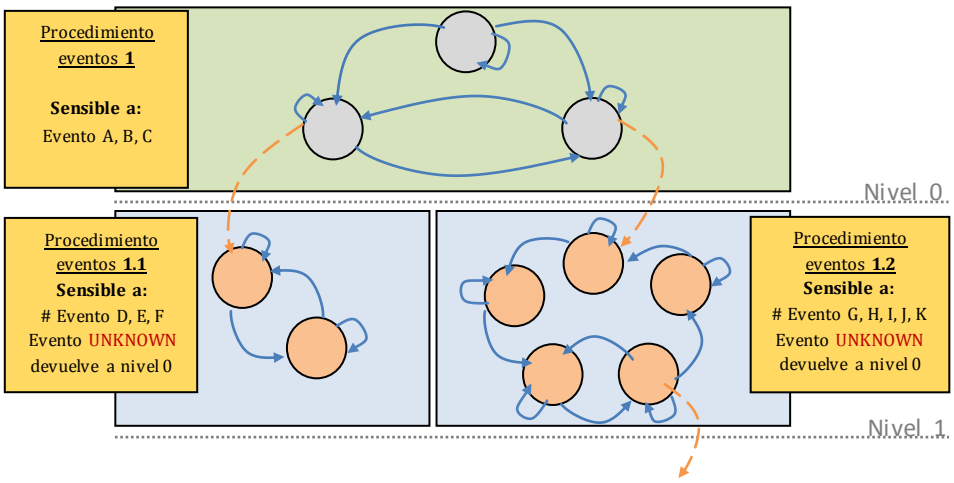


Ilustración 10. Procedimientos de eventos y jerarquía

Se muestran los procedimientos correspondientes a cada máquina, los eventos a los que es sensible dicha FSM y los eventos de retorno (Evento Unknown).

- **Se pueden especificar los niveles que sean necesarios**. Únicamente añadiendo una submáquina a un estado ya estamos incluyendo jerarquía. El número de niveles vendrá determinado por el número de estados y a su vez subestados que tengan submáquinas asociadas.
- **No se recomienda un uso masivo de la jerarquía**. Hay que tener cuidado con esta metodología porque puede llegar a hacerse incontrolable si el número de estados/subestados es demasiado grande.

- **Cada máquina define su procedimiento de eventos**. Una máquina es sensible a los eventos de su procedimiento, de forma que cada una sea independiente.

    - **Una submáquina de estados sólo retornará el control a una máquina superior cuando NO reconozca un evento**. Para la gestión correcta de una submáquina, es necesario establecer un evento desconocido para que ésta entre en estado ocioso y retorne el control a la máquina superior. (Añadiríamos un evento desconocido (UNKNOWN) en el procedimiento de eventos de dicha submáquina).

    - **La idea es que el estado que tiene la submáquina sea invisible en su gestión**. Es recomendable que este súper-estado no contenenga ninguna acción asociada para evitar casos de "acción por defecto".

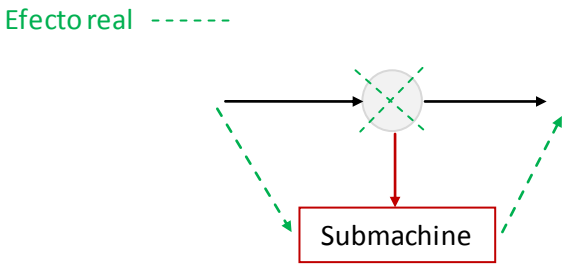


Ilustración 11. Efecto en la jerarquía

A continuación, se detallan los dos tipos base y los subtipos, con sus efectos en la demostración y los descartes o FSM no incluidas finalmente, además de cómo debe ser la especificación del diseño vía xml, ya que existen varias particularidades en cada una de ellas.

Se omite la explicación de las combinaciones del tipo base (Clásica determinista y clásica probabilista) **con los subtipos** (Inercial, Pilas y Estados

Concurrentes) para evitar repetir información, pues la explicación de estas máquinas híbridas sería redundante.

4.2.2 FSM Clásica-Determinista

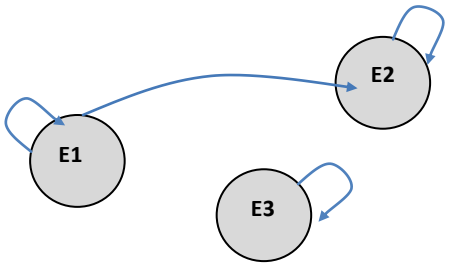
En este caso, la máquina determinista (realmente es una máquina clásica con el *flag* de probabilidad a 'NO', no se indica explícitamente que sea determinista) especifica una serie de estados, transiciones entre ellos y acciones asociadas a cada uno de estos dos "nodos". Es la máquina más simple, enfocada a los comportamientos sencillos o muy generales.

Es interesante anidar una máquina Determinista con otras, de manera que sea posible gobernar el comportamiento de un personaje con una FSM Determinista general que a su vez tenga varias FSM por debajo que otorguen ciertos matices de sofisticación de un comportamiento.

Particularidades de la máquina determinista

Existen una serie de aspectos a tener en cuenta a la hora de hacer uso de una FSM clásico-determinista.

- **No se deben especificar estados a los que no se pueda transitar**. Necesita la especificación de un grafo de estados dirigido, con transiciones entre ellos.



En este caso, no existe una transición activable al estado E3, por lo tanto es imposible alcanzarlo y ejecutar la lógica que contenga.

Ilustración 12. Estados no alcanzables

- **No es adecuada para comportamientos complejos**. Es una máquina determinista, es decir, se basa en la reacción inmediata respecto a los eventos que recibe. Se recomienda su uso para gobernar aspectos muy generales o para gestionar submáquinas asociadas a sus estados.

- **Sensible únicamente a eventos de tipo BASIC**. No tiene sentido utilizar el evento de tipo STACKABLE pues no es una máquina que soporte el apilado de estados (el uso de un evento de apilado tiene un efecto basic en la práctica para esta FSM).

- **Esta máquina admite un único evento por ciclo**. La FSM debe recibir una lista de eventos de longitud uno.

4.2.3 FSM Clásica-Probabilista

Esta máquina posibilita la inserción de probabilidades de activación en las transiciones.

Es un *modelo de máquina de estados indeterminista* y otorga la posibilidad de modelar un comportamiento mucho más impredecible que con la FSM básica.

Particularidades de la máquina probabilista

Para la creación de una máquina de estados probabilista y para su óptimo funcionamiento, es necesario tener en cuenta lo siguiente:

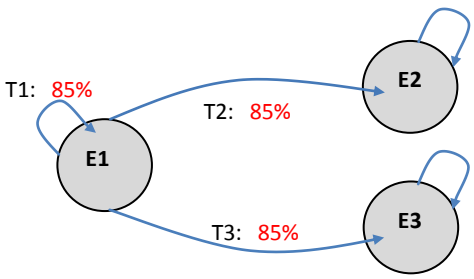
- **No se deben especificar estados a los que no se pueda transitar**. Necesita la especificación de un grafo de estados dirigido, con transiciones entre ellos.

  Sucede lo mismo que en la máquina determinista (ver ejemplo).

- **La máquina puede normalizar las probabilidades determinadas por el diseñador**. Sí y sólo si la probabilidad especificada es inferior a cero o superior a cien.

- **Para la máquina, las transiciones son independientes entre sí**. La probabilidad de la primera no afecta a la segunda, no hay acumulado de probabilidad, etc…

  Un ejemplo gráfico:



Este caso **es perfectamente posible**. La suma de probabilidades es de un 180%, pero se interpreta que cada transición es independiente.

Ilustración 13. Distribución de probabilidad

Realmente, la distribución anterior de probabilidad es correcta siempre que llegue un evento que pueda activar todas las transiciones que salen de E1

(incluido el bucle del estado). Es decir, **la probabilidad depende en parte del evento que recibe la FSM** (a la probabilidad del bucle, hay que añadir la probabilidad de que no se dé ninguna otra salida), así pues, para que ésta fuera completamente probabilista, debería realizar una redistribución de probabilidad en cada ciclo de la FSM.

Esto supone un coste por ciclo muy elevado y por tanto se ha decidido omitir, haciendo las probabilidades independientes del evento recibido, de forma que únicamente afectan cuando las transiciones se activan. La distribución de probabilidad la realiza el usuario en el diseño de la FSM, no en cada ciclo.

Esta opción trae ciertas consecuencias incontrolables en la ejecución de la FSM; la probabilidad de que se ejecute un bucle de estado siempre incluye otra probabilidad implícita variable dependiente del evento recibido:

$$Pr(bucle) \mathrel{+}= Pr(\nexists \text{ salida})$$

Ecuación 1. Probabilidad real del bucle

De forma, que la probabilidad del bucle puede ser del cien por cien aun definiéndola con un porcentaje de cero.

Este hecho se debe a que la máquina debe facilitar una vía por defecto, es factible la opción "quedarse igual" si no se activa una salida.

Ejemplo gráfico:

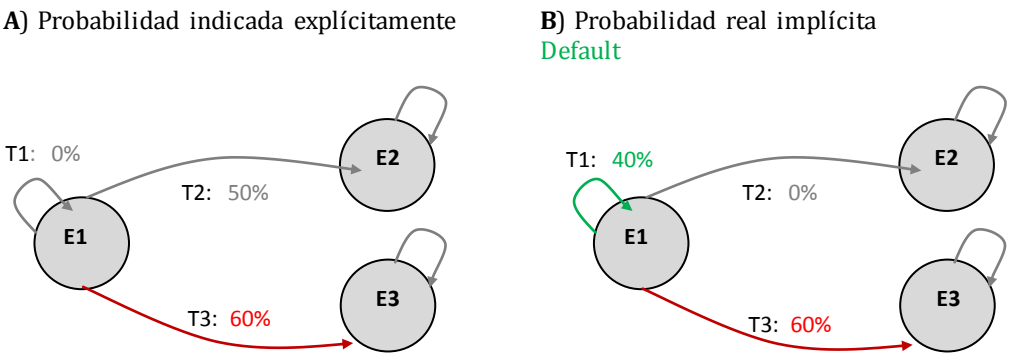


Ilustración 14. Probabilidad real

Se observa que la probabilidad real difiere de la especificada debido a que la FSM está sujeta a eventos que pueden no activar todas las transiciones (o no activar ninguna). Para el ejemplo, sólo se activa una transición (T3), de forma que la distribución de probabilidad no es la especificada en **A**, sino otra implícita **B**.

Se ha decidido implementarlo de esta manera debido a que sigue otorgando un carácter indeterminista con sencillez y rapidez.

- **La máquina trata las probabilidades de las transiciones en el orden en el que dichas transiciones han sido añadidas a un estado**. De esta forma, las que se han añadido primero son gestionadas en primer lugar y pueden activarse antes. Por tanto tienen un extra ínfimo de prioridad que no se puede controlar. Esto se debe a cuestiones de diseño de la máquina aunque se podría decir que afecta de manera despreciable al resultado final.

- **Sensible únicamente a eventos de tipo BASIC**. No tiene sentido utilizar el evento de tipo STACKABLE pues no es una máquina que soporte el apilado de estados (el uso de un evento de apilado tiene un efecto basic en la práctica para esta FSM).

- **Esta máquina admite un único evento por ciclo**. La FSM debe recibir una lista de eventos de longitud uno.

4.2.4 FSM Inercial

Esta máquina es una versión alternativa de la FSM clásica que soporta una latencia de ejecución de un estado.

Esto implica que cuando nos llega un evento que activa el cambio de estado, dicho cambio no se efectuará hasta que no se cumpla el tiempo de latencia determinado. Realmente, esta máquina supone un añadido más a la FSM clásica que evita de primera mano el problema de la oscilación de estados, donde es frecuente el cambio de estados en pocos ciclos otorgando un comportamiento excesivamente nervioso.

Particularidades de la máquina inercial

Para crear una FSM inercial de manera óptima, es necesario tener en cuenta lo siguiente:

- **No se deben especificar estados a los que no se pueda transitar**. Necesita la especificación de un grafo de estados dirigido, con transiciones entre ellos.

Sucede lo mismo que en la máquina determinista (ver ejemplo).

- **Se puede añadir latencia a un estado o no.** Es decir, si especificamos latencia cero, ese estado tratará el cambio de estado inmediatamente. Si todos los estados de la máquina tienen latencia cero, la FSM será únicamente de su tipo base (Indeterminista o determinista).

- **No es posible añadir latencia a una transición.** Se entiende que los objetos que hacen uso de una máquina mantienen su posición en los nodos/estados y no en las transiciones.

- **La latencia se añade en milisegundos.** Por ejemplo, si se especifica latencia 2000 en un estado, éste activará el cambio de estado pasados 2 segundos.

- **El tiempo de latencia no es exacto.** Aunque definamos una latencia exacta, la interacción y frecuencia de ciclo con las FSM dependerá mucho de la carga de ejecución, de forma que se generará un pequeño desfase de tiempo que se sumará a la latencia establecida. Dicho desfase puede ser ínfimo, aunque variable.

- **Sensible únicamente a eventos de tipo BASIC**. No tiene sentido utilizar el evento de tipo STACKABLE pues no es una máquina que soporte el apilado de estados (el uso de un evento de apilado tiene un efecto basic en la práctica para esta FSM).

- **Esta máquina admite un único evento por ciclo**. La FSM debe recibir una lista de eventos de longitud uno.

4.2.5 FSM Basada en pilas

Este tipo de FSM permite otorgar memoria a un personaje que está haciendo una determinada tarea siempre que la acción entrante sea más prioritaria que la que estaba realizando hasta el momento.

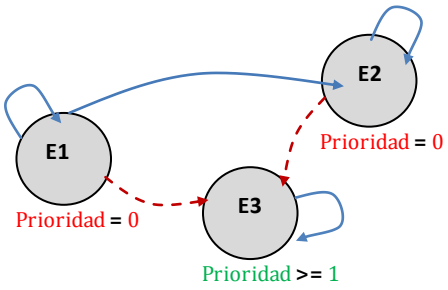
Esta máquina abre un nuevo conjunto de posibilidades, aunque su diseño pueda llegar a resultar algo tedioso.

Útil para situaciones en las que un personaje no puede acabar una tarea y empezar otra de manera secuencial y no deba abandonar totalmente el estado original en el que se encontraba.

Particularidades de la máquina basada en pilas

Para crear una FSM basada en pilas de manera correcta y garantizar un buen funcionamiento es necesario tener en cuenta los siguientes aspectos:

- **Esta máquina permite añadir prioridad a los estados**. De esta forma se da soporte al apilado de estados menos prioritarios (y que se pueden interrumpir, por tanto). Si un estado es igual de prioritario que otro, no se podrán interrumpir entre sí y sólo se podrá cambiar de uno a otro si hay transiciones entre ellos.
- **En esta FSM es necesario especificar transiciones sensibles a eventos de tipo apilable (*Stackable events*)**. Las transiciones a estados más prioritarios tendrán que ser definidas mediante eventos de tipo *stackable* para que el apilado de estados funcione correctamente.



Las **transiciones al estado E3** (más prioritario) **deben ser sensibles a eventos de tipo apilable**. De esta manera se garantiza el apilado de estados de manera correcta.

En cada ciclo se comprobará si el estado prioritario (actual) puede seguir ejecutándose o activar otro estado más prioritario. Si la FSM entiende que ha acabado, se devolverá el control al estado apilado.

Ilustración 15. Eventos de tipo stackable

- **Esta máquina admite un único evento por ciclo**. La FSM debe recibir una lista de eventos de longitud uno.

4.2.6 FSM de Estados Concurrentes (basada en redes de Petri)

Esta versión de FSM otorga un potencial añadido al API brindando la posibilidad de ejecutar varios estados al mismo tiempo. La FSM activará tantos estados como permita el diseñador, que debe indicar el número máximo de estados que se tratarán en paralelo, además de hacer uso de "créditos" de ejecución basándose en el modelo de las ***redes de Petri***.

Esta FSM es útil si queremos controlar más de un aspecto a la vez y necesitamos tratar comportamientos muy específicos, no demasiado sofisticados.

Aunque bien podríamos dividir un personaje en módulos (animación – IA – etc…) y asociar una FSM a cada uno de dichos aspectos (garantizando ya la concurrencia de máquinas), debido al carácter modular de esta FSM y al potencial jerárquico

añadido, es posible anidar máquinas a dos estados concurrentes, garantizando también implícitamente la concurrencia mencionada de FSM para controlar un mismo módulo. Gráficamente:

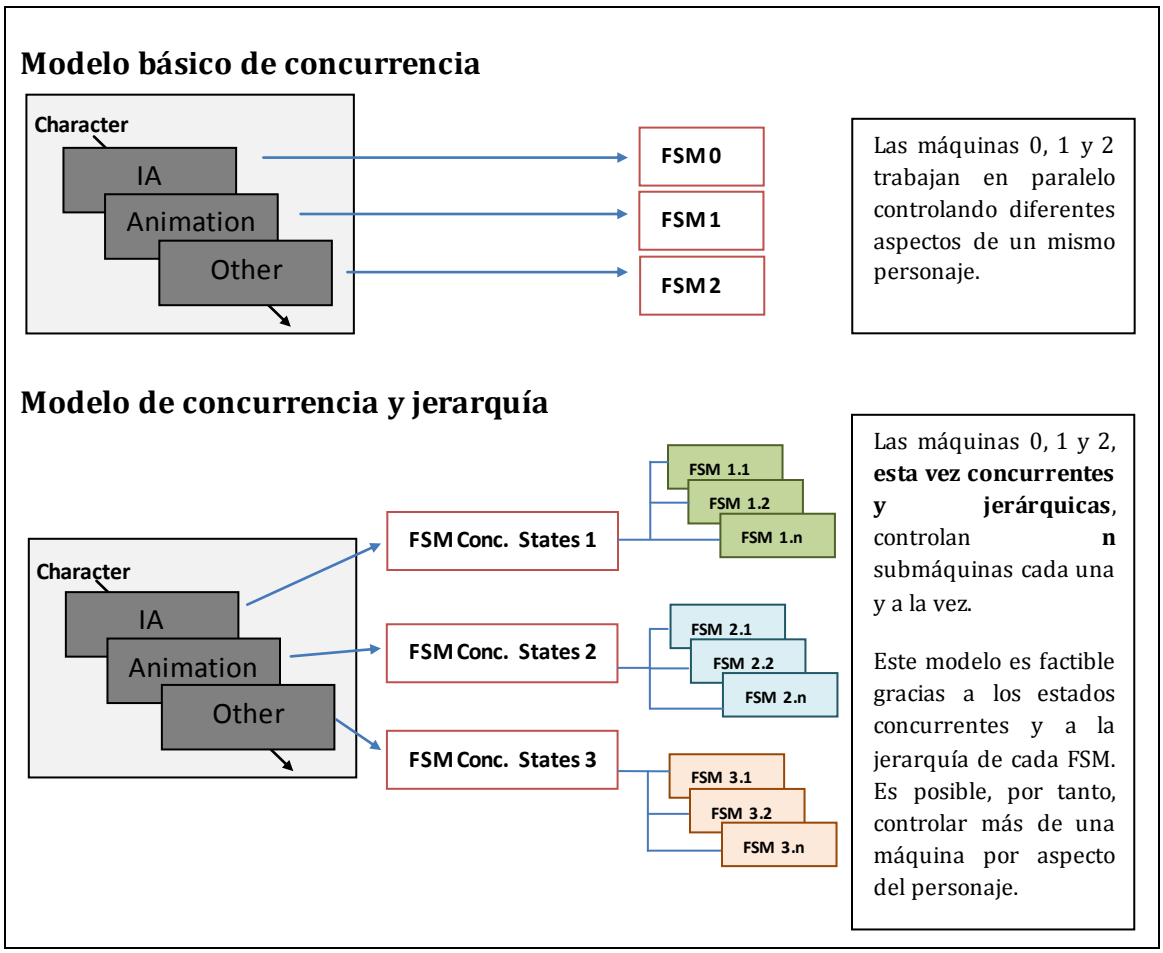


Ilustración 16. Modelos de concurrencia

Como se ha comentado, está máquina garantiza la concurrencia basándose en las redes de Petri, mencionadas y explicadas a continuación.

Redes de Petri

Una red de Petri (definida en la década de los años 1960 por Carl Adam Petri) [12] es la representación matemática o gráfica de un sistema sensible a eventos discretos como un sistema distribuido o concurrente.

Estas redes son una extensión del paradigma de autómatas que basan su carácter distribuido en marcas, las cuales son creadas en los estados activos y consumidas en los estados de los que se transita.

Dichas marcas posibilitan la ejecución paralela de la lógica o tarea de los estados en los que están presentes. En realidad, son como señales de validación para ejecutar una orden, simples consumibles o créditos.

Así pues, en una red de Petri, se reside en lugares, de los que se parte por transiciones, accesibles mediante arcos, y son consumidas (u otorgadas) fichas de validación o ejecución.

Estas fichas o marcas son idénticas en el sistema, aunque existen extensiones que introducen marcas de colores, tiempos, etc…

A continuación se ilustra un modelo de red de Petri extraído de [13] basado en el funcionamiento de una máquina expendedora.

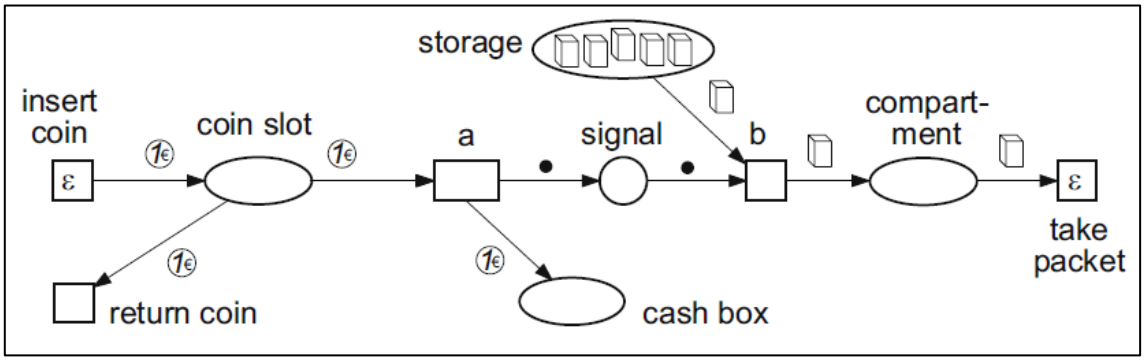


Ilustración 17. Ejemplo de una red de Petri

a – collect coin
b – give out cookie packet

Donde los "coins" son marcas en la red transmitidas que hacen que el sistema genere señales o active transiciones entre estados.

Para el estudio propuesto en la tesina, he basado mi máquina de estados concurrentes en estas marcas (llamadas créditos) que permiten la ejecución de una serie de estados activos. Dichos estados por tanto podrán ser tratados si contienen créditos de ejecución, de forma que éstos sean recibidos u otorgados en el transcurso del recorrido de la máquina de estados teniendo en cuenta los estados que llegan.

Existe una comunidad web activa [14] basada en noticias, publicaciones, avances, implementaciones y adaptaciones de las redes de Petri a diferentes tecnologías y campos de aplicación.

Particularidades de la máquina de Estados Concurrentes

Para garantizar un buen diseño y un buen funcionamiento de una FSM de Estados Concurrentes hay que tener en cuenta los siguientes aspectos:

- **El diseñador debe indicar en la creación el número máximo de estados activos/concurrentes**. Si el número de estados activos es 1, la FSM es una máquina clásica al uso.

- **La FSM sólo activará todos los estados iniciales al principio**. En cada ciclo, acorde a lo sucedido en el anterior, determinará los que siguen activos y deben ejecutarse en el nuevo ciclo.
    - **Los créditos (puntos de ejecución, Redes de Petri) son en realidad contadores existentes en los estados que determinan la ejecución de éstos**. Si un estado tiene créditos de ejecución podrá estar activo. Si se salta de un estado a otro, el primero otorgará un crédito de ejecución al segundo. Si el primero se queda sin créditos quedara inactivo.

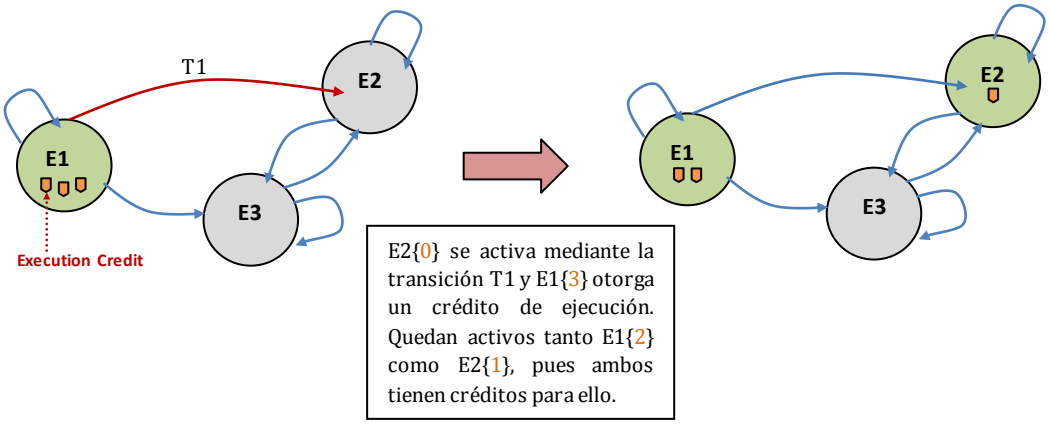


Ilustración 18. FSM de estados concurrentes utilizando créditos de ejecución

- **La FSM no contempla que los estados se solapen**. Es decir, se sobreentiende que la especificación de un ejemplo de esta FSM no debería incluir estados que afecten a la ejecución de otros, pues podría darse un problema de ***incoherencia de estados***.

    - **La incoherencia de estados aparece cuando la ejecución de un estado activo afecta a la de otro también activo**. No es lógico que se ejecuten en paralelo, por ejemplo, los estados 'andar' y 'correr'.

- **Un estado no concede créditos si se ha ejecutado en ese ciclo y únicamente dispone de uno.**

- **Sensible únicamente a eventos de tipo BASIC**. No tiene sentido utilizar el evento de tipo STACKABLE pues no es una máquina que soporte el apilado de estados (el uso de un evento de apilado tiene un efecto basic en la práctica para esta FSM).

- **Es la única FSM que admite la gestión de más de un evento por ciclo.** Las demás únicamente admitían uno por ciclo y respondían acorde a él.

4.2.7 Descartes

Se ha decidido omitir la explicación relativa a las múltiples combinaciones de FSM híbridas que soporta la API por no ofrecer nada novedoso respecto a lo visto en las demostraciones ya mostradas, pues por ejemplo, una FSM híbrida que sea probabilista e inercial, aporta lo visto en la probabilista más lo visto en la inercial.

Además de lo anterior, es una tarea imposible si se tiene en cuenta la jerarquía, ya que las combinaciones son dependientes del diseño y el número de combinaciones y niveles de FSM puede ser inalcanzable para crear una demostración cerrada que las incluya todas.

4.2.8 Resumen del diseño

Se muestra a continuación una tabla que resume los diferentes aspectos/características de cada máquina.

Tipo FSM	Número eventos por ciclo	Tipo eventos	Jerarquía	Latencia de estados	Estados prioritarios	Complejidad en el diseño	Estados en paralelo
FSM Clásico-determinista	1	BASIC	Sí	No	No	Baja	No
FSM Clásico-probabilista	1	BASIC	Sí	No	No	Media	No
FSM Inertial	1	BASIC	Sí	Sí	No	Baja	No
FSM Stack based	1	BASIC/STACKABLE	Sí	No	Sí	Alta	No
FSM Concurrent States	>= 1	BASIC	Sí	No	No	Muy alta	Sí

Tabla 6. Resumen del diseño

4.3 El flujo de datos

Para clarificar todo aquello que ocurre desde que se lanza la aplicación hasta que se cierra, considero necesario sintetizar y explicar el flujo que debería seguir la información.

Dependiendo del punto de partida del que decidamos dar uso a esta herramienta, dicho recorrido podría verse levemente reducido. Así pues, hay dos enfoques: la inserción de las máquinas **a través de una herramienta/tecnología externa**, o la **creación por programa** de éstas.

4.3.1 Por programa

En este caso, la carga de datos queda parcialmente omitida, de forma que la creación de nuestras máquinas de estados finitos es bastante más rápida. De todas formas, el diseñador debe ser el que valore si es interesante realizar *ad hoc* el trabajo de creación de las FSM para cada proyecto software que se tenga que desempeñar.

Además, bajo mi punto de vista, de igual forma que el coste computacional se reduce considerablemente en la fase de carga utilizando la creación por programa, el coste por trabajo de un desarrollador se incrementa de manera notable siguiendo esta vía, así que quizá el ahorro sea innecesario e incluso contraproducente en algunos casos.

Sin la figura de un *parser* de información que nos permita introducir la información rápidamente, se hace necesaria la inclusión de estados, transiciones, acciones y eventos posibles desde el código. Es necesario instanciar el FSM_Manager con el constructor de la clase por defecto (*public FSM_Manager()*).

A continuación se muestra un diagrama del recorrido que realiza la información siguiendo esta vía para el ejemplo del videojuego.

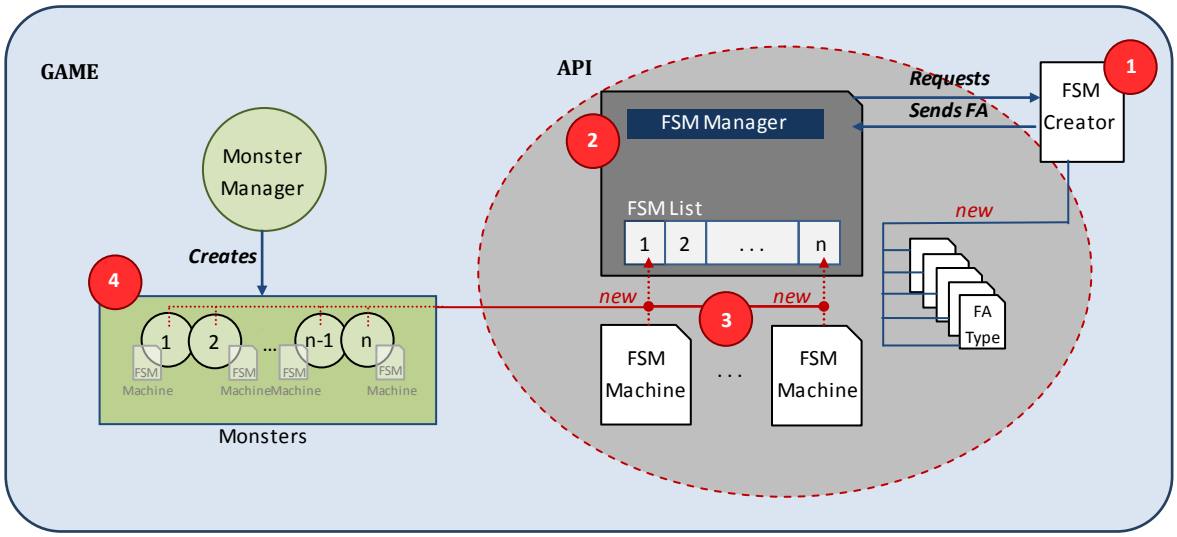


Ilustración 19. Flujo de datos

1 Clase o jerarquía de clases (si se requiere, llamada FSM Creator únicamente en el gráfico) que se encarga de mandar las órdenes necesarias a la API para cargar/crear los AF (Autómatas Finitos) que serán recorridos como FSM. Crea las instancias de los AF disponibles en la API siempre que sea necesario. Las disponibles son:

- FA_Classic
- FA _Inertial
- FA _Concurrent_States
- FA _Stack

Con prefijo (FA) – Finite Automaton – se pretende dar a entender que la entrada que admite la API son autómatas finitos. Posteriormente, de forma interna, se utilizarán para generar FSM que se puedan recorrer.

Finalmente, el FSM_Manager contendrá todos los autómatas y éstos podrán ser referenciados. En caso de utilizar un parser de información propio, será necesario cargar cada máquina al FSM_Manager utilizando el método de adición oportuno, *public void addFSM(FA_Classic),* donde *FA_Classic* es una instancia de autómata finito (que puede ser de cualquier subtipo).

2 Así pues, el FSM_Manager se encarga de almacenar los Autómatas Finitos generados tras la carga desde fichero y/o los creados por programa, además de atender las peticiones de las diferentes clases que soliciten las FSMs basadas en dichos autómatas. Es necesario instanciar un FSM_Manager desde la clase que controle la IA.

Internamente, se irá almacenando cada autómata finito en un diccionario definido como Dictionary<TKey, TValue>, donde TKey son dos claves (Un int que determina el tipo de autómata finito a cargar y un string que lo identifica) y TValue es la instancia del autómata finito a almacenar.

3 Cuando una entidad solicita al FSM_Manager una máquina de un tipo determinado, éste crea la referencia al autómata finito solicitado (presente en su diccionario de AFs) y crea un nuevo objeto FSM_Machine que se encargará del recorrido de dicho autómata. Es aquí donde aparece el término Máquina de Estados o FSM.

NOTA
La idea es que cualquier objeto que sea usuario potencial de las FSM tenga una referencia al FSM_Manager correspondiente para poder solicitarle las máquinas pertinentes. Para las demos en Unity, el objeto que tiene una referencia al FSM_Manager es el MonsterManager, encargado de crear instancias Monster.

Nota 2. Nota sobre el FSM_Manager

4 El ejemplo de la creación de una de las entidades que usan las FSM. En este ejemplo, hay un *MonsterManager* que se encarga de crear *Monsters*, entidades que tienen un objeto *FSM_Machine* que inicializan solicitando al FSM_Manager una de las máquinas mediante *FSM_Manager.CreateMachine(System.Object, FSM_Tag, FSM_ID),* donde System.Object es una referencia al objeto que usará la *FSM* (normalmente a sí mismo, *this*), FSM_Tag es el Token asociado a esa máquina (Ver tags), y FSM_ID es un *string* identificador de la FSM.

4.3.2 Por vía externa (vía por defecto)

Para esta vía, el flujo de información es exactamente igual, pero no se utiliza un parser propio, ni se cargan las FSM "a mano", sino **un objeto de tipo xmltest.FSM_Parser**, que se ocupa de la carga automática de cada FSM desde

documentos de texto etiquetado xml. Es necesario instanciar el FSM_Manager utilizando el constructor *public FSM_Manager(parser)* donde parser es una instancia de tipo xmltest.FSMparser, el parser de información aportado como herramienta adicional a la API.

Gráficamente:

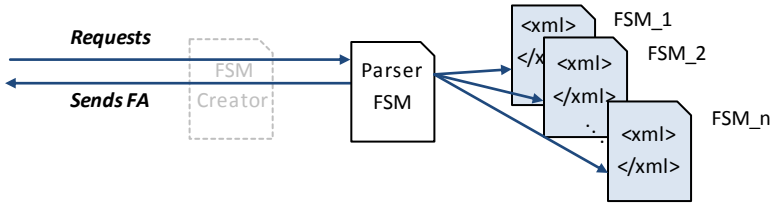


Ilustración 20. Utilización de FSM_Parser

Se observa que la clase encargada de la creación o carga es sustituida por otro objeto encargado de extraer la información pertinente (*xmltest.FSM_Parser*).

4.4 Observaciones importantes

Considero necesario clarificar ciertos aspectos relativos al diseño y que afectan a la implementación, pues en ocasiones podrían confundirse términos que parecen hacer referencia a lo mismo o detalles necesarios para el buen funcionamiento.

1) *El diseñador crea Máquinas de Estados Finitos o FSM (Finite State Machine). Ahora bien, la API almacena Autómatas finitos o FA (Finite Automata).*

El FSM_Manager (explicado posteriormente) forma parte del núcleo de la API y almacena autómatas finitos que serán referenciados para crear Máquinas de Estados Finitos que serán las que garanticen el recorrido.

Así pues, es necesario diferenciar entre la Máquina de estados entendiendo ésta como la entidad a recorrer, y el autómata o grafo en el que se basan, que es la estructura de datos inteligible por la API.

Podríamos decir que el diseñador debe transformar su FSM diseñada a la estructura que entiende la API y pasarla como entrada a ésta en forma de autómata finito (FA), y posteriormente, la salida es una máquina basada en dicho autómata (FSM).

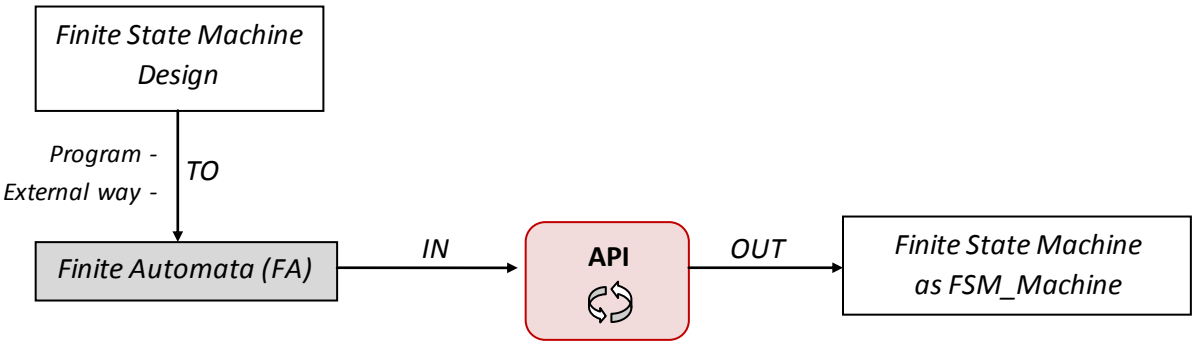


Ilustración 21. Entrada y salida de la API

Tanto en la creación de FSM por programa o por vía externa (explicadas después), el diseñador debe conocer que la API recibe en realidad un simple grafo o autómata. Cuando una entidad requiera de una FSM, solicitará al FSM_Manager la que corresponda, y será éste el que internamente referencie dicho autómata y otorgue una FSM_Machine como salida.

2) ***Es necesario identificar numéricamente cada objeto que forma parte de un autómata. Estados, Transiciones, Eventos, Acciones, y la propia estructura de datos (Autómata/Grafo).***

En cada ciclo de programa se hace imposible realizar múltiples comparaciones si los identificadores son cadenas de caracteres, es mucho más eficiente en términos computacionales realizar comparaciones numéricas.

Así pues, es muy recomendable que el diseñador realice la tarea de definir una serie de tags en la clase Tags y que rellene el método correspondiente a la conversión de cadena a Tag (String2Tag) también presente en dicha clase. Todo esto es opcional, pero tremendamente útil para llevar el control de los identificadores añadidos y tenerlos localizados.

Habrán una serie de Tags fijas que necesita la API para su funcionamiento (No se deben borrar), como son las de los tipos de autómatas añadidos (Classic, Inertial, Stack_based y Concurrent_States) y otras que serán variables y por tanto *ad-hoc* al programa o videojuego a desarrollar (eventos, acciones, estados, y transiciones definidas).

NOTA MUY IMPORTANTE

NO PUEDE HABER DOS ESTADOS/TRANSICIONES/EVENTOS/ACCIONES CON EL MISMO IDENTIFICADOR NUMÉRICO

Es posible (no necesario) que un estado comparta identificador con una transición (por ejemplo, estado_ataque.id = transición_atacar.id), pues entre ellos no son solapables, pero dos entidades iguales (dos estados, o dos transiciones, etc…) no pueden tener el mismo identificador pues no tendría sentido y podría entorpecer el funcionamiento.

La aplicación no hace un análisis interno de los identificadores añadidos, así pues, es necesario tener en cuenta estas pautas para evitar en la medida de lo posible identificadores repetidos y para ello se recomienda encarecidamente el uso de la clase Tags.

Nota 3. Uso de los tags

Este aspecto se explica con más detalle en las secciones posteriores.

4.5 Utilidades añadidas

A continuación se explica el proceso de volcado de información mediante el *parsing* de documentos y algunos datos sobre el software utilizado en el desarrollo de proyecto.

4.5.1 Parsing

A continuación se especifica el proceso relativo al parsing de los documentos xml que contienen el diseño de las máquinas de estados.

Introducción

En un primer momento, teniendo como ejemplo el proyecto de Space Invaders (un juego creado desde cero) visto en la asignatura de Animación por Computador y Videojuegos, se pretendía realizar el parsing de manera similar (realizar un análisis "al vuelo" del documento XML y utilizar un sistema de pilas para extraer información). Por ahorro de tiempo, se ha procedido de manera distinta, aunque más costosa en términos computacionales.

Para mantener el carácter modular del proyecto, se han creado diferentes documentos xml que contienen el diseño de cada máquina de estados y se ha realizado un parseado de la información contenida en dichos documentos de texto utilizando el espacio de nombres System.XML (una nueva interfaz de acceso a datos XML que ofrece la plataforma .NET como sustituto de la anterior, SAX), que construye un grafo en memoria y posibilita la extracción de información de cualquier etiqueta del documento XML en cualquier momento.

Debido a que dicho grafo no es una estructura de datos conocida por nuestro programa, se ha creado una estructura de datos básica que permite crear un grafo (a partir de la información extraída) e ir añadiéndole nodos (estados) y transiciones entre ellos (API pbFSM).

En resumen, se extrae la información de un XML mediante la estructura resultante de usar System.XML y se vuelca en una estructura de datos conocida que permite el flujo de información y ser recorrida con el fin de ejecutar diferentes acciones (especificadas en cada estado/transición, en el diseño).

FSM_Parser

En esta sección explicaré el uso del espacio de nombres System.XML en el programa para extraer la información del documento XML.

Como comentaba antes, se crean las diferentes máquinas de estados en distintos documentos de texto XML para posteriormente cargar cada uno de ellos y extraer la información relevante.

Es necesario que el objeto FSM_Manager sea inicializado con una instancia de la clase FSM_Parser para que posteriormente se le pueda comunicar la ruta del documento de diseño correspondiente a cargar.

Internamente se crea una instancia XmlDocument y mediante su función Load(string path) se carga en memoria el documento xml correspondiente, se consultan los nodos cargados y se va rellenando una estructura de datos propia (Clases FA_X /State/Transition) que permite crear un grafo, insertar estados y transiciones.

Esta herramienta constituye un añadido muy interesante y verdaderamente útil a la API, ya que su uso puede ahorrar mucho trabajo. Como comentaba en secciones anteriores, es cierto que la fase de carga es más lenta en términos computacionales (se utiliza una estructura adicional e intermedia XmlDocument, además de nodos y transiciones de tipo XmlNodeList que luego hay que reconvertir), pero supone una ventaja considerable en la creación de cada FSM, tarea que puede hacerse muy complicada si dicha máquina es de un tamaño considerable.

La información relativa al FSM_Parser se amplía en el Anexo 1.

CAPÍTULO 5. Pruebas y resultados

En la presente sección se exponen las pruebas realizadas con la herramienta en una caja de arena, además de exponer algunos detalles relativos al rendimiento obtenido.

5.1 Pruebas: demostración de la herramienta

A continuación se mencionan y discuten las pruebas a las que se ha sometido la herramienta creada.

5.1.1 Introducción

Se ha decidido implementar esta demostración en la que se mostrara el uso de la herramienta y la importancia en las diferencias entre las posibilidades que ofrece.

Esta caja de arena, ha ido creciendo a la vez que la fase de análisis y la de diseño. En un primer momento fue necesario avanzar el diseño e implementación de dicha demostración para posteriormente refinar el análisis y en consecuencia las demás fases.

Aunque a priori, puede constituir un error adentrarse en la implementación de una demostración y descuidar temporalmente el análisis, cabe decir que gracias a esta metodología ha sido posible realizar los cambios más importantes relativos al diseño de las máquinas de estados.

5.1.2 Proyecto de demostración

Como proyecto de demostración se ofrece un banco de pruebas desarrollado en Unity, donde se otorga una escena y varias clases que hacen uso de la API. A continuación se muestra un listado con los diferentes recursos que conforman el proyecto:

- La clase **Monster.cs** que representa a las entidades que hacen uso de la API (esferas pequeñas).

- La clase **MonsterManager.cs** que representa la capa de control que crea y sitúa monsters en la escena.

- **Una clase Tags ya rellenada** con lo necesario en esta demostración.

- Un directorio **Resources** en donde hay varios scripts de movimiento (*PruebaMovimiento.cs* y *MovimientoAtaque.cs*) relativos a las esferas (*monsters*) y el prefab que representa a un *monster (MonsterPrefab.prefab)*, que se invoca desde *MonsterManager.cs*.

- Un script en Javascript llamado **Sphere_test.js** utilizado para mover por teclado la esfera principal que representa al personaje jugable.

- Un script **Game.cs** que hace uso de la API referenciando lo necesario.

- Dos directorios **FSM** y **Parser** que contienen las herramientas desarrolladas. En Parser residen también documentos xml como ejemplos de máquinas de estados.

- Un directorio **Templates** que incluye **plantillas para los diferentes tipos de máquinas de estados** y una plantilla para la clase Tags llamada **Tags_Template.cs**.

- También hay una serie de Assets por defecto (*Standard Assets*).

- Una escena **Demo.unity** que agrupa y utiliza todo lo especificado. La escena está formada es una caja cuya base tiene una textura de barro.

- Al ejecutar la aplicación, se vuelca el proceso de parsing a un fichero txt llamado **ParsingLog.txt** situado en el directorio Parser, en el que se puede observar la carga de máquinas que se ha llevado a término.

Para probar la demostración en Unity, es necesario abrir el proyecto con el motor, y abrir la escena Demo.unity. Una vez abierta hay que ejecutarla (hacer clic sobre el botón play).

Es posible la **interacción** por teclado. **El usuario puede**:

- Mover la esfera principal con las teclas direccionales y observar la reacción de las esferas pequeñas.

- Se puede alternar entre las diferentes máquinas con los botones 0, 1, 2, 3 y 4.

- Es posible reiniciar la aplicación pulsando R.

- Se puede hacer clic sobre las esferas pequeñas y seleccionarlas. Se extraerá un panel de información.

A continuación se explica una demostración por cada máquina diferente incluída.

Demostración de la Máquina Clásica-Determinista

Para la demo, la API hace uso del Parser de información y carga la FSM desde un fichero XML.

Cada enemigo/esfera de la escena tiene una única FSM clásico-determinista. Los enemigos estarán en el estado SPAWN hasta que detecten al personaje (esfera blanca) y manden un evento de ataque que active el estado ATTACKING. Si dejan de detectar al personaje, enviarán un evento de espera y pasarán al estado WAITING.

Para ilustrar la jerarquía, se ha decidido incluir una submáquina de estados sujeta al estado ATTACKING. De esta forma, el estado de ataque delega su gestión a otra máquina de un nivel inferior.

Los "monstruos" perciben, generan y mandan un evento y reciben una respuesta de la API acorde a ese evento generado. Al ser determinista, el evento que se envía activará la correspondiente transición siempre que sea alcanzable desde el estado actual (Current State).

Se muestra un fragmento del ejemplo creado utilizando el parser de xml.

```
<?xml version="1.0" encoding="utf-8" ?> <!--Author: José Alapont Luján-->

<FSMtype Probabilistic="NO">CLASSIC</FSMtype> <!--FSM specification -->
<FSMid>MonsterClassicDet</FSMid>
<Fsm>
        <Callback>CheckEvents</Callback> <!--Method for events that concern to this FSM -->
        <States>
                <State Initial="YES">
                        <S_Name>SPAWN</S_Name>
                        <S_Action>ACTION_ATTACK</S_Action>
                        <S_inAction>NULL</S_inAction>
                        <S_outAction>NULL</S_outAction>
                        <S_Fsm></S_Fsm>
                </State>
                <State Initial="NO">
                        <S_Name>ATTACKING</S_Name>
                        <S_Action>NULL</S_Action>
                        <S_inAction>NULL</S_inAction>
                        <S_outAction>NULL</S_outAction>
                        <S_Fsm>/SubFSM/Attacking.xml</S_Fsm> <!– link to subFSM -->
                </State>
                <!–there are more states-->
                  . . .
        </States>
        <Transitions>
                <Transition>
                        <T_Name>ATTACK</T_Name>
                        <T_Origin>SPAWN</T_Origin>
                        <T_Destination>ATTACKING</T_Destination>
                        <T_Action>NULL</T_Action>
                        <Events>
                                <Event>
                                        <ID>EVENT_ATTACK</ID>
                                        <Type>BASIC</Type>
                                </Event>
                                <!-- add more events if it is required-->
                        </Events>
                </Transition>
                <!–there are more transitions-->
                . . .
```

```
        </Transitions>
</Fsm>
```

Tabla 7. Fragmento xml relativo a la FSM clásica determinista

No se incluye el xml completo debido a su longitud. De igual forma, se muestra lo necesario para ver el formato del documento.

Si se desea conocer con mayor profundidad lo que significa cada etiqueta, **recomiendo echar un vistazo al Anexo 3**, que contiene las plantillas relativas a cada máquina de estados.

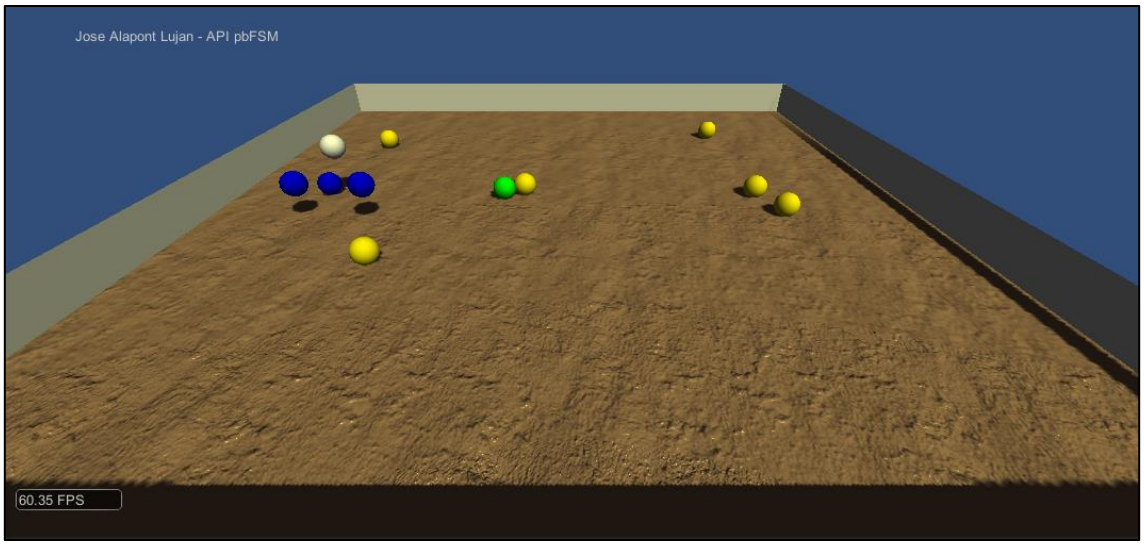


Ilustración 22. Demo clásica determinista

En la imagen se aprecian esferas de hasta 3 colores (sin contar la esfera blanca). Las esferas azules se corresponden con el estado PHYSICAL_ATTACK de una submáquina adherida al estado ATTACKING de la FSM superior.

Las demás esferas están en el estado WAITING (amarillo) o SPAWN (verde).

Demostración de la Máquina Clásica-Probabilista

En este caso, la FSM también se especifica vía xml haciendo uso de la herramienta FSM_Parser.

La demo creada es muy sencilla pero ilustra perfectamente el indeterminismo que se quiere obtener con este tipo de FSM. Básicamente se ha dividido el ataque en dos estados ATTACK y ATTACK2. Al establecer probabilidades en sus transiciones de entrada, siendo estas sensibles al mismo evento, existe la posibilidad de ir a uno o a otro estado.

En el resultado final hay enemigos que atacan de una forma y otros que atacan de otra. Su comportamiento se torna impredecible.

A continuación se muestra un fragmento del documento xml que modela la correspondiente FSM clásica-probabilista.

```
<?xml version="1.0" encoding="utf-8" ?> <!--Author: José Alapont Luján-->

<FSMtype Probabilistic="YES">CLASSIC</FSMtype> <!--FSM specification -->
<FSMid>MonsterClassicProb</FSMid>
<Fsm>
        <Callback>CheckEvents</Callback> <!--Method for events that concern to this FSM -->
        <States>
                <State Initial="YES">
                        <S_Name>SPAWN</S_Name>
                        <S_Action>PRUEBA</S_Action>
                        <S_inAction>NULL</S_inAction>
                        <S_outAction>NULL</S_outAction>
                        <S_Fsm></S_Fsm>
                </State>
                <State Initial="NO">
                        <S_Name>ATTACKING</S_Name>
                        <S_Action>PRUEBA2</S_Action>
                        <S_inAction>NULL</S_inAction>
                        <S_outAction>NULL</S_outAction>
                        <S_Fsm></S_Fsm>
                </State>
                    .
                    .
                    .
        </States>
        <Transitions>
                <Transition>
                        <T_Name>ATTACK</T_Name>
                        <T_Origin>SPAWN</T_Origin>
                        <T_Destination>ATTACKING</T_Destination>
                        <T_Action>NULL</T_Action>
                        <Events>
                                <Event>
                                        <ID>EVENT_ATTACK</ID>
                                        <Type>BASIC</Type>
                                </Event>
                        </Events>
                        <Probability>100</Probability>
                </Transition>
                    .
                    .
                    .
       </Transitions>
</Fsm>
```

Tabla 8. Fragmento xml relativo a la FSM clásica probabilista

Se subraya de color amarillo las principales diferencias con la FSM clásica sin probabilidad. En este caso, el flag Probabilistic está definido como 'YES' y cada transición tiene una etiqueta <Probability> asociada.

Si se desea conocer con mayor profundidad lo que significa cada etiqueta, **recomiendo echar un vistazo al Anexo 3**, que contiene las plantillas relativas a cada máquina de estados.

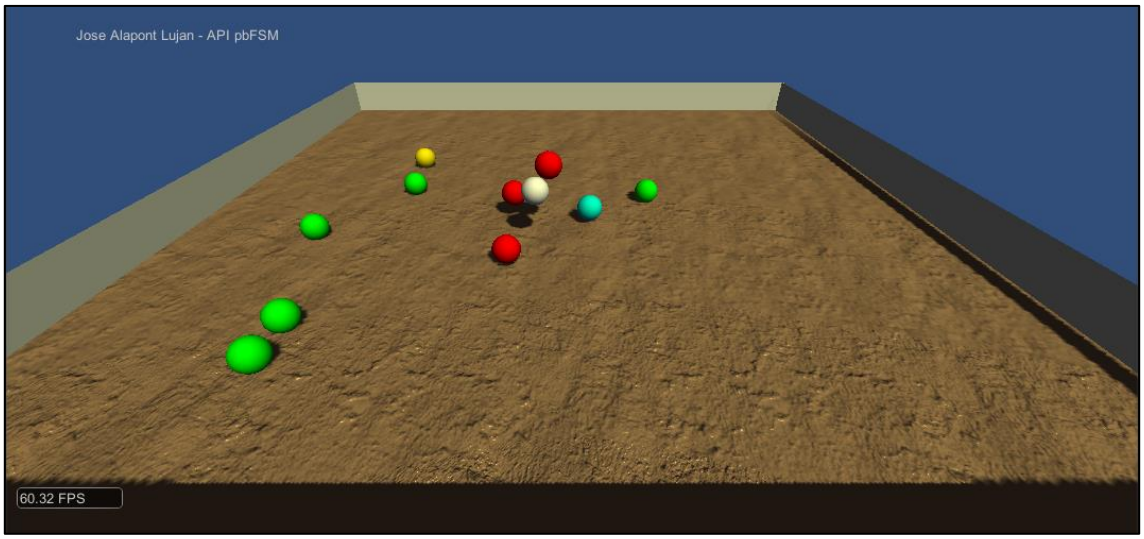


Ilustración 23. Demo clásica probabilista

En la imagen se pueden apreciar esferas de cuatro colores diferentes (sin contar la blanca).

Las rojas y cyan, están en dos estados diferentes de ataque y por tanto atacando de forma distinta pese a que sus respectivas FSM reciben el mismo evento (EVENT_ATTACK), de ahí el indeterminismo; esto lo garantiza la probabilidad de activación de estados.

Las demás están en el estado SPAWN (verde) o WAITING (amarillo).

Demostración de la Máquina Inercial

Se ha utilizado la misma demostración que en la determinista, esta vez con la máquina inercial, pues ilustra perfectamente el funcionamiento/idea de este modelo de FSM.

Los enemigos funcionan de la misma manera, se mantienen en el estado SPAWN hasta que ven al personaje (esfera blanca) y pasan al estado ATTACKING, sólo que dicho estado tiene un tiempo de latencia establecido y tardarán más en dejar de atacar y pasar al estado WAITING. Del mismo modo, cambiar del estado WAITING a ATTACKING llevará una latencia añadida y no será inmediato.

Como ya he comentado, este tipo de máquina es útil para evitar comportamientos nerviosos donde la ejecución de la lógica es demasiado cambiante.

A continuación se muestra un fragmento del documento xml que modela la FSM inercial (cuya base es determinista).

```
<?xml version=”1.0” encoding=”utf-8” ?> <!--Author: José Alapont Luján-->

<FSMtype Probabilistic="NO">INERTIAL</FSMtype> <!--FSM specification -->
<FSMid>MonsterIner</FSMid>
<Fsm>
        <Callback>CheckEvents</Callback> <!--Method for events that concern to this FSM -->
```

```
        <States>
                <State Initial="YES">
                        <S_Name>SPAWN</S_Name>
                        <S_Action>PRUEBA</S_Action>
                        <S_inAction>PRUEBA</S_inAction>
                        <S_outAction>PRUEBA</S_outAction>
                        <S_Fsm></S_Fsm>
                        <S_Latency>2000</S_Latency>
                </State>
                    .
                    .
                    .
        </States>
        <Transitions>
                  <Transition>
                        <T_Name>ATTACK</T_Name>
                        <T_Origin>SPAWN</T_Origin>
                        <T_Destination>ATTACKING</T_Destination>
                        <T_Action>NULL</T_Action>
                        <Events>
                                <Event>
                                        <ID>EVENT_ATTACK</ID>
                                        <Type>BASIC</Type>
                                </Event>
                        </Events>
                </Transition>
                    .
                    .
                    .

        </Transitions>
</Fsm>
```

Tabla 9. Fragmento xml relativo a la FSM inercial

Subrayadas de color amarillo las principales diferencias respecto a la FSM clásica (en este caso, únicamente la latencia de estado).

Si se desea conocer con mayor profundidad lo que significa cada etiqueta, **recomiendo echar un vistazo al Anexo 3**, que contiene las plantillas relativas a cada máquina de estados.

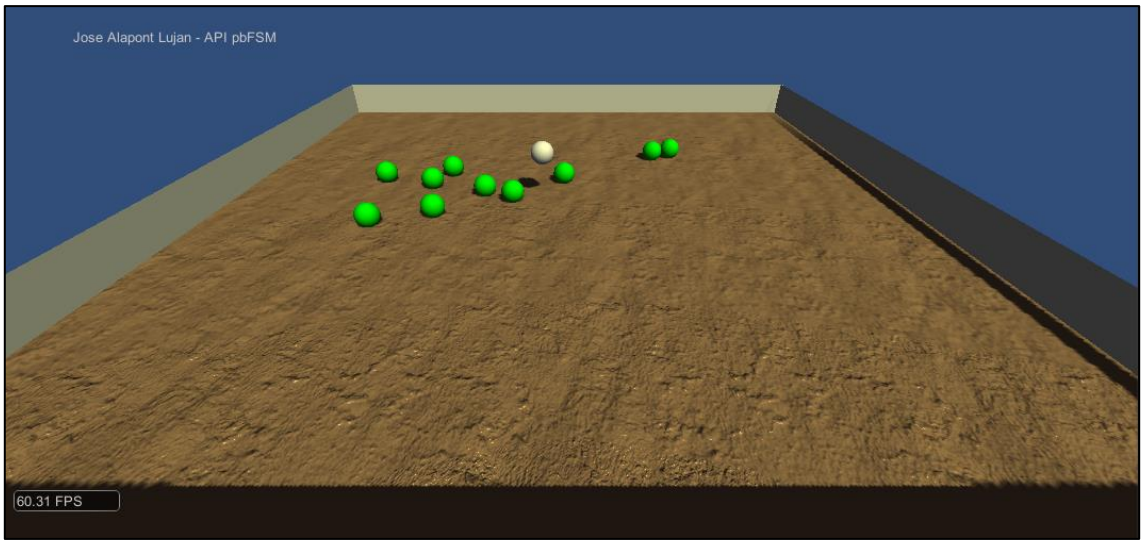


Ilustración 24. Demostración de la máquina inercial

En la imagen anterior todas las esferas están en el mismo estado SPAWN, pero no todas reciben el mismo evento.

Las cercanas a la esfera blanca, están recibiendo el evento de ataque EVENT_ATTACK. Pese a ello, no ejecutan la lógica necesaria para trasladarse de estado debido a su latencia de permanencia en su estado actual. Pasados dos segundos, atacan.

Demostración de la Máquina basada en Pilas

Se muestra a continuación un fragmento del documento xml para la FSM basada en pilas.

```
<?xml version="1.0" encoding="utf-8" ?> <!--Author: José Alapont Luján-->
<FSMtype Probabilistic="NO">STACK_BASED</FSMtype> <!--FSM specification -->
<FSMid>MonsterStack</FSMid>
<Fsm>
        <Callback>CheckEvents</Callback> <!--Method for events that concern to this FSM -->
        <States>
                <State Initial="YES">
                        <S_Name>SPAWN</S_Name>
                        <S_Action>PRUEBA</S_Action>
                        <S_inAction>XXX</S_inAction>
                        <S_outAction>PRUEBA2</S_outAction>
                        <S_Fsm></S_Fsm>
                        <S_Priority>0</S_Priority>
                </State>
                   .
                   .
                   .
        </States>
        <Transitions>
                <Transition>
                        <T_Name>STOP</T_Name>
                        <T_Origin>ATTACKING</T_Origin>
                        <T_Destination>STOPPED</T_Destination>
                        <T_Action>NULL</T_Action>
                        <Events>
                                <Event>
                                        <ID>EVENT_STOP</ID>
                                        <Type>STACKABLE</Type>
                                </Event>
                        </Events>
                </Transition>
                   .
                   .
                   .

      </Transitions>
</Fsm>
```

Tabla 10. Fragmento xml relativo a la FSM basada en pilas

Subrayadas en amarillo las diferencias con la FSM clásica. En este caso, la etiqueta prioridad en los estados que determina el peso que tienen en la gestión de la FSM y el uso de eventos de tipo STACKABLE que apilan el estado del que parten (T_Origin).

Si se desea conocer con mayor profundidad lo que significa cada etiqueta, **recomiendo echar un vistazo al Anexo 3**, que contiene las plantillas relativas a cada máquina de estados.

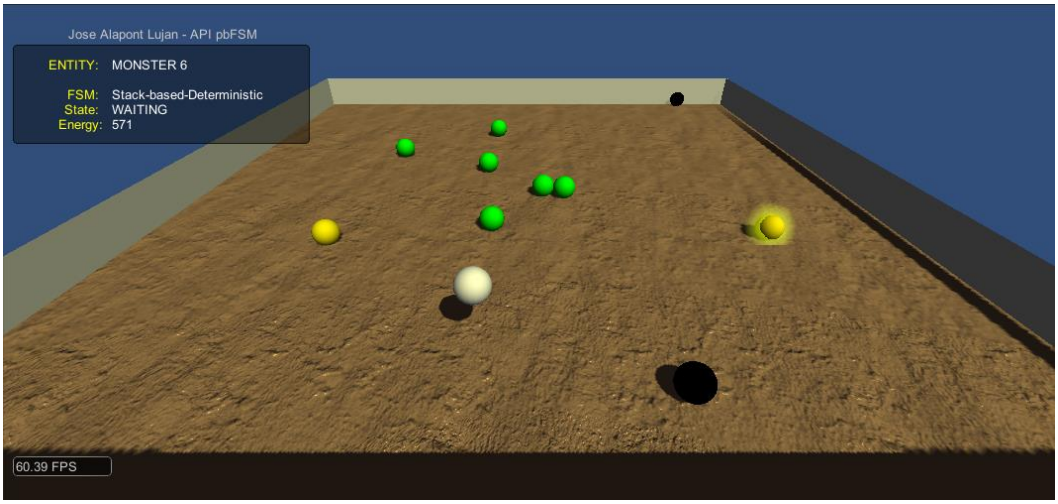


Ilustración 25. Demostración de la máquina basada en pilas

En la imagen anterior se aprecian hasta 3 colores diferentes sin contar la esfera controlable (blanca).

La esfera negra de la parte inferior de la imagen estaba en el estado ATTACKING y por tanto atacaba a la esfera blanca. Cuando ha agotado su energía, ha apilado el estado de ataque y ha tratado un evento de tipo STACKABLE que requería pasar al estado STOP, con una prioridad mayor al anterior. De esta forma, en dicho estado de parada, las esferas recuperan en energía; cuando está completa, vuelven al estado anterior interrumpido.

Las demás esferas están en el estado SPAWN (verde) o WAITING (amarillo) consumiendo energía. Si sobrepasan el umbral de los mínimos de energía, transitarán al estado STOP también.

Una de las esferas amarillas está seleccionada (de ahí ese halo amarillo), pueden verse sus datos en la parte superior izquierda de la imagen (Identificador, FSM asociada, estado actual y energía).

Demostración de la Máquina de Estados Concurrentes

También se hace uso de la herramienta FSM_Parser. En este caso, se ha decidido hacer un diseño de FSM acorde al tipo de máquina, pues en el ejemplo anterior de ataque y espera, los estados eran excluyentes.

Existen cuatro estados, dos de ellos iniciales (*COLOR*, *MOVEMENT*, *SHAPE* y *JUMP*). Estos estados no son excluyentes entre sí y pueden ejecutarse en paralelo.

En cada ciclo se mandarán hasta dos eventos de forma que siempre haya dos estados ejecutándose a la vez y mostrando dos comportamientos.

A continuación se muestra un fragmento del documento xml que modela la FSM de estados concurrentes.

```
<?xml version="1.0" encoding="utf-8" ?> <!--Author: José Alapont Luján-->

<FSMtype Probabilistic="NO">CONCURRENT_STATES</FSMtype> <!--FSM specification -->
<FSMid>MonsterConcu</FSMid>
<Fsm MaxConcurrent="2">
        <Callback>CheckEvents</Callback> <!--Method for events that concern to this FSM -->
        <States>
                <State Initial="YES">
                        <S_Name>COLOR</S_Name>
                        <S_Action>CHANGECOLOR</S_Action>
                        <S_inAction>NULL</S_inAction>
                        <S_outAction>NULL</S_outAction>
                        <S_Fsm></S_Fsm>
                        <S_Credits>1</S_Credits>
                </State>
                    .
                    .
                    .
         <States>
         <Transitions>
                <Transition>
                        <T_Name>TOCOLOR</T_Name>
                        <T_Origin>COLOR</T_Origin>
                        <T_Destination>COLOR</T_Destination>
                        <T_Action>NULL</T_Action>
                        <Events>
                                <Event>
                                        <ID>EVENT_COLOR</ID>
                                        <Type>BASIC</Type>
                                </Event>
                        </Events>
                </Transition>
                    .
                    .
                    .
         </Transitions>
</Fsm>
```

Tabla 11. Fragmento xml relativo a la FSM de estados concurrentes

En amarillo, como antes, quedan subrayadas las principales diferencias con la FSM clásica. En este caso, el atributo MaxConcurrent de la etiqueta Fsm, que define un tope de estados concurrentes y la etiqueta S_Credits relativa a un estado que hace referencia a los créditos de ejecución de éste (basado en las redes de Petri).

Si se desea conocer con mayor profundidad lo que significa cada etiqueta, **recomiendo echar un vistazo al Anexo 3**, que contiene las plantillas relativas a cada máquina de estados.

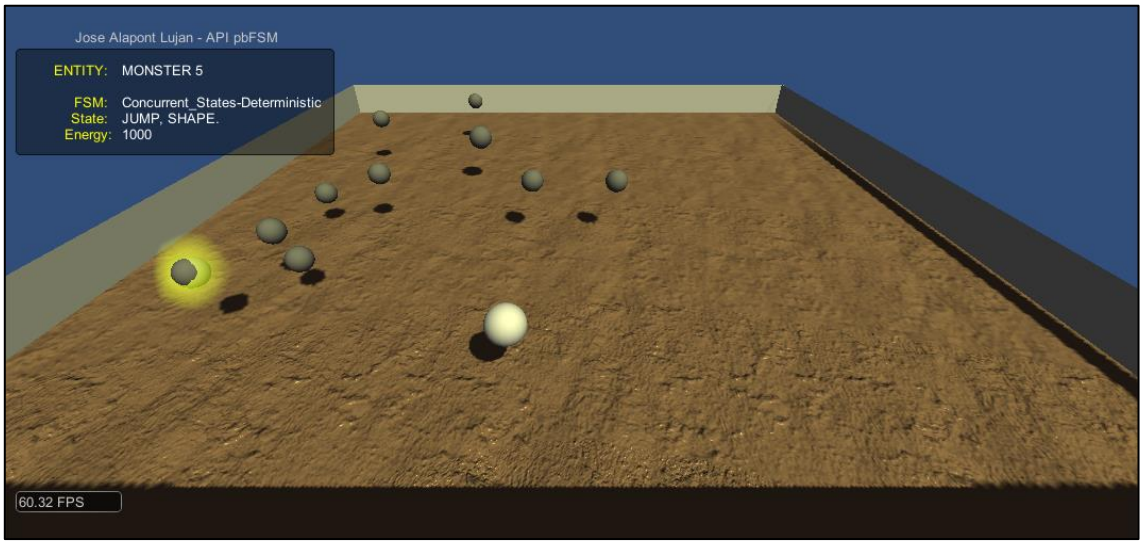


Ilustración 26. Demostración de la máquina de estados concurrentes

En la anterior imagen, todas las esferas están en dos estados a la vez. Obviamente, al ser una imagen estática no se puede observar bien el comportamiento, pero están cambiando su forma (estado SHAPE) y saltando (estado JUMP).

Se puede observar en las características de la esfera con identificador "MONSTER 5" que está seleccionada.

En este caso, a diferencia de los anteriores, la esfera blanca no influye en nada sobre las demás.

5.1.3 Rendimiento

Las demostraciones se han realizado en el siguiente terminal:

- *Alienware x17 Intel(R) Core(TM) i7-3630QM CPU 2.40 GHz*
- *Memoria RAM 16 GB*
- *Sistema operativo Windows 8.1 de 64 bits*
- *Gráfica AMD Radeon HD 7970M*

Dichas pruebas se han llevado a cabo primera instancia con **diez esferas** como NPCs sensibles al uso de las máquinas de estados integradas en la herramienta. El uso de hasta diez máquinas no alteraba el *frame rate*, estable y constante por encima de los 60 *fps*.

Se han hecho **pruebas de rendimiento gráfico** de hasta doscientas esferas sin cambios excesivamente grandes en el tiempo de cómputo. El coste de la ejecución de la aplicación se resiente poco aun usando un número elevado de FSM, llegando a estabilizarse en los 58 *fps* utilizando la máquina de estados concurrentes, que es la más exigente en términos de coste debido a la cantidad de cálculos que realiza internamente.

Previsiblemente, con mayor carga gráfica, esta tasa podría verse afectada en mayor medida, pues en esta demostración, se hacen uso de las primitivas básicas del motor *Unity3D*.

Por otro lado, **la fase de carga**, en la que se hace uso del *parser* de información implementado como utilidad adicional y el posterior volcado implícito a la estructura de datos de la herramienta, supone un coste computacional despreciable, incluso teniendo en cuenta que dicha fase puede suponer un coste elevado si hay una ingente cantidad de documentos xml con el diseño de las FSM a cargar.

Se hizo uso de la librería *System.Diagnostics* y de la clase *Stopwatch,* que proporciona herramientas de medición de tiempos. La carga relativa a la demostración, supuso un coste temporal de menos de *150 ms*.

CAPÍTULO 6. Conclusiones del trabajo

A continuación se exponen las conclusiones tras los meses de trabajo en esta tesis, las materias que han hecho posible el desarrollo y el trabajo futuro.

6.1 Conclusión

Después de meses de trabajo en el proyecto y de los resultados que se han obtenido con las demostraciones es posible afirmar que se ha logrado algo funcional y que además aporta un rápido nivel de control sobre la capa de inteligencia artificial de un proyecto de desarrollo de software.

Si bien es cierto que se conoce la utilización de máquinas de estados finitos en muchos de los proyectos que se han llevado a cabo en la industria, además de suponer su uso en muchos otros, no hay constancia de que exista una librería que aúne características de esta técnica de inteligencia artificial para este tipo de proyectos de desarrollo, o al menos, no es de acceso sencillo y/o público.

Este hecho era un aliciente más para llevar a término este trabajo, aparte de realizar una aportación significativa al sector de los videojuegos como primera toma importante de contacto, con la idea de seguir formándome en este ámbito y entrar a formar parte de la industria.

Considero que los objetivos determinados al inicio se han cumplido; se ha desarrollado un proyecto desde un punto de vista pragmático y que a priori es innovador (entendiéndolo como conjunto).

Las pruebas realizadas confirman que la herramienta es sensible a múltiples combinaciones de modelos de máquinas de estados finitos, aunque existen otros no incluidos que podrían llevarse a cabo o incluso se podría innovar en este sentido modelando nuevos tipos de autómatas que no están contemplados en el estado del arte.

Aun así, se han incluido cuatro variantes (incluyendo la visión probabilista de la máquina clásica) importantes de la máquina estándar, y se ha logrado implementar la jerarquía de máquinas como característica común a todas ellas, pese a que el primer enfoque la entendía como una máquina de estados finitos adicional.

Todas aportan matices que las diferencian y otorgan un potencial adicional a la API, además de multiplicar las posibilidades para realizar ejemplos prácticos.

No es el primer trabajo largo e importante que llevo a término, aunque sí que lo considero uno de los más útiles, ya no sólo por el hecho de que no existen demasiadas librerías que traten esta problemática y se basen en las máquinas de

estados finitos, sino porque es relativamente simple y práctico, aunque soy consciente de que es un proyecto mejorable.

Algunos términos empleados dan lugar a confusión y esto se ha querido reducir lo máximo posible, además de que hacer una aplicación genérica que pueda utilizarse en cualquier proyecto de desarrollo de software implica más trabajo por parte del diseñador y/o programador del que hubiera deseado, pues cada proyecto es variable y requerirá de unas máquinas u otras. Un ejemplo claro de trabajo extra para el diseñador, es la construcción de la clase Tags, que hasta el final intenté que fuera lo más automática posible, idea que deseché por ser muy poco (o nada) viable.

Estoy bastante satisfecho con los resultados obtenidos tras varios meses de trabajo, especialmente, como ya digo, por la utilidad práctica del programa (enfocado para ser utilizado en un entorno que permita el desarrollo de aplicaciones en *C#*, como el ahora de moda el creciente motor de videojuegos *Unity3D*).

6.2 Disciplinas del posgrado

Este trabajo no hubiera sido posible sin algunas de las materias vistas en el posgrado de Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital cursado en la Universidad Politécnica de Valencia.

Las más importantes para esta tesina han sido aquéllas basadas en Inteligencia Artificial, pues es el campo de aplicación de este trabajo.

Destacan sobre todo *Animación por Computador y Videojuegos*, impartida por Ramón Mollá (director de esta tesina) por sus contenidos basados en videojuegos (Inteligencia Artificial desde este prisma) y las asignaturas de agentes, como *Sistemas Inteligentes* o *Sistemas Multiagente*, impartidas por Vicente Julián Inglada y Carlos Carrascosa, pues descubrí *Unity* en ellas, además del paradigma de agentes, que siempre guarda similitud.

En menor medida (por su docencia más general), estaría *Técnicas de Inteligencia Artificial* (asignatura de nivelación) y su segunda parte *Aplicación de Técnicas de Inteligencia Artificial*, impartidas ambas por Federico Barber, en la que conocí las técnicas hoy en día utilizadas en el campo de la Inteligencia Artificial y en las cuales se basan algunas de las máquinas de estados finitos revisadas. También fue importante en el inicio de esta tesis (que iba a ser un proyecto conjunto con mis dos compañeros, Javier López Punzano y Steve Rossius) la asignatura de *Realidad Virtual y Aumentada*, impartida por M. Carmen Juan Lizandra, aunque finalmente se avanzó más por la parte de la IA y se dejó aparte el enfoque basado en esta disciplina.

6.3 Trabajo futuro

Esta sección podría ser muy extensa. La herramienta podría ser complementada con diferentes utilidades adicionales que facilitarían y harían su uso más cómodo.

Además, existe la posibilidad de implementar de nuevo ciertos aspectos relativos al diseño de algunos autómatas, los cuales se han llevado a cabo siguiendo una vía y tomando una serie de decisiones que no son absolutas.

6.3.1 Nuevas FA Clásica-probabilista

El enfoque empleado en la API otorga una serie de posibilidades en lo referente a este tipo de autómata que podrían ampliarse o mejorarse. Por el momento, este autómata está basado en las transiciones probabilistas, independientes unas de otras y también de los eventos de entrada a cada FSM_Machine.

Se decidió llevar a cabo este punto de vista debido a su eficiente funcionamiento y a que cumplía con el objetivo de lograr el indeterminismo perseguido.

Otro punto de vista podría ser el de realizar una normalización de la distribución de probabilidad realizada por el usuario, que en principio se llevaría a cabo en el arranque del autómata, determinando que todas las transiciones de salida de un estado sumaran el cien por cien. Esto tiene varios inconvenientes, y es que como la distribución de probabilidad es independiente del evento, la distribución podría llegar a contener valores muy bajos, mermando el posterior uso de la máquina, además de no garantizar los valores de probabilidad indicados por el usuario al inicio del diseño de la FSM.

Esta implementación podría llevarse a cabo en el método Start() de cada FA; el programador debería recontar las transiciones de salida de un estado y redistribuir la probabilidad realizando una normalización siguiendo una serie de fórmulas para ello.

Teniendo en cuenta este sumatorio de todas las probabilidades:

$$\sum_{i=0}^{n} \Pr(i)$$

Ecuación 2. Sumatorio de probabilidades

*Siendo **n** el número total de transiciones de salida probabilistas de un estado* →

Si el sumatorio es > 100	$\Pr(i)_{norm} = \frac{\Pr(i)*100}{\sum total}$
Si este sumatorio es < 100	$\Pr(i)_{norm} = \frac{\sum total*\Pr(i)}{100}$

Ecuación 3. Probabilidad normalizada

Por otro lado, otro tercer enfoque podría ser contemplado: una máquina de estados completamente probabilista que distribuyera la probabilidad por ciclo de ejecución y que fuera completamente dependiente del evento.

Esta máquina sería lo más fiel a lo que una máquina de estados probabilista pura representa. Tiene el inconveniente de ser realmente costosa computacionalmente.

La idea sería que el programador reimplementara parte del núcleo de la API, concretamente, la clase *FSM_Machine*.

En la sección probabilista de cada máquina, **debería**:

- Recontar las transiciones activables por el evento que llega.
- Una vez extraídas, distribuir la probabilidad acorde al porcentaje indicado por el usuario siguiendo las mismas fórmulas especificadas arriba.
- Finalmente, otorgar a cada transición una probabilidad de activación acorde a la normalización en tiempo de ciclo.

6.3.2 Herramientas de apoyo

Otra clase de complementos a la API creada podrían ayudar y facilitar la tarea de diseño de máquinas de estados finitos, de igual manera que lo hace el *FSM_Parser*.

Algunos **ejemplos**:

- **Una aplicación que permitiera insertar gráficamente las diferentes entidades necesarias de la API** (estados, transiciones…), con la posibilidad de exportar el resultado gráfico a documento en formato xml.
    - Muchas veces es necesario realizar dibujos de los grafos. Si la máquina de estados finitos que se pretende obtener es muy compleja, esta tarea es casi obligatoria.

- **Una aplicación de detección de errores ortográficos**. Esta utilidad evitaría muchos errores en el diseño de las máquinas donde es fácil cometer errores al escribir algunas cosas en varias partes del código.

    - El programador debería llevar a cabo una comparación entre los Tags definidos explícitamente y la información extraída de los xml, informando de posibles incongruencias. Esto reduciría el porcentaje de error considerablemente.

Referencias

[1] RUSSELL, Stuart J. & NORVIG, Peter (2009) Artificial Intelligence, a modern approach

[2] RABINER, L. R. & JUANG, B. H. (1986) An introduction to hidden Markov models

[3] SCHWAB, Bryan (2004) AI Game Engine Programming

[4] RABIN, Steve (2005) Introduction to Game Development

[5] SELIC, Bran, GULLEKSON & WARD, Paul T. (1994) Real-Time Object-oriented Modeling

[6] CHAKRABORTY, Samarjit (2003) Formal Languages and Automata Theory. Regular Expressions and Finite Automata

[7] MOLLÁ, Ramón (2013) Animación por Computador y Videojuegos (MIARFID)

[8] BARBER, Federico (2013) Técnicas de Inteligencia Artificial (MIARFID)

[9] GameDevelopment, Artificial Intelligence Tutorials: Finite State Machines, Theory and Implementation http://gamedevelopment.tutsplus.com/categories/artificial-intelligence

[10] Unity3D Game Engine web page http://unity3d.com/es

[11] Trees and Graphs http://www.nakov.com/blog/2014/01/13/free-programming-book-csharp-fundamentals-nakov-presentations-slides-videos-lessons-exercises-tutorial

[12] REISIG, Wolfgang (1985) Petri Nets – An Introduction

[13] REISIG, Wolfgang (2013) Understanding Petri Nets

[14] Community Petri Nets http://www.informatik.uni-hamburg.de/TGI/PetriNets/

[15] Vuforia Developer Portal https://developer.vuforia.com/

[16] ShiVa 3D Game Engine http://www.stonetrip.com/

[17] Unity3D FSM Animation use http://unity3d.com/es/unity/animation

[18] List of Game Engines http://devmaster.net/devdb/engines

[19] AZUMA, R. (2011) Special Section on Mobile AR

[20] GameDev. Introducción a Wave Engine http://www.gamedev.es/?p=1272

ANEXO 1. GUIA DE USUARIO

Se proporciona a continuación una guía de usuario con el fin de facilitar el uso de la API mediante recomendaciones y consejos varios, así como la manera de controlar ciertos aspectos relativos a la creación de máquinas de estados finitos.

Introducción

Aunque algunos de los aspectos que incluye la API sean bastante intuitivos y su uso no requiera entrar demasiado en detalle, hay otros que sí requieren una guía para que el usuario potencial de la herramienta pueda entender y clarificar matices que reduzcan significativamente la aparición de errores o excepciones.

Se ha decidido orientar la guía a la creación de ejemplos que ilustren qué tipos de máquinas son más adecuados y a todo lo relativo al proceso de creación y carga de una máquina desde cero o mediante el uso de la herramienta FSM_Parser.

Estructura de esta guía

La guía sigue el siguiente índice. En total hay 2 capítulos, el primero basado en la creación y carga de las FSM y el segundo en la interacción con ellas.

El índice es el siguiente:

CAPÍTULO 1. PROCESO DE CREACIÓN Y CARGA DE UNA MÁQUINA

1. **Creación de una Máquina de Estados Finitos**
   - 1.1 Crear una FSM usando la herramienta FSM_Parser
   - 1.2 Crear una FSM desde cero
   - 1.3 Jerarquía de máquinas
   - 1.4 Definir una FSM con base determinista o probabilista
     - *1.4.1 Base determinista o probabilista desde cero*
     - *1.4.2 Base determinista o probabilista con FSM_Parser*
   - 1.5 Consejos y sugerencias

2. **Creación y uso de Estados**
   - 2.1 Constructores de un Estado
   - 2.2 Creación y uso de un Estado
     - *2.2.1 Creación y uso de un Estado con FSM_Parser*
     - *2.2.2 Creación y uso de un Estado desde cero*
   - 2.3 Creación de un estado jerárquico
     - *2.3.1 Creación de un estado jerárquico con FSM_Parser*
     - *2.3.2 Creación de un estado jerárquico desde cero*

CAPÍTULO 1. PROCESO DE CREACIÓN Y CARGA DE UNA MÁQUINA

En este apartado se realiza un análisis del proceso de creación de una máquina hasta su carga en el programa.

1. Creación de una Máquina de Estados Finitos

En realidad, como se ha dicho en la memoria de trabajo, la API recibe autómatas que posteriormente, mediante el uso de otra entidad adicional, convierte en máquina de estados finitos que puede ser recorrida.

Así pues, es necesario diferenciar entre la creación e inclusión de estas Máquinas utilizando la herramienta FSM_Parser y la creación desde cero (menos recomendable).

1.1 Crear una FSM usando la herramienta FSM_Parser

Utilizando la herramienta FSM_Parser el proceso de creación y carga es mucho más sencillo. Se explica en pasos cómo se crea una FSM desde fichero.

1. **Creación de un documento xml que contenga el diseño de la FSM.**
   El primer paso se basa en la elaboración de un documento etiquetado que albergue lo necesario en una FSM dependiendo de su tipo (Se proporcionan plantillas para cada uno de los tipos).

   En las propias plantillas se indica qué etiqueta es opcional rellenar y qué otras son obligatorias.
2. **Preparación de la clase Tags (Obligatorio)**
   La clase Tags es una entidad no instanciable que la API utiliza en cada ciclo constantemente, lo cual significa que su preparación es de gran importancia.

   Es necesario preparar dicha clase con las constantes enteras que referencien cada entidad creada en el documento xml del punto 1. Es decir, **siempre que creemos un estado nuevo, una transición nueva, un evento nuevo o una acción nueva, debemos referenciarla como constante entera y completar el StringToTag de la clase Tags** como sea oportuno.

   Es importante que dos entidades del mismo tipo pero diferentes como tal (dos estados, dos transiciones, dos eventos o dos acciones) no compartan el mismo Tag. Por ejemplo:
   - Creamos los **estados** *ATTACK* y *DEFENSE* (comparten tipo, pero son dos estados completamente diferentes).

- Definimos **acciones,** una para cada uno, respectivamente *AttackWithSword* y *DefenseWithShield*.

- Creamos **transiciones** entre los estados, llamadas *ToDEFENSE* y *ToATTACK.*

- Creamos eventos, uno que active cada transición, respectivamente *EVENT_ATTACK* y *EVENT_DEFENSE*.

La clase Tags de ejemplo, debería quedar de la siguiente manera:

```
public static class Tags {

    /*
    TAGS NO MODIFICABLES OMITIDOS
    ...
   */

    //Tags de los estados
    public const int ATTACK   = 0;
    public const int DEFENSE  = 1;
    //Tags de las acciones
    public const int AttackWithSword     = 0;
    public const int DefenseWithShield   = 1;
    //Tags de las transiciones
    public const int ToATTACK    = 0;
    public const int ToDEFENSE   = 1;
    //Tags de los eventos
    public const int EVENT_ATTACK    = 0;
    public const int EVENT_DEFENSE   = 1;

    public static int StringToTag(string word){

        switch (word[0]) {
        case 'A':

            if(word.Equals("ATTACK"))
                return Tags.ATTACK;
            if(word.Equals("AttackWithSword"))
                return Tags.AttackWithSword;
        break;

        case 'D':
            if(word.Equals("DEFENSE"))
                return Tags. DEFENSE;
            if(word.Equals("DefenseWithShield"))
                return Tags.DefenseWithShield;
        break;

        case 'E':
            if(word.Equals("EVENT_ATTACK"))
```

```
                    return Tags. EVENT_ATTACK;
                if(word.Equals("EVENT_DEFENSE"))
                    return Tags. EVENT_DEFENSE;
            break;

            case 'T':
                if(word.Equals("ToATTACK"))
                    return Tags.ToATTACK;
                if(word.Equals("ToDEFENSE"))
                    return Tags.ToDEFENSE;
              break;

            }
        }
        /*Código omitido*/
}
```

NOTA

La FSM resultante es muy sencilla, únicamente creada para ilustrar lo correspondiente a la clase Tags.

3. Uso del FSM_Parser y su Log de carga

- En la clase que controle la IA del juego crearemos el parser:

```
xmltest.FSM_Parser parser = new xmltest.FSM_Parser();
```

- Lanzaremos el log que notificará en un documento de texto todo lo que sucede en la carga:

```
//Por defecto lo creamos vacío y se nos creará en la ruta por defecto
parser.StartLog("");
```

4. Uso del FSM_Manager con el objeto FSM_Parser creado

- Posteriormente creamos el FSM_Manager correspondiente que gestionará una carga de máquinas:

```
//Lo creamos pasándole el parser inicializado como parámetro
```

```
FSM_Manager fsm_manager = new FSM_Manager(parser);
```

- El siguiente paso ya es indicarle al manager una ruta para que pueda utilizar su parser mediante uno de sus métodos *addFSM()*:

```
//Como parámetro pasamos el path absoluto al fichero xml de la FSM
fsm_manager.addFSM("C:/Proyecto/FSM/Determinista.xml");
```

Internamente, el manager intentará acceder al documento xml pasado como parámetro con el parser que le hemos asociado en el punto anterior. Si no hubiera un parser añadido previamente, el método *addFSM()* devolvería un error (ver especificación de la API).

Este método es el encargado de llamar a los métodos correspondientes de la clase FSM_Parser que a su vez se encargan de crear los autómatas, haciendo uso de los objetos FA_Tipo que existen en la API. **Estos métodos tendremos que invocarlos a mano de no usar el parser**.

5. **Cierre del Log de carga**

- **Tras la carga** se recomienda cerrar el Log del parser, abierto antes.

```
//Log inicializado antes como parser.StartLog("")
parser.FinishLog();
```

Si no se ha abierto un Log al inicio con *StartLog*, llamar a *FinishLog* provocará una **excepción *NullReferenceException***. Se debe tratar esta utilidad del parser como un bloque, es decir, si lanzamos el *Start*, lo propio es lanzar el *Finish* en algún momento.

Así pues, este método cerrará el fichero abierto previamente (y liberará memoria).

1.2 Creación de una FSM desde cero

El proceso de creación de una máquina de estados finitos "a mano" o desde cero es un proceso bastante más complejo que utilizando la herramienta FSM_Parser. De

esta forma tenemos que realizar las invocaciones que realizaba la herramienta de carga parser internamente, lo cual supone un coste en la práctica bastante importante, aunque el coste computacional se ve reducido también.

Se explica, de igual manera que antes, en varios pasos:

1. **Creación de un objeto FSM_Manager vacío**
   A diferencia de antes, el objeto FSM_Manager no tiene un parser para realizar la carga, así pues, se instancia vacío en la clase que deba controlar la IA:

```
//Creación de un objeto FSM_Manager vacío
FSM_Manager fsm_manager = new FSM_Manager();
```

2. **Preparación de la clase Tags (Opcional pero recomendable)**
   Se recomienda el uso de esta clase para asociar los tags correspondientes rápidamente. De todas formas, su uso es opcional, y de no usarla, es necesario definir los identificadores a mano, lo cual conlleva cierto riesgo por errores de tags repetidos, etc…

   En el siguiente paso veremos cómo crear los autómatas en el caso de que se use la clase Tags y en el caso de que se omita su uso.

3. **Creación de un autómata**
   Es necesario tener en cuenta muchas cosas para crear un autómata: se requiere la definición del propio autómata, además la adición de todos los estados, transiciones, acciones y eventos que necesite para estar completo.

   La mejor forma de ver las similitudes y diferencias, es haciendo una comparativa de lo definido en un xml (usando el parser) y de las invocaciones que éste hace internamente. Así pues, a continuación se muestra una tabla que pretende clarificar los pasos a seguir para la creación de un autómata mediante una comparación entre el diseño de una FSM en un documento xml y las llamadas internas que se realizan para cargar dicha máquina:

Diseño xml de la FSM

```xml
<?xml version="1.0" encoding="utf-8" ?>
  <FSMtype Probabilistic="NO">CLASSIC</FSMtype>
  <FSMid>Maquina_A</FSMid>
  <Fsm>
    <Callback>CheckEvents</Callback>
    <States>
      <State Initial="YES">
          <S_Name>ATTACK</S_Name>
          <S_Action>AttackWithSword</S_Action>
          <S_inAction>NULL</S_inAction>
          <S_outAction>NULL</S_outAction>
          <S_Fsm></S_Fsm>
      </State>
      <State Initial="NO">
          <S_Name>DEFENSE</S_Name>
          <S_Action>DefenseWithShield</S_Action>
          <S_inAction>NULL</S_inAction>
          <S_outAction>NULL</S_outAction>
          <S_Fsm></S_Fsm>
      </State>
    </States>

    <Transitions>
      <Transition>
        <T_Name>ToATTACK</T_Name>
        <T_Origin>DEFENSE</T_Origin>
        <T_Destination>ATTACK</T_Destination>
        <T_Action>NULL</T_Action>
        <Events>
          <Event>
            <ID>EVENT_ATTACK</ID>
            <Type>BASIC</Type>
          </Event>
        </Events>
      </Transition>
      <Transition>
        <T_Name>ToDEFENSE</T_Name>
        <T_Origin>ATTACK</T_Origin>
        <T_Destination>DEFENSE</T_Destination>
        <T_Action>NULL</T_Action>
        <Events>
          <Event>
            <ID>EVENT_DEFENSE</ID>
            <Type>BASIC</Type>
          </Event>
        </Events>
      </Transition>
    </Transitions>
  </Fsm>
```

Llamadas internas para crear el autómata

```
//Creamos un objeto relativo a un autómata clásico
FA_Classic fsm;
//Inicializamos la fsm (ID, ID numérico, Callbackmethod, Probabilistic_flag)
fsm = new FA_Classic ("Maquina_A", Tags.StringToTag("CLASSIC"), "CheckEvents", false);
//Creamos un objeto State para la máquina Clásica
State aux;
aux = new State("ATTACK",                          /* State ID          */
                "YES",                             /* Initial flag      */
                Tags.StringToTag("ATTACK"),        /* ID tag            */
                Tags.StringToTag("AttackWithSword"),/* State Action     */
                Tags.StringToTag("NULL"),          /* State IN Action   */
                Tags.StringToTag("NULL"));         /* State OUT Action  */
//Añadimos el estado a la fsm
fsm.addState(aux);
//Rellenamos aux con otro estado nuevo
aux = new State("DEFENSE",                         /* State ID          */
                "NO",                              /* Initial flag      */
                Tags.StringToTag("DEFENSE"),       /* ID tag            */
                Tags.StringToTag("DefenseWithShield"),/* State Action   */
                Tags.StringToTag("NULL"),          /* State IN Action   */
                Tags.StringToTag("NULL"));         /* State OUT Action  */
//Añadimos el estado a la fsm
fsm.addState(aux);

//Creamos un objeto Transition para la máquina Clásica
Transition trans;
//Creamos el State origen (A) y State (B) destino para la transición
State A, B;
//Creamos un objeto List<FSM_Events> que cargará los eventos de la transición
List<FSM_Event>EventsList = new List<FSM_Event>();
A = fsm.getStateByID("DEFENSE");
B = fsm.getStateByID("ATTACK");

//Cargamos el único evento que activa la transición
EventsList.Add(new FSM_Event("EVENT_ATTACK",
                             Tags.StringToTag("EVENT_ATTACK"),
                             "BASIC"));
//Creamos una nueva transición
trans = new Transition ("ToATTACK",                /* Transition ID     */
                        A,                         /* State origin      */
                        B,                         /* State destination */
                        Tags.StringToTag("ToATTACK"),/* Transition ID tag */
                        Tags.StringToTag("NULL"),  /* Transition Action */
                        EventsList);               /* List of events    */

//Añadimos la transición al estado origen
A.addTransition(trans);
//Añadimos la transición a la fsm
fsm.addTransition(trans);

//Cargamos el único evento que activa la transición
EventsList.Add(new FSM_Event("EVENT_DEFENSE",
                             Tags.StringToTag("EVENT_DEFENSE"),
                             "BASIC"));
//Limpiamos la lista de eventos de la anterior transición
EventsList.Clear();
//Rellenamos trans con una nueva transición
trans = new Transition ("ToATTACK",                /* Transition ID     */
                        B,                         /* State origin      */
                        A,                         /* State destination */
                        Tags.StringToTag("ToATTACK"),/* Transition ID tag */
                        Tags.StringToTag("NULL"),  /* Transition Action */
                        EventsList);               /* List of events    */

//Añadimos la transición al estado origen
B.addTransition(trans);
//Añadimos la transición a la fsm
fsm.addTransition(trans);
//Lanzamos el Start de la fsm para que realice operaciones necesarias
fsm.Start();
```

(No se ha realizado control de errores)

4. **Adición del autómata al objeto FSM_Manager creado**
   Hasta este punto hemos creado el autómata pero no lo hemos añadido al FSM_Manager creado. Hasta que no realicemos este paso, no será posible solicitar la FSM correspondiente a nuestro autómata.

   Para ello invocamos al método de adición correspondiente de nuestro FSM_Manager:

```
//fsm_manager creado antes, añadimos la fsm creada de la tabla anterior
fsm_manager.addFSM(fsm);
```

1.3 Jerarquía de máquinas

Se recomienda leer la sección 2.3 "Creación de un estado Jerárquico" del punto 2 "Creación y uso de Estados".

1.4 Definir una FSM con base determinista o probabilista

Aunque se pueden crear diferentes autómatas (FA_Classic, FA_Inertial, FA_Stack y FA_Concurrent_States), todos heredan de la clase origen (el primero, FA_Classic), de forma que es posible crear un autómata cuya base sea determinista o probabilista (indeterminista).

Esto se consigue con la activación de un flag, una simple variable booleana. Nuevamente la definición de esta base determinista o probabilista dependerá de si estamos utilizando la herramienta FSM_Parser o no.

1.4.1 Base determinista o probabilista desde cero

Cuando creamos un autómata desde cero, basta con definir a verdadero o falso el *Probabilistic flag* de su constructor. En el ejemplo anterior:

```
FA_Classic ("Maquina_A", Tags.StringToTag("CLASSIC"), "CheckEvents", false);
```

*Definíamos a **false** este atributo porque queríamos una base completamente determinista.*

*En caso de haber requerido una base indeterminista (con probabilidades de activación en las transiciones) lo hubiéramos definido como **true**.*

Si lo activamos a true, las posteriores transiciones a incluir en la máquina deberán contener probabilidad para que el recorrido de la FSM guarde coherencia con lo indicado.

De no ser así, aparecerán excepciones del tipo *NullReferenceException*.

1.4.2 Base determinista o probabilista con FSM_Parser

La herramienta FSM_Parser determina, según lo que detecte en el xml si la máquina a la que hace referencia es determinista o probabilista.

Esta detección depende de que definamos a ***YES*** o ***NO*** el atributo ***Probabilistic*** de la etiqueta **<FSMtype>** en el documento de diseño xml.

En el ejemplo anterior:

```
<FSMtype Probabilistic="NO">CLASSIC</FSMtype>
```


*Como se puede observar, hemos definido como **NO** el atributo **Probabilistic**, aunque nuestra máquina sigue siendo de su tipo (en este caso CLASSIC, aunque podría ser cualquier otro implementado).*

*Para definir la FSM como probabilista, únicamente habría que cambiar ese **NO** por un **YES**.*

1.5 Consejos y sugerencias

Lo visto anteriormente es un simple ejemplo para ver los pasos a seguir a la hora de crear una FSM desde fichero o "a mano". En este punto, ya se han creado estados y transiciones básicas, pero esto podrá variar dependiendo de que FSM pretendamos construir, se recomienda consultar secciones posteriores para conocer la inserción de Estados y Transiciones adecuadas a cada tipo de máquina a cargar.

A continuación se indican una serie de consejos que ayudan a que el proceso de creación sea correcto:

1. **Utilizar la herramienta FSM_Parser**. Es muy cómoda y conseguiremos sortear un buen número de errores.

2. **Dependiendo de qué autómata estemos creando** (si nos decantamos por la opción de carga a mano), **necesitaremos un constructor de objeto State u otro, además de un constructor de objeto Transition que también sea adecuado.** **Ver secciones (Creación y uso de Estados y Creación y uso de Transiciones)**.

3. **Acostumbrarse al uso de la clase Tags.** Es importante concentrar todas las posibles etiquetas y tenerlas accesibles en cualquier momento desde programa; de esta forma evitaremos tener que almacenar, recorrer y comparar estructuras de objetos que ralentizarán nuestra aplicación.

4. **Si partimos de cero (no usamos FSM_Parser), es recomendable crear primero el autómata, después los estados y añadirlos, y posteriormente las transiciones** (habiendo definido la lista de eventos para cada una de ellas), **por ese orden**. Es el esquema de creación que se ha utilizado al implementar la herramienta FSM_Parser y se puede garantizar que funciona.

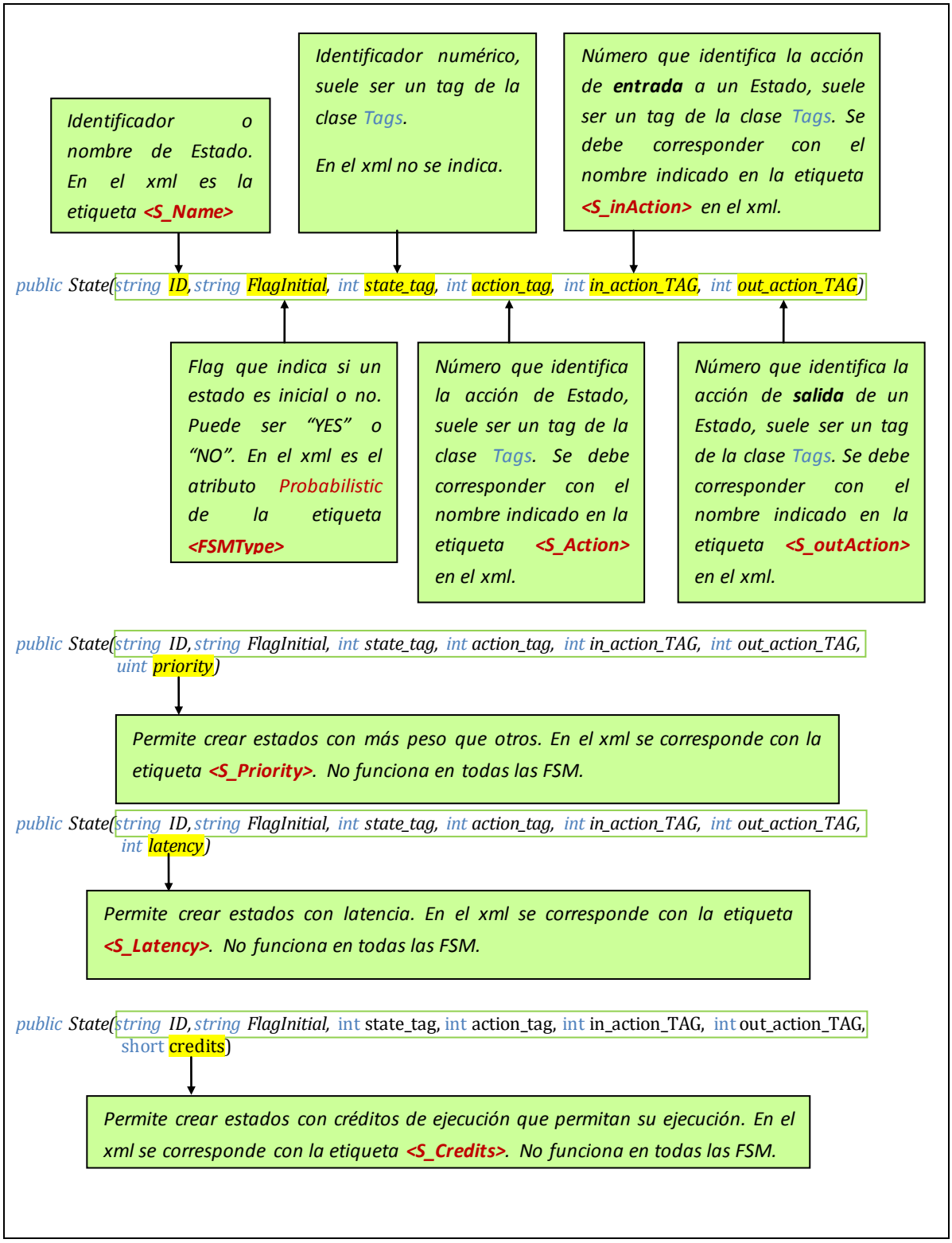
2. Creación y uso de Estados

Aunque que se ha visto ya el proceso de creación de un autómata, el cual implica la creación de Estados, es necesario comentar todas las posibilidades que se ofrecen a la hora de crear nodos/estados de un grafo, además de todas las características que le dan forma.

Un estado es una entidad software que guarda las referencias a la lógica/codificación de otras entidades software usuarias de una máquina de estados finitos.

2.1 Constructores de un estado

A continuación se muestran todos los constructores relativos a un estado y se explican al detalle.



2.2 Creación y uso de un estado

La creación y uso de un estado varía según si se utiliza la herramienta FSM_Parser o no. A continuación se explica cada caso.

2.2.1 Creación y uso de un Estado de cero

Para conocer qué constructor de la clase State se debe utilizar según el autómata a crear se muestra un cuadro que clarifica la correspondencia State-FA:

Tipo *Finite Automata*	Constructor de la clase *State* a emplear
Classic	*public State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG)*
Inertial	*public State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG, int latency)*
Stack	*public State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG, uint priority)*
Concurrent_States	*public State(string ID, string FlagInitial,* int state_tag, int action_tag, int in_action_TAG, int out_action_TAG, short credits)

Utilizar un constructor de State incorrecto para una máquina que requiere de algún parámetro del constructor correcto puede acarrear errores de ejecución de la herramienta.

Si no añadimos, por ejemplo, latencia a un estado perteneciente al autómata inercial, es probable que nos encontremos con errores de referencia nula al parámetro solicitado (*NullReferenceException*), o que la API asocie valores por defecto para evitar estas excepciones.

2.2.2 Creación y uso de un estado con FSM_Parser

La creación de un estado con el parser **se realiza automáticamente**. En este caso es **necesario utilizar la clase Tags** para definir las etiquetas que identifican numéricamente a un estado.
Así pues, añadiríamos la constante entera correspondiente en la clase Tags y el correspondiente *matching* en el método de dicha clase *StringToTag(…)*.

Para observar cómo realizar esta tarea, ver la sección correspondiente a la creación de un tag.

2.3 Crear un estado jerárquico

Aunque se ha hablado de jerarquía de máquinas, esta anidación no se produce entre autómatas o máquinas en realidad. **Una submáquina no se asocia a una máquina, sino a un estado de ésta**.

La creación de un estado jerárquico depende nuevamente de si estamos utilizando la herramienta FSM_Parser o no. A continuación se explica cómo crear un estado jerárquico desde ambos enfoques.

2.3.1 Crear un estado jerárquico de cero

Para asociar una submáquina a un estado sin utilizar la herramienta de *parsing* lo único que debemos hacer es utilizar el método de adición que proporciona la clase State:

```
public void addFA(FA_Classic  subFA)
```

Este método asocia un objeto FA_Classic al State que haga la invocación. Internamente, si este objeto es diferente del valor nulo, la herramienta contempla su gestión (la gestión de la submáquina) y no la del estado superior.

Veamos un ejemplo:

```
/** (1) Código de creación de la máquina fsm omitido**/
//(2) Creación de la submáquina
FA_Classic subFSM = new(...) ;
//(3) Creación del estado
State aux;
aux = new State("DEFENSE",                    /* State ID          */
                "NO",                         /* Initial flag      */
                Tags.StringToTag("DEFENSE"),  /* ID tag            */
                Tags.StringToTag("NULL"),     /* State Action      */
                Tags.StringToTag("NULL"),     /* State IN Action   */
                Tags.StringToTag("NULL"));    /* State OUT Action  */

//(4) Añadimos una submáquina a este estado
aux.addFA(subFSM);

//(5) Añadimos el estado a la fsm
fsm.addState(aux);
```

Siguiendo con el ejemplo de secciones anteriores, imaginemos que tenemos una máquina llamada *fsm* **(1)**, y queremos añadirle un estado **(3)** que a su vez contiene una submáquina llamada *subFSM* **(2)**. Como se puede observar, primero creamos ambas máquinas, y añadimos la submáquina al estado **(4)**. Posteriormente añadimos el estado a la máquina superior *fsm* **(5)**.

2.3.2 Crear un estado jerárquico con FSM_Parser

La creación de un estado jerárquico utilizando FSM_Parser como herramienta de *parsing* es automática siempre que en el documento de diseño de la FSM (xml) exista un enlace a otra máquina indicado en la etiqueta perteneciente a un estado **<S_Fsm></S_Fsm>.**

Veamos un ejemplo:

```
<State Initial="NO">
        <S_Name>DEFENSE</S_Name>
        <S_Action>NULL</S_Action>
        <S_inAction>NULL</S_inAction>
        <S_outAction>NULL</S_outAction>
        <S_Fsm><!-- Link a la FSM (a partir del raíz del proyecto)--></S_Fsm>
</State>
```

La etiqueta mencionada debe contener un **enlace** a la FSM a cargar desde el raíz del proyecto, por ejemplo:

```
<S_Fsm>/FSM/Hierarchical/Attacking.xml</S_Fsm>
```

La aplicación buscará la carpeta raíz del proyecto con ***System.IO.Directory.GetCurrentDirectory()*** y lo concatenará con lo indicado entre las etiquetas.

Donde ***Attacking.xml*** es el nombre del documento xml que contiene el diseño de otra FSM.

Si un estado no es jerárquico, **esta etiqueta debe aparecer** pero no contener información.

2.4 Consejos y sugerencias

Se ha comentado en secciones anteriores las formas adecuadas de añadir un estado.

Se ofrecen una serie de consejos y sugerencias para hacer más sencilla la inserción de un estado:

1. Nuevamente, **se recomienda el uso de la herramienta FSM_Parser**. La creación de un estado es automática y rápida.

2. Con cada estado nuevo creado, **crear el Tag y el matching correspondiente en la clase Tags**. Es muy cómodo tener control sobre los tags/tokens asociados a cada entidad perteneciente a una FSM o autómata (ver sección Cómo crear un Tag).

3. **No asociar acción alguna a un estado jerárquico**. Aunque es posible, no es recomendable, pues si la submáquina se encuentra ociosa, podrían ejecutarse acciones del estado de la FSM superior y dar un resultado incongruente. Se ha diseñado pensando en que cuando una submáquina se activa, adquiere y tiene el **control total** de la gestión del estado al que pertenece.

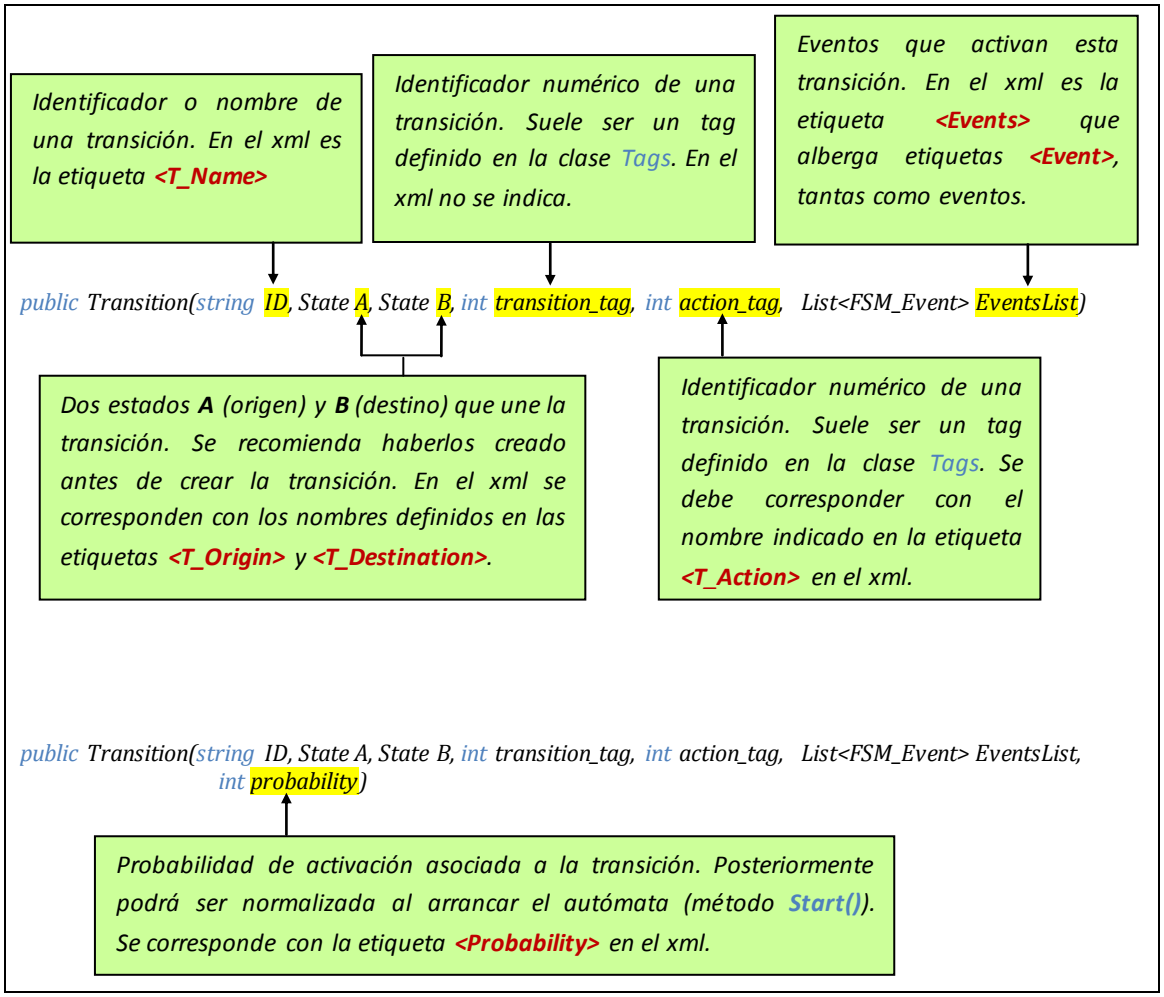
3. Creación y uso de Transiciones

Ya se ha observado la creación de transiciones en la sección en la que se creaba un autómata. De todas formas, es necesario conocer con profundidad este tipo de entidad base de una máquina de estados finitos.

En tiempo de ejecución se producen eventos y se envían a la máquina de estados pertinente. Una vez llegan, pueden activar enlaces o caminos sensibles a ellos y pueden producir un cambio en la situación actual de la máquina. Estos enlaces sensibles a los eventos que llegan son las Transiciones.

3.1 Constructores de una Transición

A continuación se muestran los constructores disponibles para crear una Transición.



Básicamente, existen dos tipos de transiciones: las que se crean para una FSM cuya base no es probabilista y las que se crean para las que cuya base sí lo es.

Dependiendo de si nuestra máquina de estados tiene activo el *flag* de probabilidad (ver definición de un autómata) será necesario un constructor de Transition u otro.

3.2 Creación y uso de un Transición

Según si partimos de cero, o si usamos el FSM_Parser, crearemos y/o asociaremos una Transición a una máquina de estados finitos de una manera u otra.

3.2.1 Creación y uso de una Transición de cero

Para conocer qué constructor de la clase Transition se debe utilizar según el autómata a crear se muestra un cuadro que clarifica la correspondencia Transition-FA:

Tipo *FiniteAutomata*	Flag Probabilistic	Constructor de la clase Transition a emplear
Classic ***Inertial*** ***Stack*** ***Concurrent_States***	'YES'/**True**	*public Transition(string ID, State A, State B,* *int transition_tag, int action_tag,* *List<FSM_Event> EventsList, int probability)*
	'NO'/**False**	*public Transition(string ID, State A, State B,* *int transition_tag, int action_tag,* *List<FSM_Event> EventsList, int)*

Únicamente se debe tener en cuenta si el *Flag* de probabilidad **es YES (True)** o **NO (False)**.

Como se ha comentado en otros casos, el uso de un constructor incoherente de acuerdo con el *Flag* de probabilidad podría acarrear errores de ejecución de referencia nula (*NullReferenceException*), por ejemplo, al solicitar una probabilidad que no ha sido añadida.

3.2.2 Creación y uso de una Transición con FSM_Parser

La creación de una transición con el parser **se realiza automáticamente**. En este caso es **necesario utilizar la clase Tags** para definir las etiquetas que identifican numéricamente a una transición.

Así pues, añadiríamos la constante entera correspondiente en la clase Tags y el correspondiente *matching* en el método de dicha clase *StringToTag(…)*.

Para observar cómo realizar esta tarea, ver la sección correspondiente a la creación de un tag.

3.3 Consejos y sugerencias

Los consejos se parecen mucho a los sugeridos en la sección de consejos y sugerencias de los estados. Realmente guardan parecido; son entidades base de un autómata finito.

1. Como antes, **se recomienda el uso de la herramienta FSM_Parser**. La creación de una transición es automática y rápida.

2. Con cada transición nueva creada, **crear el Tag y el matching correspondiente en la clase Tags**. Es muy cómodo tener control sobre los tags/tokens asociados a cada entidad perteneciente a una FSM o autómata (ver sección Crear y utilizar un Tag).

3. **Las transiciones, al igual que los estados, disponen de varios constructores**. Hay que crear la transición adecuada según la FSM (se ha explicado ya).

4. Creación y uso de Eventos

Se expone a continuación el uso de los eventos en la fase de carga, además del uso de éstos por programa, explicando las diferencias entre ellos.

4.1 Creación y uso de los eventos

El usuario solamente debe generar FSM_Events cuando quiera crear manualmente los autómatas (si utiliza el parser, ni siquiera tiene que usar FSM_Events, pues ya se generan internamente). Una vez cargadas no tiene sentido que lo haga (no debería). Es decir, **FSM_Events se utiliza cuando se está creando un autómata y no cuando ya está cargado**. Una vez cargado, las FSM_Machine necesitan Tags de Eventos que hagan matching con los tokens identificativos de cada FSM_Event que poseen sus transiciones.

Por tanto, es necesario diferenciar entre dos conceptos que en alguna ocasión se hayan podido tratar únicamente como uno para evitar confusiones:

- **Objetos *FSM_Event***
  → Objetos generados al cargar un autómata
- **Eventos (tags numéricos)**
  → Como resultado de condiciones que se cumplen en la partida/ciclo de ejecución.

4.1.1 Eventos en la carga de autómatas (FSM_Event)

Cada vez que añadimos una nueva transición, es necesario determinar una lista de eventos a la que es sensible. Es decir, **es necesario pasar como parámetro de entrada al constructor de una Transition una lista de elementos FSM_Event**, cada uno con con un nombre de evento y un tipo.

Así pues, se hace lo siguiente:

```
//Lista de elementos de tipo FSM_Event para UNA transición
List<FSM_Event> Lista_Eventos = new List<FSM_Event>();
//Evento con nombre, token y tipo
FSM_Event ev = new FSM_Event(nombre, token, tipo);*
//Se añade el evento a la lista creada
Lista_Eventos.Add(ev);
//Se crea la transición y se le pasa la lista creada como parámetro
Transition t = new Transition(*Otros Parámetros* + Lista_Eventos);
```

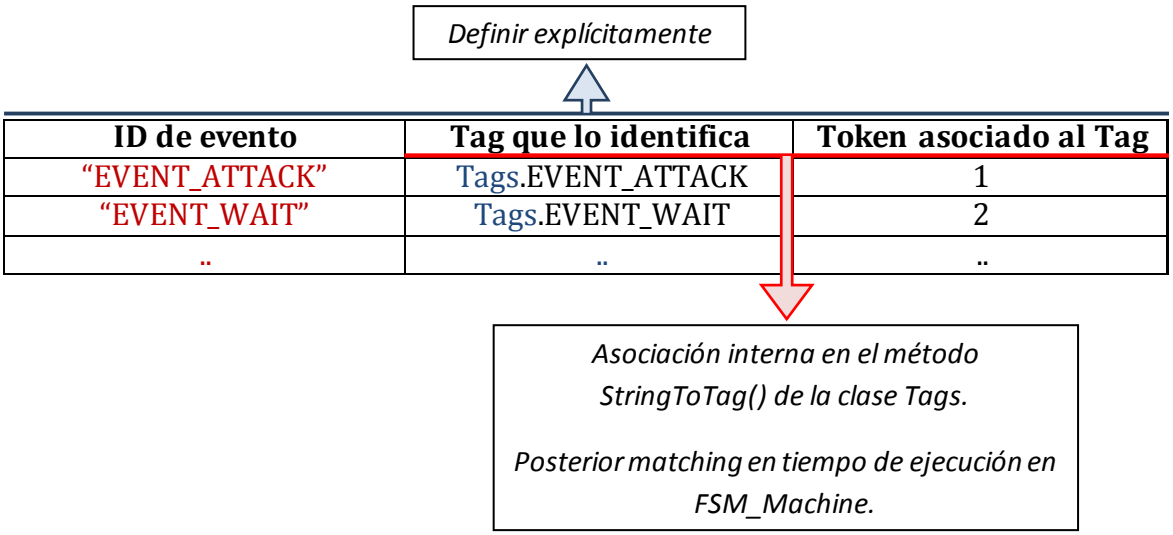
*Cuando creamos el FSM_Event, son necesarios un nombre y un tipo ya comentados, pero **además, necesitamos un token**, un número identificador completamente necesario y por tanto obligatorio (es necesario indicar el nuevo evento en la clase **Tags**).

Este token es un parámetro muy importante, pues es con el que se hará *matching* posteriormente para identificar el evento internamente.

4.1.2 Eventos por programa (Tags de eventos)

Realmente, una entidad sujeta a una FSM no genera eventos, sino Tags de Eventos. Esto puede llevar a confusión porque dicha entidad no genera FSM_Events, únicamente genera números (Tags definidos explícitamente) que internamente podrán hacer matching con el token identificativo de cada FSM_Event añadido en la carga a una transición.

Así como en los eventos en la carga, creábamos FSM_Events, los Tags de Eventos son simples números de tipo int definidos explícitamente por el diseñador en la carga, que posteriormente invoca el programador cuando una entidad hace uso de sus FSM. Se recomienda utilizar una clase Tags que albergue constantes de tipo int que identifiquen a un evento. La idea es la siguiente:



El ID de evento es una cadena que identifica un FSM_Event con un String. Será necesario que el usuario defina una serie de Tags asociados a esas cadenas explícitamente en la clase Tags. Finalmente, tendrá que rellenar el método StringToTag añadiendo los *matchings* entre cadenas y tags, de forma que esto retorne Tokens. Por pasos:

(1) Cadenas en la carga (XML o carga manual)

XML: <Event><ID>EVENT_ATTACK</ID>...</Event>	**Manual:** new FSM_Event("EVENT_ATTACK", etc);
XML: <Event><ID>EVENT_WAIT</ID>...</Event>	**Manual:** new FSM_Event("EVENT_WAIT", etc);
XML: <Event><ID>EVENT_DIE</ID>...</Event>	**Manual:** new FSM_Event("EVENT_DIE", etc):
...	...

(2) Definir explícitamente los tags en la clase Tags

```
public class Tags{
    public const int EVENT_ATTACK  = 0;
    public const int EVENT_DIE     = 1;
    public const int EVENT_WAIT    = 2;
    ...
}
```

(3) Rellenar StringToTag en la clase Tags con un matching

```
Case "E":
    if(word.Equals("EVENT_ATTACK")) return Tags.EVENT_ATTACK;
    if(word.Equals("EVENT_DIE"))    return Tags.EVENT_DIE;
    if(word.Equals("EVENT_WAIT"))   return Tags.EVENT_WAIT;
```

- Se especifican las cadenas que identifican a un evento.
- Se definen los *tags* en la clase *Tags* explícitamente, asociándoles un *token*.
- Se hace el *matching* en el método *StringToTag* de la clase *Tags*.

Esto implica que crear un FSM_Event en la carga sea algo como:

```
FSM_Event ev = new FSM_Event("EVENT_ATTACK", $tring2Tag("EVENT_ATTACK"), "BASIC");
```

ID — Token — Tipo

Por programa, cuando en la lógica de una entidad sujeta a un FSM se dé una condición, se generá un Tag de Evento que pasará a la FSM automáticamente:

(Siguiendo con el ejemplo de carga del principio de la guía, esta lógica debería ir en un método CheckEvents)

```
//Si hay cerca un enemigo añadimos el evento de ataque a la lista de eventos
if(EnemigoCerca){
        Lista_eventos.Add(Tags.EVENT_ATTACK);
}
//Se devuelve la lista de eventos que será la entrada (implícita) a la FSM
return Lista_eventos;
```

Tag de Evento de ataque

Un ejemplo extraído de una de las demostraciones, es que uno de los enemigos detecta al personaje cerca de él; automáticamente, el susodicho enemigo genera un tag de evento de ataque (Tags.EVENT_ATTACK) que probablemente activará

alguna transición de salida de su estado actual (al hacer *matching*) y constituirá algún cambio en su máquina asociada*, de forma que ésta le proporcione el *feedback* necesario para que pueda realizar las acciones correspondientes.

Esto dependerá de la lógica/tipo de su FSM.

4.2 Tipo de FSM_Event

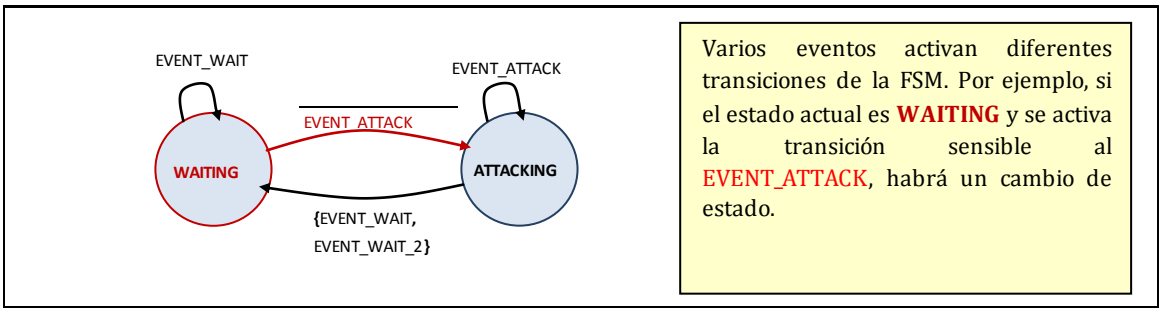
Por último, el tipo del evento, que será de una clase u otra dependiendo de la lógica que necesitemos para nuestra entidad usuaria de una FSM.

A continuación, se especifican los dos tipos implementados, con sus peculiaridades y formas de uso.

4.2.1 FSM_Event de tipo básico (BASIC)

Es el tipo por defecto y que más se utiliza. Este tipo de evento **activa una transición sensible a él** siempre que dicha transición sea alcanzable desde el estado actual de la máquina, y **no tiene ningún otro efecto sobre la FSM**.

Un ejemplo gráfico:



En el ejemplo vemos que la transición que sale de ATTACKING a WAITING es sensible a dos eventos. Esto es perfectamente posible en una FSM; de esta forma una transición puede contener una lista de eventos que pueden activarla.

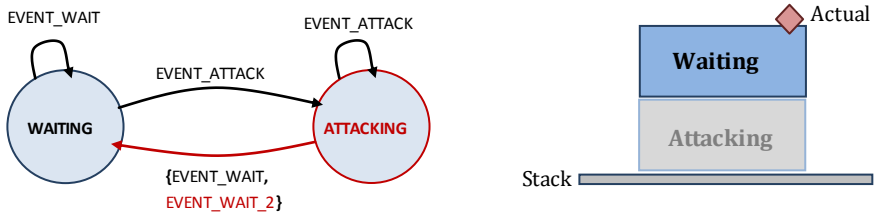
4.2.2 FSM_Event de tipo apilable (STACKABLE)

Este tipo de evento únicamente **es válido en el diseño de una máquina basada en pilas**. Para cualquier otra FSM, este evento es de tipo básico a efectos prácticos aunque se defina como apilable.

En una FSM basada en pilas, cuando un estado activo recibe un evento de tipo apilable y se activa una transición de salida, se apilará en una estructura de datos de forma que su gestión quede interrumpida. Se transitará a un estado que se ejecutará hasta que se dé un ciclo en el cual la máquina se encuentre en

estado ocioso; cuando esto suceda, para evitar la inactividad (o la actividad por defecto), se volverá al estado anterior apilado.

A priori, **será necesario que el estado destino sea más prioritario que el estado a apilar**. De no ser así, no se transitará.



Si el estado **ATTACKING** es el estado activo y se da el evento EVENT_WAIT_2 de tipo ***Stackable***, la lógica a seguir por una FSM basada en pilas es la de apilar el estado actual de ataque y sustituir su ejecución por la del estado destino WAITING. Cuando la ejecución de WAITING finalice, se retornará al estado anterior **ATTACKING**, siempre que no llegue un evento que afecte al primero.

4.3 Consejos y sugerencias

A continuación se listan una serie de consejos y sugerencias para evitar problemas a la hora de crear y utilizar eventos:

- **Es recomendable conocer previamente la diferencia entre los FSM_Events y los eventos que se disparan por programa (tags de eventos)**. La guía es casi de lectura obligatoria.
- **No utilizar tipos de FSM_Event que no funcionen en otras máquinas**. El efecto puede conllevar algún error o directamente no funcionar como se preveía.
- **Como siempre, utilizar la clase Tags para tener localizados todos los tokens de eventos.**

5. Acciones

Una acción es la única entidad de una máquina de estados finitos que no es un objeto como tal de ésta. En realidad una acción es una mera referencia a una codificación que ejecuta una tarea relativa a un comportamiento del personaje.

Esto quiere decir que realmente no se crean acciones como objetos, a una máquina de estados finitos se le añaden referencias a métodos que posee el NPC que tienen que ver únicamente con su comportamiento.

5.1 Adición de acciones a una Máquina de Estados Finitos

Por ejemplo, si una pelota (entendiendo esta pelota como un NPC sensible a una FSM) debe saltar, deberá entrar en un estado de 'Salto' y este estado deberá tener un tag que referencie al método saltar contenido en el objeto pelota.

En los constructores de State y Transition encontramos parámetros de entrada de tipo int que son tags de acciones:

```
public Transition(string ID, State A, State B, int transition_tag, int action_tag, List<FSM_Event> EventsList)

public State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG)
```

Estos Tags, como se ha comentado, son variables enteras, las cuales se recomienda especificar en la clase **Tags**, en forma de constante entera, además de en su método StringToTag, de manera que luego puedan ser referenciadas rápidamente desde el programa.

Hay dos posibilidades: o bien identificamos estos tags a mano con cuidado de no repetir números, o bien las identificamos en la clase tags. Los constructores State y Transition quedarían:

```
public Transition(string ID, State A, State B, int transition_tag, 5, List<FSM_Event> EventsList)

public State(string ID, string FlagInitial, int state_tag, 1, 2, 3)
```

En caso de definirlos a mano.

O bien:

```
public Transition(string ID, State A, State B, int transition_tag, StringToTag("A_SALTO"), List<FSM_Event> EventsList)

public State(string ID, string FlagInitial, int state_tag, StringToTag("SALTAR"), StringToTag("ENTRADA"), StringToTag("SALIDA"))
```

En caso de definirlos como Tags, en la clase **Tags**.

Donde imaginemos que esa Transition que realiza la animación para pasar hacia el estado de Salto en el que en su entrada puede realizarse alguna acción de entrada al estado, seguidamente si se permanece en dicho estado se salta y si hay un cambio de estado, puede realizarse alguna acción de salida.

5.2 Acciones como salida de una Máquina de Estados Finitos

Una máquina de estados finitos devuelve referencias (las referencias que habíamos añadido en la carga (Ver sección anterior 5.1) a métodos a la entidad a la que está sujeta.

Así pues, una FSM recibe una lista de tags de eventos (*que contiene Tags.ATTACK* cuyo token es el 3, p.e) y **devuelve una lista de acciones** de tipo int acorde a esa lista de eventos recibidos.

Imaginemos que un NPC monstruo envía una lista que contiene un solo evento Tags.ATTACK. El monstruo, que ya se encontraba en el estado correspondiente a ese evento que ha generado, realiza el bucle de estado para quedarse como está.

Y finalmente retorna la acción del bucle y la acción de permanencia en el estado.

Estas dos acciones (referencias) son retornadas al NPC, el cual ejecuta el método correspondiente a la acción y actualiza su animación.

5.3 Consejos y sugerencias

A continuación se listan una serie de consejos que pueden venir bien para entender y utilizar correctamente las acciones.

- **Entender una acción como una referencia a un método**. Una acción no es un método, sino una referencia a éste.

- **Una acción no es un objeto de una FSM**. Es únicamente una referencia, un número.

- **No asociar acciones a Estados que estén controlados por una submáquina de estados finitos**. Si la submáquina queda ociosa pueden darse comportamientos de una máquina superior y una inferior simultáneamente lo cual, si no se tiene cuidado, puede ser incongruente.

- Como siempre, **el uso de la clase Tags para tener todas las acciones claramente referenciadas (y diferenciadas) mediante tokens**.

6. Creación y uso de la clase Tags

Se ha comentado a lo largo de toda esta guía, además de en la propia memoria del proyecto, el uso de una clase Tags que sirviera para tener concetrados una serie de etiquetas o tags identificados con un número, que sirvieran para referenciar cada entidad sujeta a una máquina.

A continuación se explica la creación y uso de un Tag.

6.1 Crear y utilizar un Tag

Hay una serie de Tags fijos definidos que no pueden cambiarse ni ser sustituidos, son los tags relativos a los tipos de autómatas implementados (CLASSIC, INERTIAL, STACK_BASED y CONCURRENT_STATES).

Aparte de estos tags comentados, los demás se pueden añadir según lo que requiera el diseño de cada máquina.

Es decir, si construimos una máquina con un estado llamado ATAQUE, lo lógico es que haya un Tag que identifique dicho estado ATAQUE.

Aunque ya hemos visto cómo construir la clase tags en la sección 1.1 de esta guía, **veamos cómo construir un Tag para un estado de ataque en varios pasos**:

1. Vamos a la clase **Tags** y creamos la constante entera relativa al nuevo estado de ataque creado.

```
public class Tags{

        public const int ATTACK = 1;

        /*... */
        //código omitido

}
```

2. Creamos el *matching* entre el tag creado y el string que identifica al estado en el método StringToTag de la clase **Tags**.

```
public int StringToTag(string word){

        switch (word[0]) {
        case 'A':
            if(word.Equals("ATTACK")) return Tags.ATTACK;
        break;
        /*Código omitido*/
```

```
}
```

Rellenar el método StringToTag no es estrictamente necesario si no necesitamos convertir cadenas o identificadores en los correspondientes tags, aunque sí recomendable.

Si ahora creamos un objeto State y le pasamos un 1 como parámetro en el state_tag del constructor, lo estaremos referenciando como el estado de ataque. Personalmente considero más cómodo crear el tag y el matching en StringToTag para luego crear el State, siendo el parámetro state_tag una llamada al StringToTag, tal que así:

```
public State(string ID, string FlagInitial, StringToTag(ID), ...)
```

Hemos visto un ejemplo para un Estado, **pero las Transiciones, Eventos y acciones de una FSM, en caso de usar la clase Tags, deben ser referenciadas de la misma manera.**

Para utilizar un tag, o adquirir su valor, únicamente tenemos que utilizar su etiqueta ya previamente definida:

```
int accion = Tags.Action_jump;
```

Los únicos tags que tiene sentido utilizar explícitamente en un posible videojuego son los relativos a los eventos y a las acciones. Aunque **es necesario identificar cada entidad que forma una FSM** (*States*, *Transtions*, *Events* y *Actions*).

6.2 Consejos y sugerencias

Para crear y utilizar un Tag se listan una serie de consejos y sugerencias.

- **Otorgar un número diferente a cada tag**. Si damos un número diferente por cada tag nos evitaremos errores de *tokens* repetidos que son difíciles de detectar.
    - Si no otorgamos un número diferente a cada tag, **al menos se debe otorgar un número diferente a cada tag que pertenezca al mismo tipo de entidad de una FSM**. Por ejemplo, un estado no puede tener el mismo

tag identificativo que otro, pero no hay problema si un estado tiene el mismo identificador que una transición, pues son entidades diferentes.

- **Crear un tag y un matching** en el método *StringToTag* de la clase Tags **para cada inserción de una nueva entidad** (un nuevo estado, una nueva transición, un nuevo evento o una nueva acción). De esta forma lo cogemos como hábito y es menos probable equivocarse.

- **Definir un tag desconocido que sirva de comodín.** *StringToTag* deberá retornar el tag desconocido cuando NO haya matching posible en la conversión de cadenas. (Tags.UNKNOWN a priori ya añadido en la plantilla proporcionada).

- Por última vez, **utilizar esta clase y la herramienta FSM_Parser**. Facilita mucho la tarea de creación de máquinas de estados finitos, pues el parser utiliza automáticamente la clase Tags (invoca el método *StringToTag* y obtiene cada identificador numérico) para convertir cadenas en números.

CAPÍTULO 2: PROCESO DE USO/RECORRIDO DE UNA MÁQUINA

Visto el proceso de carga de una Máquina de Estados Finitos, es necesario conocer cómo asignarla a una entidad (un NPC en caso de un videojuego) y cómo utilizarla.

A continuación se muestran los puntos relativos a este proceso.

1. Solicitar una Máquina de Estados Finitos

Solicitar una máquina de estados es un proceso muy sencillo. A continuación veremos cómo solicitar una máquina a una entidad llamada **FSM_Manager** que forma parte del núcleo de la API y que se encarga de la carga y asignación de máquinas.

Ya se ha mostrado cómo crear el FSM_Manager y cómo utilizarlo para cargar una máquina (Ver secciones 1.1 o 1.2).

1. **Solicitar una FSM**. Es necesario que el objeto que requiera de una máquina de estados finitos tenga una referencia directa al FSM_Manager creado. La capa de control de IA tendrá que pasar como parámetro a la entidad que quiera una máquina una referencia al FSM_Manager.

   La capa de IA podría ser algo así:

```
public class control_IA{
/*Código omitido*/
    public void main(){
        /*Código creación del FSM_Manager omitido*/
        Monster m = new Monster(fsm_manager); //Paso como parámetro
    }
}
```

Imaginemos que Monster internamente hace las llamadas oportunas para solicitar una máquina de estados finitos:

```
public class Monster{
    //Manager de máquinas del Monster
    FSM_Manager fsm_manager;
    //Máquina del Monster
    FSM_Machine FSM;
    //Constructor de la clase Monster al que le llega un FSM_Manager
    public Monster(FSM_Manager fsm_manager){
        this.fsm_manager = fsm_manager; //asociación
    }
    public void Start(){
        this.FSM = this.fsm_manager.CreateMachine(this, Tags.CLASSIC,
        "NombreFSM");

    }
}
```

```
CreateMachine(this,        Tags.CLASSIC,        "NombreFSM");
```

this	Tags.CLASSIC	"NombreFSM"
Objeto que **solicita** la FSM. Por lo general, **this**.	Tag (clase Tags) que se corresponde con la FSM clásica. Es un número entero.	Nombre ID de la FSM. **Debe corresponderse con el ID de alguna FSM cargada** o dará error NullReference.

CreateMachine retorna un objeto FSM_Machine, **el objeto que contendrá la FSM solicitada**. Si la FSM_Machine retornada tiene valor nulo (**null**), querrá decir que no se ha podido crear dicha FSM.

2. Interaccionar con una Máquina de Estados Finitos

Una vez visto el proceso de solicitud de una FSM al objeto *FSM_Manager*, es momento de utilizar el objeto que éste nos retorna, el *FSM_Machine*, que contiene la máquina solicitada.

1. **La entidad actualiza su máquina de estados**. Utiliza el método Update de su máquina asociada.

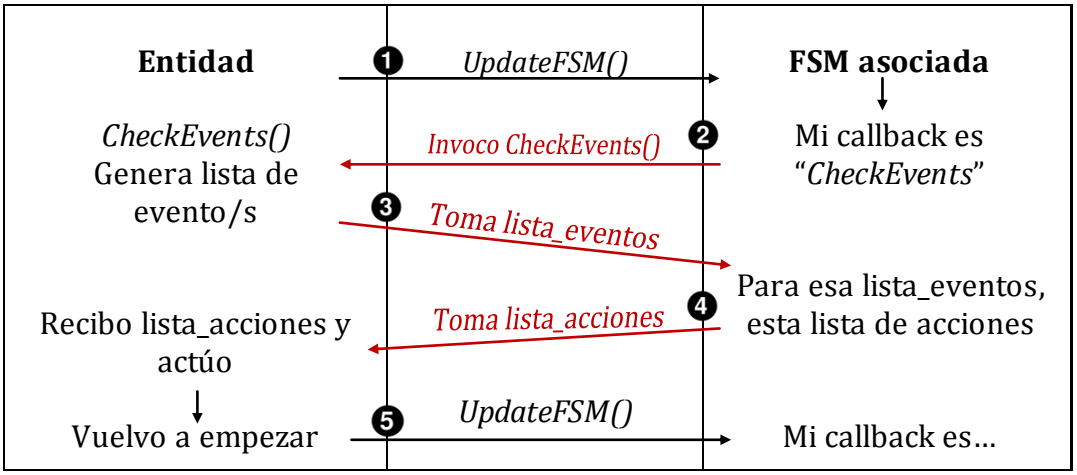
```
DoActions = FSM.UpdateFSM();
```

DoActions es una lista de referencias de tipo int, que indican las acciones que se deben realizar.

Cuando se realiza un UpdateFSM, la FSM_Machine asociada referencia a un método de la propia entidad, un callback o procedimiento que le dice qué eventos debe generar para dicha FSM.

Para el caso anterior de esta guía en el que cargábamos una FSM, la FSM solicitada tiene un *callback* llamado **CheckEvents**. La entidad que solicita la FSM tendrá que tener un método llamado **CheckEvents** que genere eventos que activen dicha FSM.

Sucede un flujo de información Entidad-FSM como la siguiente:



En **color rojo se muestran las invocaciones internas automáticas** que son invisibles para el usuario.

Si existe algún estado con alguna submáquina, esta submáquina tendrá que tener otro procedimiento de eventos asociado al que sea sensible. De igual forma, la entidad que tenga la FSM tendrá que tener implementado dicho procedimiento de eventos. (CheckEvents2, por ejemplo).

2. Estos procedimientos de eventos comentados en el punto anterior tendrán que **generar tags de eventos**. Para ver cómo realizar esta tarea ver sección 4.1.2 de esta guía.

3. El método *UpdateFSM()* retorna una lista de acciones, que no son más que referencias de tipo **int**. Si hemos creado los tags correspondientes a todas las acciones, en la clase Monster (entidad que tiene una FSM asociada y puesta de ejemplo) podremos hacer lo siguiente:

```
foreach (int action in DoActions){
        executeAction(action);
}
```

Y el método *executeAction* podría ser parecido a:

```
public void executeAction (int action) {
        switch (action) {
        case Tags.EVENT_ATTACK:
                BasicAttack();
        break;
        (…)
}
```

Si *action* coincide con algún ***case***, se ejecutará el método asociado.

De ahí la utilidad de la clase ***Tags*** para hacer comparaciones de números enteros muy rápidamente y desde cualquier parte de la aplicación.

3. *Consejos y sugerencias de uso*

A continuación se listan una serie de consejos y sugerencias para que el proceso de uso e interacción con una FSM sea más sencillo.

- Es recomendable **leer el capítulo 1** de esta guía. Muchos de los aspectos que se explican en este capítulo 2 ya se han mencionado previamente.

- **Definir todos los tags que puedan ser utilizados por programa**. Cada evento, cada acción… es muy sencillo y rápido referenciarlas mediante tags para evitar comparaciones de cadenas y hacer comparaciones entre números mucho más eficientes.

- Al crear una FSM con *CreateMachine*, **el objeto FSM_Machine que retorna podría tener valor nulo**. Este hecho indica que no se ha podido crear correctamente.

    - Es recomendable ver si el identificador (nombre de la FSM) indicado en la invocación de *CreateMachine* es el correcto, al igual que el Tag del tipo de máquina (a priori, definido por defecto, pero por si acaso).

Recuerda…

- **La entidad que use una FSM deberá implementar los *callbacks* que afecten a ésta.** De esta manera, cada FSM "exige" un procedimiento que genere una serie de eventos a los que es sensible.

- **La lista de acciones que retorna el método *UpdateFSM* del objeto *FSM_Machine* son simples números enteros.** No hay código interno que se ejecute, estos números sirven para realizar comparaciones rápidas que permitan ejecutar las acciones reales implementadas como métodos.

    - Por ejemplo, un NPC solicita una ***FSM*** a su ***FSM_Manager***. Éste le retorna una ***FSM_Machine*** con la cual el NPC invoca el método ***UpdateFSM*** que retornará una lista de acciones (números). Estos números podrán ser comparados con los ***Tags*** de las acciones definidas explícitamente, de forma que se ejecuten las acciones o métodos reales correspondientes a éstas si hay ***matching*** *(entre las acciones retornadas por la máquina y los tags de acciones).*

4. Ejemplos prácticos para distintas FSM

A continuación se listan una serie de ejemplos interesantes (únicamente a nivel de diseño) para cada tipo de máquina disponible. Es posible que puedan ser creados con otras máquinas o ser construidos siguiendo múltiples vías.

Basándose en la experiencia con la herramienta, se pretende ofrecer una serie de muestras que anticipen los perfiles de máquina de estados finitos que cubriría cada tipo implementado en la API.

4.1 Ejemplos de FSM utilizando FA_Classic

Se muestran un conjunto de ejemplos para la FSM clásica en caso de que sea determinista o indeterminista (probabilista).

4.1.1 Determinista o probabilista

Una máquina puede ser indeterminista si activamos el flag adecuado en su diseño. De no activarse, sería determinista.

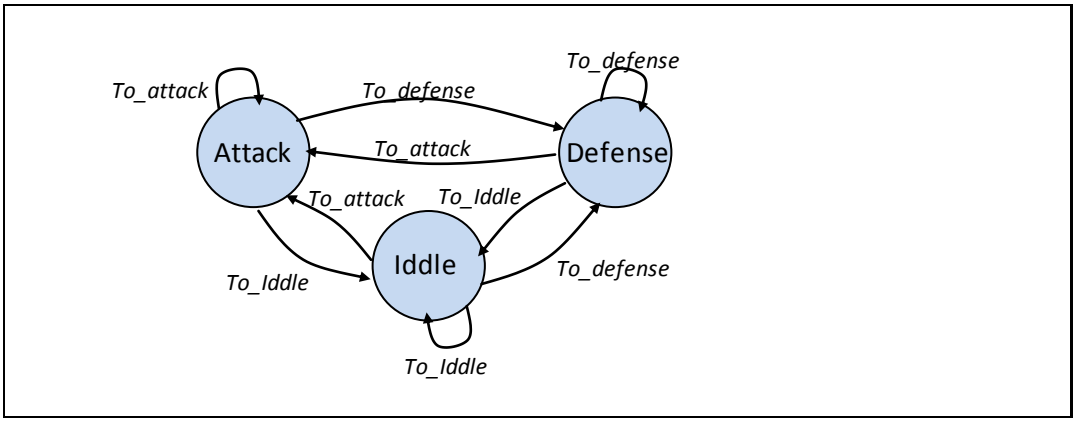
- **FA_Classic (Deterministic)**

Ejemplo 1. Ataque y defensa de un enemigo

Es un ejemplo que refleja con claridad el fin de esta máquina de estados. Un enemigo reacciona a estímulos que le hacen realizar acciones muy diferentes y simples. El ataque y defensa de un NPC puede ser gestionado con una FSM clásica-determinista rápidamente (sin entrar en la implementación de las tareas de atacar y defender).

El caso base sería crear una máquina con tres estados: uno para el estado ocioso del monstruo, uno para el ataque y otro para la defensa.

Por ejemplo:



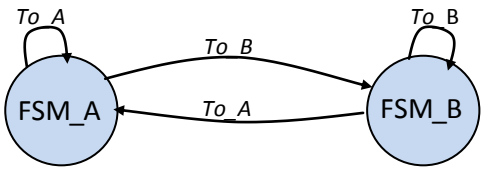
Observamos las transiciones que parten de cada estado y los bucles que conseguirían el objetivo de "quedarse en el mismo estado". Los eventos que activan dichas transiciones podrían compartir nombre.

El siguiente paso (opcional pero aconsejable) seria definir los tags relativos a cada estado-transición-evento-acción relativos a la máquina y proceder a su carga.

Ejemplo 2. Elección de FSM según situación (Meta-FSM)

Una vía muy interesante es utilizar una FSM clásica determinista para elegir la creación de una FSM u otra (estaríamos utilizando una FSM para saber qué FSM utilizar, aunque parezca redundante tiene sentido). Estaríamos diseñando una clase de meta-máquina de estados.

Un ejemplo:



Si se dan los eventos pertinentes a la FSM_A, utilizamos esa máquina de estados, si no, utilizamos la FSM_B. Esto puede ser mucho más grande que en el ejemplo. Es una opción excelente si pretendemos obtener más control sobre nuestra capa de IA.

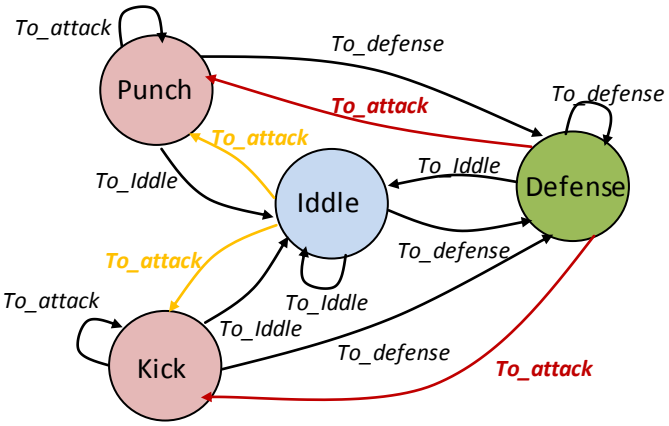
- **FA_Classic (Probabilistic)**

Ejemplo 1. Ataque y defensa imprevisible de un enemigo

Quizá con una máquina de estados finitos determinista no lleguemos a realizar una propuesta demasiado "inteligente" en lo que a la IA de un enemigo respecta. O al menos, no consigamos un comportamiento excesivamente realista.

Para ello, podemos utilizar una máquina de estados probabilista, en la que definamos una distribución de probabilidad de que sucedan ciertas cosas u otras.

Por ejemplo:



Observamos que hay dos estados de color rosado *Punch* y *Kick*. Los dos son estados de ataque. Hay únicamente uno de defensa de color verde llamado *Defense* y uno ocioso llamado *Iddle* (este estado no tiene por qué existir, sólo se asume que el NPC puede no hacer nada).

Si nos fijamos en las transiciones rojas y naranjas, nos damos cuenta de que se activan mediante el mismo evento *To_attack*. Aquí es donde reside la indeterminación (y la distribución de probabilidad que debemos definir), no se sabe si desde el estado *Defense* dado un evento *To_attack*, iremos a *Punch* o *Kick*. Y lo mismo ocurre desde *Iddle*. Podríamos definir un 50% de posibilidades de ir a Punch o Kick, por ejemplo.

La distribución de probabilidad no es exactamente tal y como define el usuario (existen normalizaciones para porcentajes superiores al 100% y para los casos por defecto), pues no depende del evento, es una probabilidad de activación. Se recomienda leer la especificación de la FSM_Classic probabilista en la memoria de trabajo.

La reacción del NPC pretende ser ahora imprevisible, lo cual denota un añadido de realismo e inteligencia.

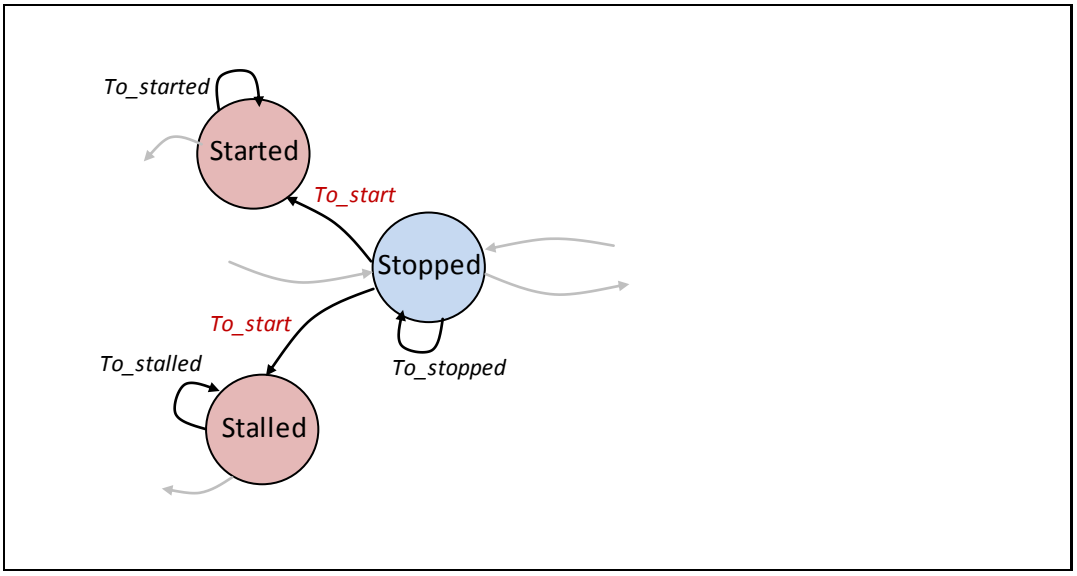
Este ejemplo puede servir para construir cualquier otro caso o situación.

Ejemplo 2. Arrancar un vehículo

Se ha puesto como situación en el *Ejemplo 1* un caso en el que podemos transitar a dos estados de ataque de manera indeterminista, azarosamente.

También es posible plantear situaciones en las que los estados sean completamente diferentes pero activables mediante el mismo evento. Un ejemplo válido, podría ser tratar de arrancar un coche.

Por ejemplo:



El ejemplo es incompleto (el grafo lo es), pero sirve para ilustrar el caso, el cual es parecido al del *Ejemplo 1*, pero diferente en cuanto a que los estados son adversos. En el estado *Stopped*, puede darse un evento para arrancar el coche (*To_start*), y de manera indeterminada pueden suceder dos cosas: o bien transitamos al estado Started (arrancado) o bien al estado Stalled (calado).

Definiendo las probabilidades adecuadas, podemos dar una situación de fallo imprevisto con una probabilidad (baja o no) de que así sea.

4.1.2 Consejos

- **Este tipo de FSM constituye la base de las demás.** Realmente se dan ejemplos únicamente con su tipo base (determinista o probabilista) pero las demás máquinas pueden activar el *flag* de probabilidad para ser probabilistas también (por defecto son deterministas).

- **Útiles para máquinas sencillas, sin comportamientos complejos.**

- **No se deberían tomar estos ejemplos como paradigma**. Es posible que existan diseños mejores para los casos mostrados.

4.2 Ejemplos de FSM utilizando FA_Inertial

Se muestran un conjunto de ejemplos para la FSM inercial y consejos de uso.

4.2.1 Ejemplos para la FSM inercial

Se ofrecen dos ejemplos de construcción de una FSM inercial.

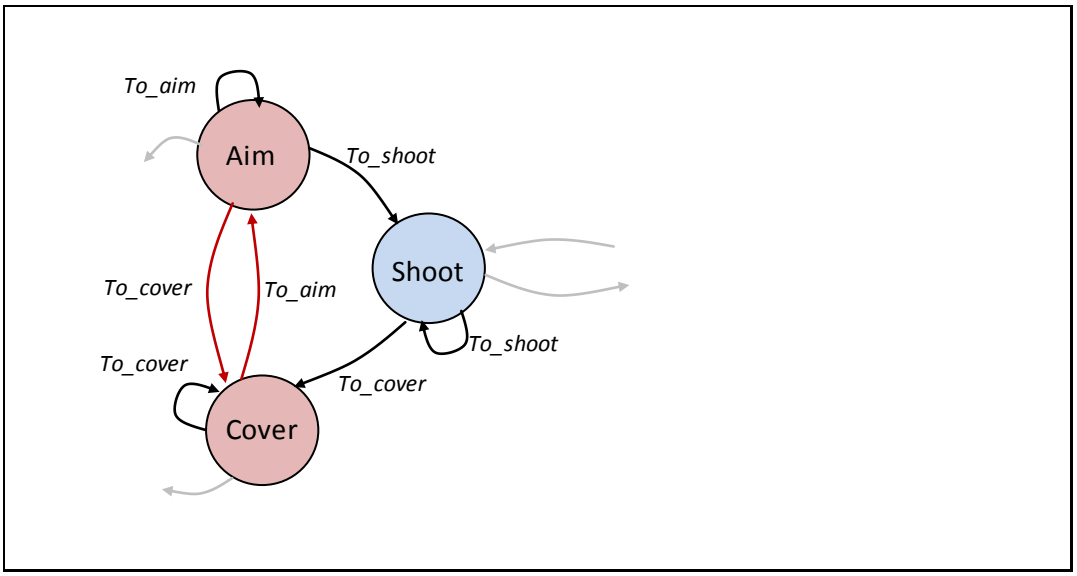
Ejemplo 1. Apuntar y cubrirse

Imaginemos un tiroteo en el cual hay un grupo de NPCs que rodean al personaje principal.

En el propio tiroteo, el NPC tiene un comportamiento sencillo, que es apuntar siempre que vea al personaje y por ejemplo cubrirse si no lo ve.

Si se da la situación nerviosa en la que el personaje aparece y se cubre muy rápidamente, el comportamiento del NPC puede ser igualmente nervioso o dar sensación de reacción muy repentina.

Si se plantea lo siguiente:



Se muestra un fragmento de lo que sería una FSM más grande correspondiente a dicha situación en la que se encuentra un NPC enemigo.

Imaginemos que el NPC está cubierto (estado *Cover*) mientras le llega el estímulo o evento de apuntar (que se dará según cualquier condición que no se contempla aquí), y transita al estado de apuntar (estado *Aim*) pero una vez apuntando llega un evento para cubrirse (bucle indicado en rojo).

Esto implica un probable comportamiento nervioso que se podría solventar indicando una **latencia** en los estados.

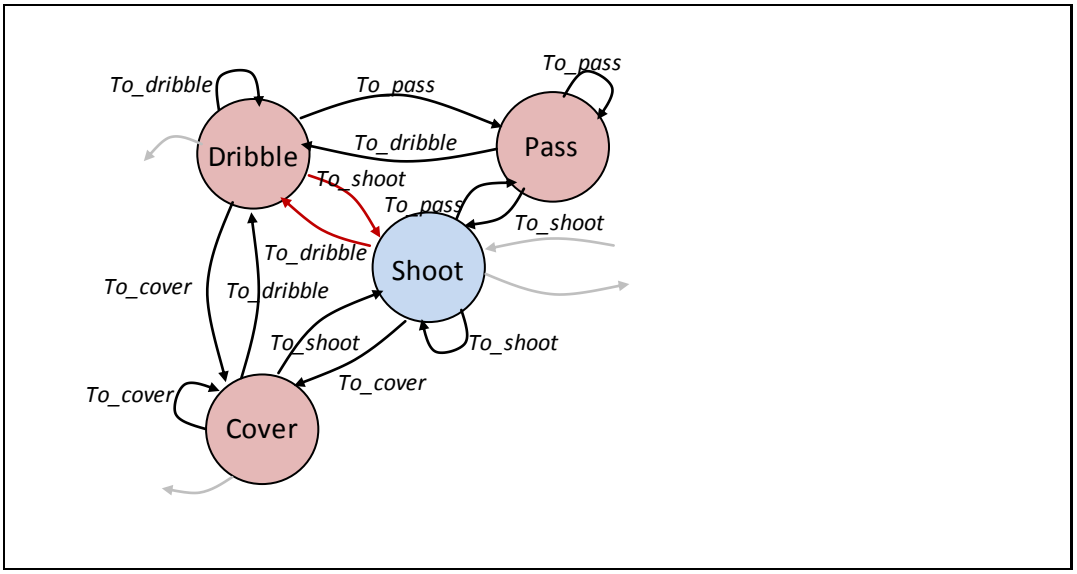
De esta forma, el NPC apuntará y seguirá haciéndolo aunque se produzca otro evento, de igual forma se cubrirá y permanecerá en dicho estado el tiempo de latencia indicado.

Ejemplo 2. Chutar a portería o driblar al rival

El siguiente ejemplo utiliza un partido de fútbol como contexto de uso.

Imaginemos que un NPC rival pretende disparar a puerta, y se da la condición de disparar cuando divisa la portería. Seguidamente, un personaje jugable le tapa su objetivo y deja de darse el evento de disparo (armar la pierna para disparar) y se da el de regate (*To_dribble*). Como en el *Ejemplo 1,* el NPC vuelve a divisar la portería y nuevamente se da el evento de disparo del balón.

Esto último es también un comportamiento nervioso. Por ejemplo:



En cualquiera de los bucles entre estados se puede dar este comportamiento aunque en el ejemplo, se muestra en color rojo el mencionado antes.

Indicando una latencia en el estado *Dribble o* en cualquiera de los estados que pueda conllevar un comportamiento nervioso, evitaríamos este problema. Aunque divisara la portería, el NPC no dispararía el balón hasta no pasada una latencia (o no driblaría, o no cubriría el balón…).

4.2.2 Consejos

- **No utilizar latencias demasiado grandes**. Esto puede provocar un comportamiento excesivamente lento.

- **La latencia se establece en milisegundos**. Un segundo de retardo se indicaría como 1000.

4.3 Ejemplos de FSM utilizando FA_Stack_based

Este tipo de máquina permite la introducción de estados prioritarios sobre los demás. Se ofrecen ejemplos relativos a la máquina basada en pilas y consejos de uso a continuación.

4.3.1 Ejemplos para la FSM Stack based

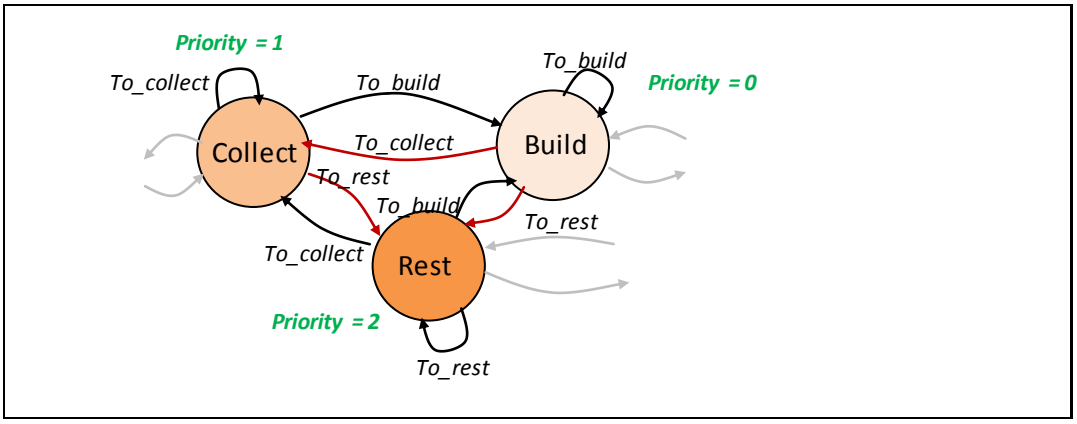
Se ofrecen dos ejemplos para dos casos diferentes.

Ejemplo 1. Recogida de recursos

En este ejemplo imaginemos que nos encontramos en un juego de estrategia en tiempo real y nuestro rival pretende construir una serie de infraestructuras para prepararse contra nosotros.

Para ello tiene que construir academias y granjas siempre que tenga recursos suficientes. Si no tiene, interrumpirá su estado de construcción para ir a recolectar lo necesario. Pero si está recolectando o construyendo, su energía puede agotarse y entonces necesite descansar para recuperarse.

Un ejemplo de máquina de estados (un fragmento de ella) para los NPCs del rival podría ser:



Este ejemplo es muy ilustrativo. Tenemos tres estados cuya prioridad es diferente. Ordenados:

Rest (2) > Collect (1) > Build (0)

Así pues, podemos afirmar que los eventos son entre ellos más o menos prioritarios también, pues es el resultado de otorgar prioridad a los estados:

To_rest > To_collect > To_build

Las transiciones marcadas en rojo deben ser activables por eventos de tipo ***STACKABLE***, que apilarán el estado anterior y tratarán el prioritario. De esta forma, *Rest* podrá apilar a *Collect* o *Build*, y *Collect* únicamente podrá apilar a *Build*. Los demás eventos podrían ser de tipo *BASIC*. Eso sí, un evento stackable no apilará el estado anterior si el estado destino es igual o menos prioritario.

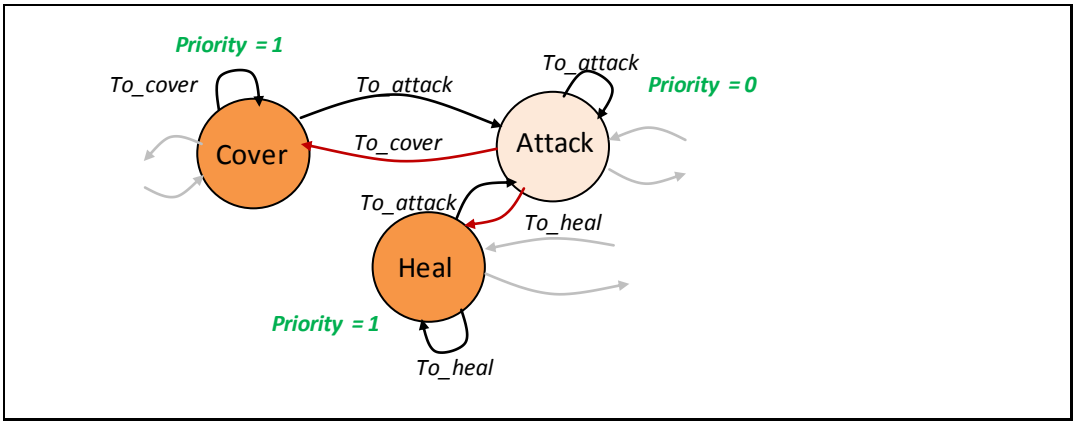
Es tarea del diseñador que la prioridad de los estados guarde coherencia con los eventos *Stackables*. Por lo general, **si un estado es más prioritario, llegarán a él transiciones que se activarán por eventos de tipo *Stackable***.

Ejemplo 2. Hechizar, cubrirse o curarse

Un NPC enemigo se encuentra en medio de un combate mágico entre hechiceros (parecido a los ejemplos anteriores). La toma de decisiones puede ser crucial para salvarse en medio del fuego.

El NPC puede lanzar maleficios con su varita, puede cubrirse en algún objeto cercano o lanzarse hechizos curativos.

Un fragmento de su FSM:



Para este caso, el reparto de prioridades es:

Cover (1) & Heal (1) > Attack (0)

El NPC no puede lanzarse hechizos curativos si está cubriéndose (quizá porque requiere realizar movimientos específicos para ello). Esto implica que tanto *Cover* como *Heal* tengan la misma prioridad y no sean accesibles entre ellos.

Así pues, las transiciones que parten de *Attack* (marcadas en rojo) hacia dichos estados son activables mediante eventos de tipo ***STACKABLE***. Los demás pueden ser de tipo BASIC.

El NPC atacará hasta que llegue un evento más prioritario y decida cubrirse o curarse. Luego volverán al estado de ataque para intentar ganar el combate.

4.3.2 Consejos

- **La prioridad mínima es 0 y no hay un máximo**. Este valor no tiene sentido que sea negativo.

- **Utilizar si se quiere otorgar memoria a un personaje**. Útil para situaciones en las que hay que recordar un caso anterior.

- **La máquina de pilas no tiene que utilizar únicamente eventos STACKABLE**. Como se ha mostrado, pueden coexistir en el diseño eventos BASIC y STACKABLE, y utilizar estos últimos únicamente para los apilados requeridos.

- **Es recomendable poner especial atención en la definición de prioridades.** Los resultados pueden ser completamente incoherentes de no definirlas bien.

4.4 Ejemplos de FSM utilizando FA_Concurrent_States

A continuación se muestran ejemplos para FSMs de estados concurrentes. Son las más complicadas de llevar a buen término dado que es relativamente sencillo equivocarse en el diseño.

4.4.1 Ejemplos para la FSM de estados concurrentes

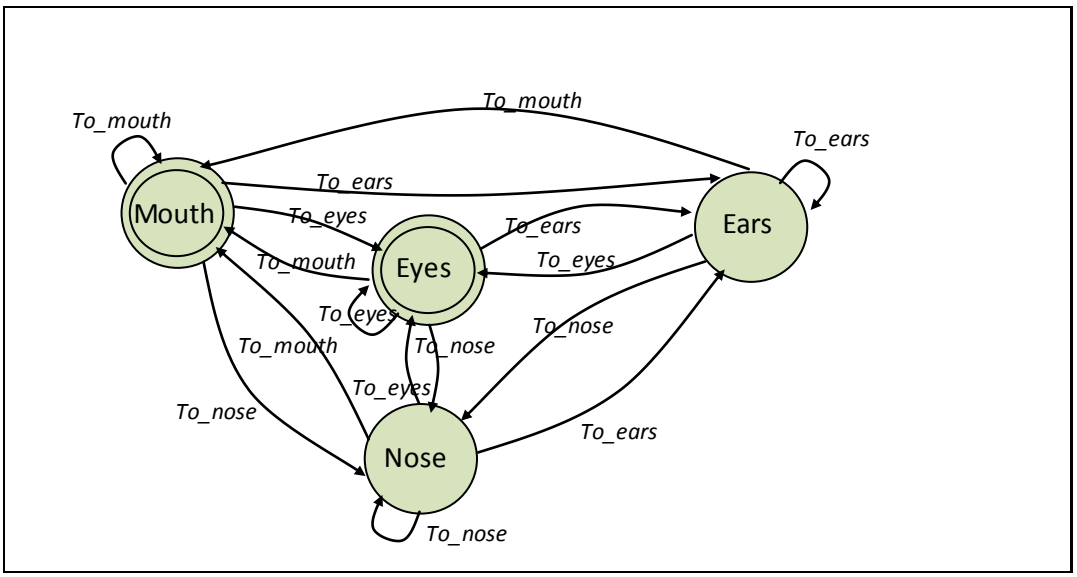
Se muestran dos ejemplos diferentes de control de estados concurrentes.

Ejemplo 1. Animaciones faciales

Imaginemos que queremos obtener el control sobre la ejecución de las animaciones faciales de un personaje concreto.

Estas animaciones no influyen sobre las otras (este tipo de FSM puede otorgar comportamientos incongruentes si sus estados son excluyentes), así que pueden ejecutarse en paralelo.

Un ejemplo de FSM:



Podríamos definir *Mouth* y *Eyes* como estados iniciales y con **un crédito de ejecución** cada uno. Responderíamos a más de un evento y podríamos estar hasta en dos estados a la vez.

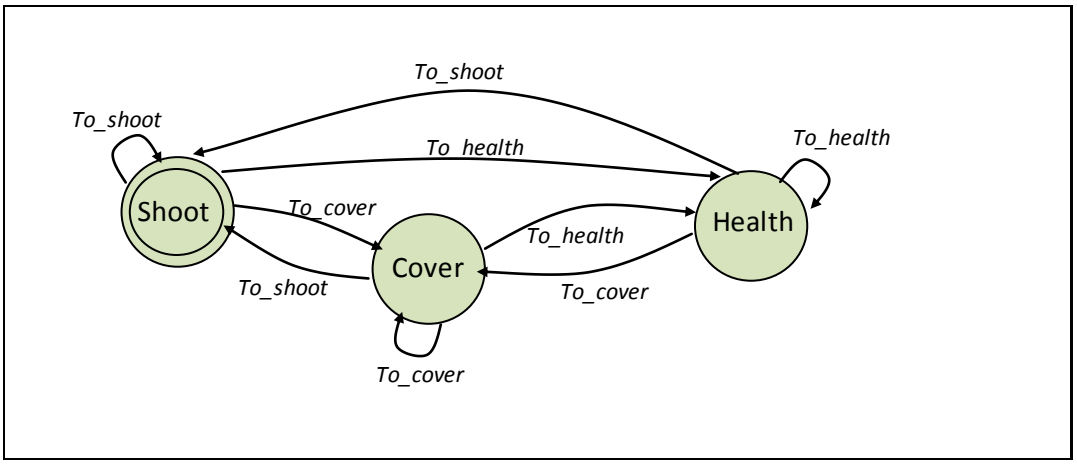
Si llegaran los eventos *To_nose* o *To_ears*, transitaríamos a dichos estados concediéndoles nuestro crédito de ejecución siempre que en ese ciclo no nos hubiéramos ejecutado a nosotros mismos.

Siguiendo esta vía, un personaje podría mover la boca y mover la nariz a la vez. O pestañear y menear las orejas, etc…

Ejemplo 2. Disparar cubierto, curarse disparando, etc…

Imaginemos un NPC enemigo en un tiroteo (sí, otro). Dicho enemigo puede tener una FSM que le haga realizar varias acciones no excluyentes a la vez.

Un ejemplo sería:



Para este ejemplo, definiríamos el estado *Shoot* como inicial y le otorgaríamos, por ejemplo, **dos créditos de ejecución**.

De esta manera, el personaje únicamente dispararía hasta que llegara otro evento que podría hacer que se curase (transitar con un crédito de ejecución a *Health*) o se cubriese (transitar con un crédito de ejecución a *Cover*) automáticamente sin dejar de disparar. De igual manera podría cubrirse mientras se está curando.

Es otro ejemplo más de máquina de estados concurrentes los cuales no son excluyentes (es decir, no se molestan en términos de ejecución).

4.4.2 Consejos

- **Utilizar con estados no excluyentes entre sí**. Se ha comentado en los ejemplos; una construcción incoherente conllevará a resultados igualmente incoherentes.

- **La concurrencia de máquinas puede llevarse a cabo utilizando una meta-FSM de estados concurrentes con jerarquía**. Estas máquinas tampoco deberían ser excluyentes. También es posible utilizar dos máquinas (con dos objetos FSM_Machine) por separado y de forma paralela.

- **El diseño de esta FSM es complicado**. Se recomienda evitar utilizarla en casos sencillos o cuyo uso no esté claro ni sea necesario.

- **Se recomienda repartir los créditos de ejecución entre los estados marcados como iniciales**. Posteriormente, el flujo natural de la máquina

puede hacer el resto. Es necesario poner especial atención en el reparto de dichos créditos o no se conseguirán los efectos deseados.

- **Esta máquina sí puede tratar más de un evento**, a diferencia de las demás.

ANEXO 2. Enlaces a Tablas, Ilustraciones y Ecuaciones

ILUSTRACIONES

TABLAS

ECUACIONES

ANEXO 3. Plantillas XML

Plantilla para la FSM Clásica Determinista

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Author: José Alapont Luján-->

<FSM_Machine>
  <!--FSM specification -->
  <FSMtype Probabilistic="NO">CLASSIC</FSMtype>
  <!--Probabilistic has to be "NO" to be Deterministic -->
  <!--TYPE can be CLASSIC, INERTIAL, STACK_BASED or CONCURRENT_STATES-->
  <FSMid>
    <!--put here the name of this machine-->
  </FSMid>
  <Fsm>
    <Callback>
      <!--put here the name of events routine-->
    </Callback>
    <!--Method for events that concern to this FSM -->
    <States>
      <State Initial="">
        <!--Initial has to be "YES" or "NO" -->
        <S_Name>
          <!--Put here the name of this state (no quotes)-->
        </S_Name>
        <S_Action>
          <!--Put here the name of this state action (no quotes, it can be NULL)-->
        </S_Action>
        <S_inAction>
          <!--Put here the name of this state IN action (no quotes, it can be NULL) -->
        </S_inAction>
        <S_outAction>
          <!--Put here the name of this state OUT action state (no quotes, it can be NULL) -->
        </S_outAction>
        <S_Fsm>
          <!--Put here the PATH to a submachine (this could be empty) -->
        </S_Fsm>
      </State>
      <!--ADD MORE STATES JUST LIKE THE FIRST -->
    </States>

    <Transitions>
      <Transition>
        <T_Name>
          <!--Put here the name of this transition (no quotes)-->
        </T_Name>
        <T_Origin>
          <!--Put here the name of the origin state (no quotes)-->
        </T_Origin>
        <T_Destination>
          <!--Put here the name of destination state (no quotes)-->
        </T_Destination>
        <T_Action>
          <!--Put here the name of this state (no quotes, it can be NULL)-->
        </T_Action>
        <Events>
          <!--List of events that enable this Transition-->
          <Event>
            <ID>
              <!--Put here the name of this Event (no quotes)-->
            </ID>
            <Type>
              <!--Put here the type of this Event (no quotes)-->
            </Type>
            <!--Type can be BASIC or STACKABLE-->
          </Event>
          <!--Add another <Event> if you want-->
        </Events>
      </Transition>
      <!--ADD MORE TRANSITIONS JUST LIKE THE FIRST-->
    </Transitions>
```

```
  </Fsm>
</FSM_Machine>
```

Plantilla para la FSM Clásica Probabilista

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Author: José Alapont Luján-->

<FSM_Machine>
  <!--FSM specification -->
  <FSMtype Probabilistic="YES">CLASSIC</FSMtype>
  <!--Probabilistic has to be "YES"-->
  <!--TYPE can be CLASSIC, INERTIAL, STACK_BASED or CONCURRENT_STATES-->
  <FSMid>
    <!--put here the name of this machine-->
  </FSMid>
  <Fsm>
    <Callback>
      <!--put here the name of events routine-->
    </Callback>
    <!--Method for events that concern to this FSM -->
    <States>
      <State Initial="">
        <!--Initial has to be "YES" or "NO" -->
        <S_Name>
          <!--Put here the name of this state (no quotes)-->
        </S_Name>
        <S_Action>
          <!--Put here the name of this state action (no quotes, it can be
NULL)-->
        </S_Action>
        <S_inAction>
          <!--Put here the name of this state IN action (no quotes, it can be
NULL) -->
        </S_inAction>
        <S_outAction>
          <!--Put here the name of this state OUT action state (no quotes, it
can be NULL) -->
        </S_outAction>
        <S_Fsm>
          <!--Put here the PATH to a submachine (this could be empty) -->
        </S_Fsm>
      </State>
      <!--ADD MORE STATES JUST LIKE THE FIRST -->
    </States>

    <Transitions>
      <Transition>
        <T_Name>
          <!--Put here the name of this transition (no quotes)-->
        </T_Name>
        <T_Origin>
          <!--Put here the name of the origin state (no quotes)-->
        </T_Origin>
        <T_Destination>
          <!--Put here the name of destination state (no quotes)-->
        </T_Destination>
        <T_Action>
          <!--Put here the name of this state (no quotes, it can be NULL)-->
        </T_Action>
```

```
        <Events>
          <!--List of events that enable this Transition-->
          <Event>
            <ID>
              <!--Put here the name of this Event (no quotes)-->
            </ID>
            <Type>
              <!--Put here the type of this Event (no quotes)-->
            </Type>
            <!--Type can be BASIC or STACKABLE-->
          </Event>
          <!--Add another <Event> if you want-->
        </Events>
        <Probability>
          <!--put here the probability of execution of this Transition (between
0 and 100)-->
        </Probability>
      </Transition>
      <!--ADD MORE TRANSITIONS JUST LIKE THE FIRST-->
    </Transitions>
  </Fsm>
</FSM_Machine>
```

Plantilla para la FSM Inercial

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Author: José Alapont Luján-->

<FSM_Machine>
  <!--FSM specification -->
  <FSMtype Probabilistic="NO">INERTIAL</FSMtype>
  <!--Probabilistic has to be "YES" (PROBABILISTIC) or "NO" (DETERMINISTIC) -->
  <!--TYPE can be CLASSIC, INERTIAL, STACK_BASED or CONCURRENT_STATES-->
  <FSMid>
    <!--put here the name of this machine-->
  </FSMid>
  <Fsm>
    <Callback>
      <!--put here the name of events routine-->
    </Callback>
    <!--Method for events that concern to this FSM -->
    <States>
      <State Initial="">
        <!--Initial has to be "YES" or "NO" -->
        <S_Name>
          <!--Put here the name of this state (no quotes)-->
        </S_Name>
        <S_Action>
          <!--Put here the name of this state action (no quotes, it can be NULL)-->
        </S_Action>
        <S_inAction>
          <!--Put here the name of this state IN action (no quotes, it can be NULL) -->
        </S_inAction>
        <S_outAction>
          <!--Put here the name of this state OUT action state (no quotes, it can be NULL) -->
        </S_outAction>
        <S_Fsm>
          <!--Put here the PATH to a submachine (this could be empty) -->
        </S_Fsm>
        <S_Latency>
          <!--Put here LATENCY of this state -->
        </S_Latency>
      </State>
      <!--ADD MORE STATES JUST LIKE THE FIRST -->
    </States>

    <Transitions>
      <Transition>
        <T_Name>
```

```
          <!--Put here the name of this transition (no quotes)-->
        </T_Name>
        <T_Origin>
          <!--Put here the name of the origin state (no quotes)-->
        </T_Origin>
        <T_Destination>
          <!--Put here the name of destination state (no quotes)-->
        </T_Destination>
        <T_Action>
          <!--Put here the name of this state (no quotes, it can be NULL)-->
        </T_Action>
        <Events>
          <!--List of events that enable this Transition-->
          <Event>
            <ID>
              <!--Put here the name of this Event (no quotes)-->
            </ID>
            <Type>
              <!--Put here the type of this Event (no quotes)-->
            </Type>
            <!--Type can be BASIC or STACKABLE-->
          </Event>
          <!--Add another <Event> if you want-->
        </Events>
      </Transition>
      <!--ADD MORE TRANSITIONS JUST LIKE THE FIRST-->
    </Transitions>
  </Fsm>
</FSM_Machine>
```

Plantilla para la FSM basada en pilas

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Author: José Alapont Luján-->

<FSM_Machine>
  <!--FSM specification -->
  <FSMtype Probabilistic="NO">STACK_BASED</FSMtype>
  <!--Probabilistic has to be "NO" to be Deterministic -->
  <!--TYPE can be CLASSIC, INERTIAL, STACK_BASED or CONCURRENT_STATES-->
  <FSMid>
    <!--put here the name of this machine-->
  </FSMid>
  <Fsm>
    <Callback>
      <!--put here the name of events routine-->
    </Callback>
    <!--Method for events that concern to this FSM -->
    <States>
      <State Initial="">
        <!--Initial has to be "YES" or "NO" -->
        <S_Name>
          <!--Put here the name of this state (no quotes)-->
        </S_Name>
        <S_Action>
          <!--Put here the name of this state action (no quotes, it can be NULL)-->
        </S_Action>
        <S_inAction>
          <!--Put here the name of this state IN action (no quotes, it can be NULL) -->
        </S_inAction>
        <S_outAction>
          <!--Put here the name of this state OUT action state (no quotes, it can be NULL) -->
        </S_outAction>
        <S_Fsm>
          <!--Put here the PATH to a submachine (this could be empty) -->
        </S_Fsm>
        <S_Priority>
          <!--Put here the priority of this state (>=0) the greater, the more priority -->
        </S_Priority>
      </State>
```

```
      <!--ADD MORE STATES JUST LIKE THE FIRST -->
    </States>

    <Transitions>
      <Transition>
        <T_Name>
          <!--Put here the name of this transition (no quotes)-->
        </T_Name>
        <T_Origin>
          <!--Put here the name of the origin state (no quotes)-->
        </T_Origin>
        <T_Destination>
          <!--Put here the name of destination state (no quotes)-->
        </T_Destination>
        <T_Action>
          <!--Put here the name of this state (no quotes, it can be NULL)-->
        </T_Action>
        <Events>
          <!--List of events that enable this Transition-->
          <Event>
            <ID>
              <!--Put here the name of this Event (no quotes)-->
            </ID>
            <Type>
              <!--Put here the type of this Event (no quotes)-->
            </Type>
            <!--Type can be BASIC or STACKABLE (needed here)-->
          </Event>
          <!--Add another <Event> if you want-->
        </Events>
      </Transition>
      <!--ADD MORE TRANSITIONS JUST LIKE THE FIRST-->
    </Transitions>
  </Fsm>
</FSM_Mach
```

Plantilla para la FSM de estados concurrentes

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Author: José Alapont Luján-->

<FSM_Machine>
  <!--FSM specification -->
  <FSMtype Probabilistic="NO">CONCURRENT_STATES</FSMtype>
  <!--Probabilistic has to be "YES" or "NO" -->
  <!--TYPE can be CLASSIC, INERTIAL, STACK_BASED or CONCURRENT_STATES-->
  <FSMid>
    <!--put here the name of this machine-->
  </FSMid>
  <Fsm MaxConcurrent="">
    <!--Max number of concurrent states "x"-->
    <Callback>
      <!--put here the name of events routine-->
    </Callback>
    <!--Method for events that concern to this FSM -->
    <States>
      <State Initial="">
        <!--Initial has to be "YES" or "NO" -->
        <S_Name>
          <!--Put here the name of this state (no quotes)-->
        </S_Name>
        <S_Action>
          <!--Put here the name of this state action (no quotes, it can be NULL)-->
        </S_Action>
        <S_inAction>
          <!--Put here the name of this state IN action (no quotes, it can be NULL) -->
        </S_inAction>
        <S_outAction>
          <!--Put here the name of this state OUT action state (no quotes, it can be NULL) -->
        </S_outAction>
```

```
        <S_Fsm>
          <!--Put here the PATH to a submachine (this could be empty) -->
        </S_Fsm>
        <S_Credits>
          <!--Put here initial execution credits (>=0, no quotes) -->
        </S_Credits>
      </State>
      <!--ADD MORE STATES JUST LIKE THE FIRST -->
    </States>

    <Transitions>
      <Transition>
        <T_Name>
          <!--Put here the name of this transition (no quotes)-->
        </T_Name>
        <T_Origin>
          <!--Put here the name of the origin state (no quotes)-->
        </T_Origin>
        <T_Destination>
          <!--Put here the name of destination state (no quotes)-->
        </T_Destination>
        <T_Action>
          <!--Put here the name of this state (no quotes, it can be NULL)-->
        </T_Action>
        <Events>
          <!--List of events that enable this Transition-->
          <Event>
            <ID>
              <!--Put here the name of this Event (no quotes)-->
            </ID>
            <Type>
              <!--Put here the type of this Event (no quotes)-->
            </Type>
            <!--Type can be BASIC or STACKABLE-->
          </Event>
          <!--Add another <Event> if you want-->
        </Events>
      </Transition>
      <!--ADD MORE TRANSITIONS JUST LIKE THE FIRST-->
    </Transitions>
  </Fsm>
</FSM_Machine>
```