UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Diseño y especificación de entornos virtuales inteligentes basados en el meta-modelo MAM5

Proyecto Final de Carrera

Ingeniería informática

Autor: Raúl Giner Campos

Director: Carlos Carrascosa Casamayor

Codirector: María Emilia García Marqués

23 de septiembre de 2013

Índice general

Capítulo 1

Introducción

El uso de técnicas de inteligencia artificial (IA) utilizando agentes inteligentes es una disciplina cada vez con mayor actividad y relevancia. Muchas aplicaciones, basadas en este paradigma, están siendo utilizadas hoy en día gracias a su alta capacidad de adaptación a problemas dinámicos y complejos. Estos modelos tienen como función construir sistemas que representen interacciones entre agentes dentro de un entorno, provocando, por ejemplo, situaciones de razonamiento, comunicación o reactividad (actividades propias del ser humano).

Básicamente, un agente inteligente es un proceso computacional capaz de realizar tareas de forma autónoma, comunicándose y cooperando con otros agentes u objetos en un medio, con el fin de lograr un conjunto de objetivos [1]. Un sistema multi-agente, de esta manera, será un conjunto de agentes inteligentes que interactúan entre ellos dentro de un entorno, con el fin de satisfacer una serie de objetivos. Cada agente actúa como un ente autónomo y asíncrono respecto a otros agentes, lo que produce un sistema totalmente heterogéneo y escalable [2]. Este modelo ayuda, sobretodo, a la resolución de problemas de forma distribuida, donde se procesan cantidades masivas de información en diferentes situaciones geográficas [3].

Toda la teoría sobre sistemas multi-agente tiene como base el establecimiento de los agentes en un entorno de actuación. La definición de los agentes y su entorno puede llegar a ser tan concreto o complejo como el usuario crea conveniente para su problema a resolver. En nuestro caso buscamos el definir el entorno como un mundo o universo físico donde situar nuestros agentes como si fueran habitantes inteligentes obteniendo una representación lo más parecida posible al mundo real. Esta simulación la conseguimos a partir de un entorno virtual inteligente (*IVE, Intelligent Virtual Environment*). En un

IVE los entes que lo habitan pueden ser tanto objetos dinámicos o estáticos como representaciones virtuales de formas de vida e, incluso, avatares de usuarios reales [4]. Podemos decir, entonces, que un entorno virtual inteligente es una simulación de un sistema compuesto por agentes cuyo razonamiento o comportamiento está basado en técnicas de inteligencia artificial.

El uso de sistemas multi-agentes e IVEs es muy variado. Algunos campos donde se han aplicado de forma exitosa son:

- Aplicaciones distribuidas [6].
- Simulaciones sociales [7] [8] [9].
- Simulación de mercados [10].
- Simulación de transporte y tráfico [11] [12].
- Comercio electrónico[13] [14].
- Procesos de producción [15] [16].
- Motores gráficos para videojuegos 3D [17] [18].

El presente proyecto pertenece al ámbito del diseño y especificación de entornos virtuales inteligentes y forma parte de un proyecto mayor. Este proyecto global abarca la implementación de una serie de herramientas interoperables basadas en estándares que faciliten el diseño y permitan la simulación gráfica de entornos virtuales inteligentes basados en un modelo ya existente de sistema multi-agente. En concreto este proyecto se encargará de la parte del diseño y la especificación formal de toda la información del entorno virtual inteligente, complementándose con el proyecto, actualmente en fase de desarrollo, encargado de la simulación gráfica y la gestión.

1.1. Objetivos

Este proyecto parte de la propuesta desarrollada en el artículo científico *MAM5: Multi-Agent Model For Intelligent Virtual Environments* [5] sobre la integración de sistemas multi-agente con entornos virtuales inteligentes. El objetivo global, en el que se enmarca el objetivo concreto de este trabajo,

es el de facilitar al desarrollador el diseño, la creación y la posterior representación gráfica de entornos basados en el modelo enunciado anteriormente.

Para conseguir este objetivo global, se realizan los dos siguientes pasos:

- Crear un método de especificación y representación de los datos que forman un IVE y un proceso que lea y analice estos datos a través de un formato interoperable y basado en estándares.
- Implementar un proceso gestor, llamado *Manager*, que ofrezca un entorno gráfico para la simulación de IVEs (utilizando los motores Unity3D y ODE) y que, además, se encargue de la creación y mantenimiento de todos sus elementos.

El objetivo concreto del presente proyecto se centrará, únicamente, en cumplir lo establecido en el primer punto. Para poder conseguir esto se ha planteado la realización de las siguientes tareas generales:

- Especificar formalmente el modelo MAM5 a través de un documento esquema XSD. Esto permitirá la generación de plantillas XML donde se podrán especificar entornos virtuales inteligentes que cumplan lo establecido según el modelo.
- Implementar un proceso que analice la información del IVE, introducida a través de un archivo XML, y la almacene en estructuras de datos.
- Integrar el proceso del punto anterior con el proceso encargado de la simulación y gestión del entorno.

En la figura 1.1 se ilustra el ciclo de diseño y representación de un entorno virtual inteligente a través de la propuesta desarrollada en los puntos anteriores. El ámbito de aplicación de este proyecto abarcaría aquello que queda rodeado en rojo.

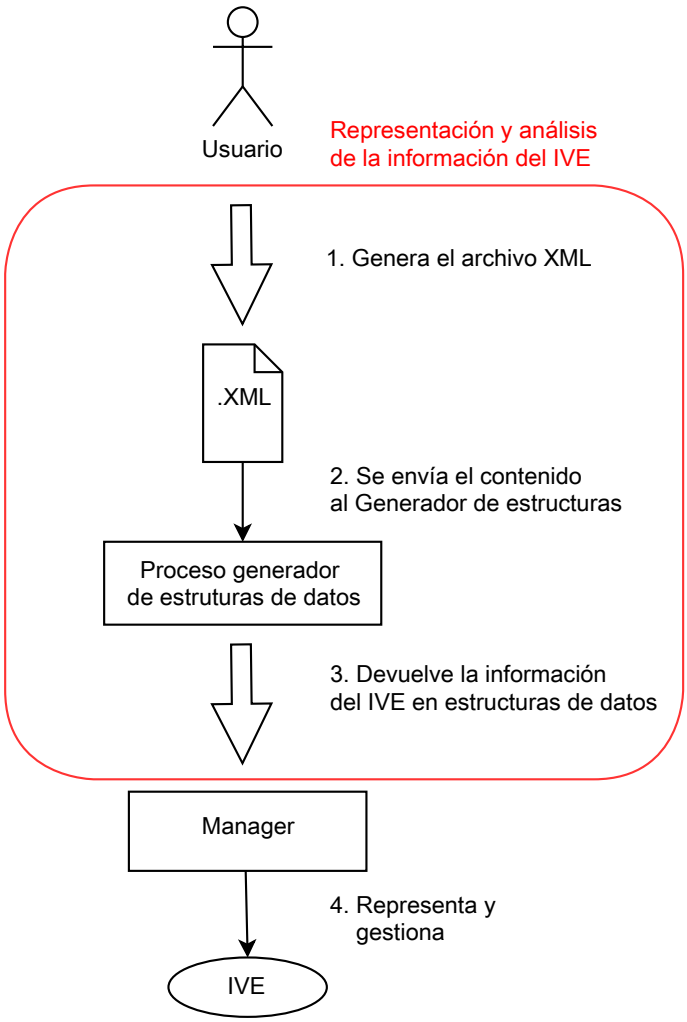

Figura 1.1: Proceso global de creación y representación de IVEs

Capítulo 2

Estado del arte

En este capítulo de la memoria se presentan los fundamentos teóricos necesarios para el entendimiento de nuestra propuesta. Se definen en profundidad los conceptos relacionados con los sistemas multi-agente y los entornos virtuales inteligentes así como el modelo multi-agente que tomamos como base, MAM5. A continuación se expone un pequeño análisis de las diferentes herramientas que se ha considerado utilizar.

2.1. Sistemas Multi-agente

Un agente inteligente es un proceso capaz de realizar tareas de forma autónoma a través de una serie de acciones de comunicación y cooperación con otros agentes para poder conseguir una serie de objetivos. Estos objetivos son, básicamente, los que definen el comportamiento cognitivo del agente con respecto a su entorno. Un sistema multi-agente es la aplicación de esta metodología de forma distribuida en pequeñas partes. Se establece una problemática a resolver y, la ejecución por separado de cada agente, más la unión de lo que hayan producido, permitiría obtener una solución.

Un agente puede ser ejecutado en diversas plataformas, una de las más utilizadas actualmente es Jason [1]. Esta infraestructura de desarrollo de sistemas multi-agente está basada en Java y utiliza una versión extendida del lenguaje AgentSpeak para definir el comportamiento de los agentes.

AgentSpeak es un lenguaje abstracto basado en la arquitectura BDI (*Beliefs-Desires-Intentions*). En este lenguaje los *Beliefs* representan la base de datos de conocimiento del agente sobre el entorno que le rodee. Los *Desires*

[1] http://jason.sourceforge.net/wp/

forman parte del estado motivacional y representan objetivos o situaciones que el agente debe conseguir realizar. Las intenciones o *Intentions* se corresponden con la decisión del agente en su ciclo de razonamiento, es decir, la decisión de las acciones a realizar en cada momento.

2.1.1. Sistemas basados en A&A

Un sistema multi-agente no tiene porqué estar formado solamente por agentes, sobretodo cuando los agentes forman parte de un entorno. Los agentes son proactivos y tienen capacidad de razonamiento, sin embargo, cabe la posibilidad, al diseñar un sistema multi-agente, de que deban existir elementos estáticos sin capacidad de cognición. En estos casos se puede utilizar un sistema basado en el modelo de agentes y artefactos (A&A).

En este modelo se tiene un conjunto dinámico de agentes que trabajan conjuntamente en *workspaces*. Los *workspaces* contienen, a su vez, conjuntos, dinámicos también, de artefactos. Los agentes explotan y usan los artefactos como objetos estáticos no autónomos (i.e. objetos inanimados del mundo real). Los artefactos poseen una serie de propiedades observables (por los agentes del entorno) y una serie de acciones (a ejecutar por parte de los agentes). Estas propiedades formarán parte de la base de conocimientos de un agente cuando haya una acción que las modifique.

Una de las más famosas herramientas de utilización de agentes y artefactos es CArtAgO [2] [25]. Esta plataforma, programada en Java, posee una serie de funciones que permiten implementar en un sistema multi-agente el uso de artefactos. CArtAgO, sin embargo, no proporciona métodos de creación de agentes inteligentes, para esto será necesario usar Jason. La integración entre Jason y CArtAgO tiene todo lo necesario para crear sistemas multi-agente basados en el modelo de A&A.

Para la utilización de CArtAgO se necesitan crear una serie de clases que representen a los artefactos. Estas clases heredan de la clase principal *Artifact* y son utilizadas por los agentes a través del código Jason.

[2] cartago.sourceforge.net/

2.2. Entornos virtuales inteligentes

Los entornos virtuales inteligentes (IVE) aparecen con la aplicación de técnicas de inteligencia artificial (IA) a entornos virtuales (VE). Un entorno virtual es una representanción gráfica de un espacio donde se pueden encontrar situados tanto el usuario como otros elementos que formen parte del sistema y que intenta emular un mundo basado en las interacciones de sus elementos.

La aparición de entidades inteligentes, cuyo comportamiento utiliza algoritmos de IA, dentro de un mundo virtual, es lo que produce la creación de entornos virtuales inteligentes. En estos entornos se intenta simular un mundo físico o real habitado por las entidades inteligentes o agentes del sistema. Estos agentes, en cierta manera, intentarán interactuar con los usuarios situados en el entorno además de con otros agentes, formando un sistema multi-agente con representanción virtual.

2.2.1. Meta-modelo MAM5

Existen múltiples ejemplos de integración de SMAs con IVEs [20] [21] [24]. Normalmente estos ejemplos plantean soluciones específicas dependiendo concretamente del tipo de IA de los agentes y del motor gráfico o sistema utilizado para la representación virtual. Son soluciones *ad-hoc* y, en consecuencia, difícilmente escalabes o reusables dado otro tipo de SMA.

A la hora de implementar una herramienta de creación de IVEs basados en SMAs se ha de buscar lo opuesto a lo anterior. El modelo a utilizar debe permitir al usuario definir cualquier elemento del entorno independientemente de como vaya a representarse el IVE en cuestión. Es por tanto necesario que el usuario sea capaz de definir tanto los agentes como los objetos del entorno y, además, el comportamiento o la forma que tendrá el entorno en sí, para poder representarlo.

El meta-modelo de diseño MAM5 cumple lo anteriormente descrito. Este meta-modelo ha sido concebido para poder definir completamente los elementos que aparezcan en el IVE, desde la parte cognitiva de los agentes (el SMA) hasta la parte relacionada con el entorno virtual (la representación), al mismo nivel de abstracción. Estas dos partes, sin embargo, se dividen internamente. Se tiene, por un lado, la parte de inteligencia y razonamiento de los agentes y por otro, su visualización en el entorno. De esta manera se consigue

realizar la simulación virtual teniendo en cuenta solamente los elementos que han de visualizarse, sin importar la parte autónoma que posean los agentes; con lo que se consigue un único entorno de simulación capaz de representar IVEs totalmente diferentes entre sí.

El meta-modelo MAM5, además, divide el entorno en dos categorías: una que contiene todos aquellos elementos que no se representan virtualmente y otra con los elementos que sí poseen entidad física en el entorno virtual, y que, por tanto, han de visualizarse. Los elementos no virtualmente representables se basan en el modelo de A&A utilizado en la infraestructura CArtAgO mientras que los virtualmente visibles son extensiones de las abstracciones de A&A. Estos últimos elementos heredarán algunas caracterísicas de los no representables, añadiendo aquellas prestaciones necesarias para realizar su representación virtual. En la figura 2.1 se muestra el meta-modelo de un IVE según MAM5.

A continuación se expondrán en detalle los elementos principales de cada una de las partesde la que se compone el meta-modelo MAM5, la parte con elementos representables virtualmente

Elementos principales no representables virtualmente

Estos elementos coinciden totalmente con aquellos presentados en el modelo de A&A y, además, son usados por la herramienta CArtAgO.

- ***Agents***

  Representan las entidades autónomas pro-activas del sistema. Cada agente poseerá un archivo .asl con su código en Jason.

- ***Artifacts***

  Se usan para modelar todos aquellos elementos no autónomos, sin razonamiento del sistema. Representan aquellos objetos o recursos básicos que pueden ser utilizados por los agentes dentro del entorno. Poseen una serie de propiedades observables por otros agentes en el entorno además de las operaciones que se les pueden aplicar.

- ***Workspaces***

  Los *workspaces* son divisiones del entorno que contienen una serie de artefactos y de agentes. Éstos artefactos y agentes pueden estar agrupados por algún tipo de similitud en un mismo *workspace*, lo que provoca que los *workspaces* en sí definan la topología o estructura del entorno.

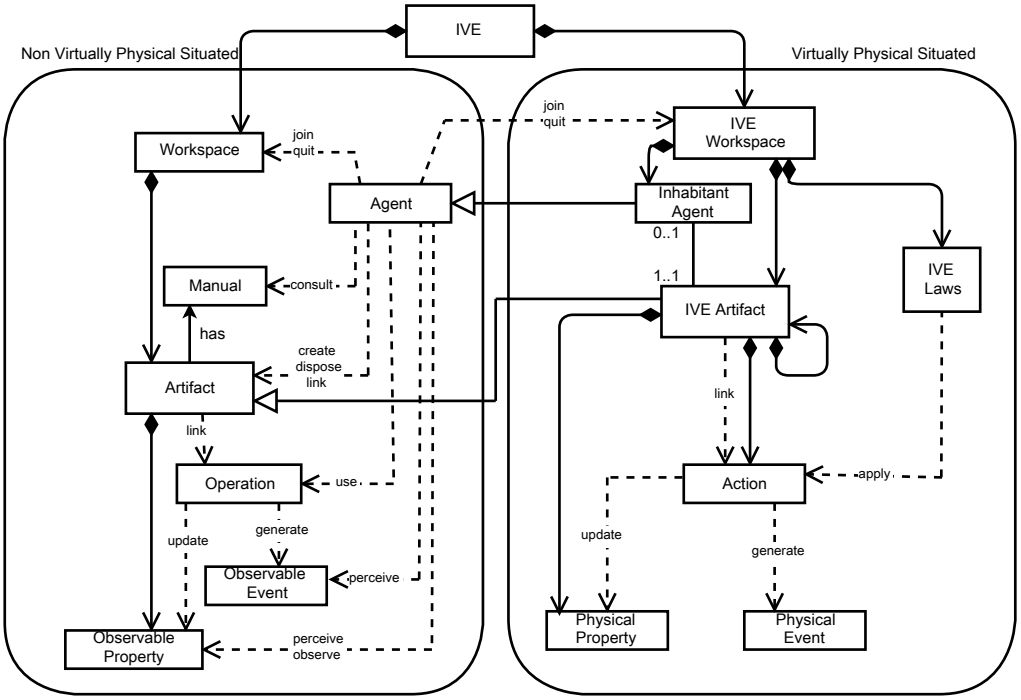

Figura 2.1: Meta-modelo MAM5 de un IVE basado en A&A

Un posible uso de los anteriores elementos en un entorno inteligente podría observarse en un videojuego masivo multijugador en línea (MMOG). Podríamos considerar un agente gestor que se encargara de controlar el ingreso y la salida de los jugadores dentro del mundo del juego. Para ello usaría, entre otros, algún artefacto que almacenara la lista de jugadores conectados cuyas operaciones fueran añadir o eliminar usuario. Todos los agentes y artefactos que se encargaran de la gestión de conexión de jugadores podrían esar contenidos en un mismo *workspace* obteniendo una comunicación e interacción rápida y directa.

Elementos representables de la parte virtual

Al hablar de un entorno virtual inteligente debemos tener en cuenta que debe aparecer un “mundo" o “universo" virtual que, en cierta medida, intente emular el mundo real del ser humano. Se hace necesario, en consecuencia, crear una serie de elementos que vayan a aparecer en este “mundo" físicamente de forma virtual. El meta-modelo MAM5 define los siguientes elementos representables virtualmente que están íntimamente relacionados con el modelo de A&A visto anteriormente.

- ***IVE workspaces***

  Al igual que en el modelo A&A, los *IVE workspaces* definen la topología o estructura del entorno. En este caso esta topología está relacionada con la del mundo virtual, es decir, la forma o topografía del mapa y la disposición física o geográfica de los *IVE artifacts* y *Inhabitant agents* que aparezcan en él. Los *IVE workspace*, además, controlan qué pueden percibir los *Inhabitant agents* y con qué *IVE artifacts* pueden interactuar dependiendo de su posición en el mapa.

  Un *IVE workspace* posee también leyes físicas (*IVE laws*). Estas leyes afectan a aquellas acciones que se ejecuten dentro del *IVE workspace*; un ejemplo podría ser la gravedad del medio (que variaría si el *workspace* fuera un medio acuático o fuera un espacio abierto).

  En un *IVE workspace*, en definitiva, definimos como es el mapa en función de que artefactos y agentes aparecen y en que posición y las leyes físicas que posee quedando definido el comportamiento del entorno en ese *IVE workspace* concreto.

- ***IVE artifacts***

  Un *IVE artifact*, al igual que un artefacto, es un objeto inanimado sin razonamiento propio que puede situarse dentro del mundo virtual ocupando un espacio (cajas, mesas, árboles, cuerpos, etc.). Los *IVE artifacts* pueden ser usados por cualquier *Agent* o *Inhabitant agent* a través de acciones, lo que provoca que se generen una serie de eventos físicos en el entorno.

  Además, un *IVE artifact* poseerá una serie de propiedades físicas (análogas a las propiedades observables de un *Artifact*) que se modificarán a través de acciones o eventos.

  Un *IVE artifact* al estar situado en un mapa virtual debe poseer una serie de propiedades físicas básicas (posición, velocidad, forma, etc.). Algunas de estas propiedades podrán ser observadas por otras entidades del sistema y otras serán internas al artefacto. Los *IVE artifacts* a nivel de CArtAgO poseerán una clase que hereda de la clase *Artifact* general añadiendo las propiedades físicas básicas descritas anteriormente.

Un ejemplo simple de un *IVE artifact* podría ser un árbol. Un árbol posee una serie de propiedades físicas observables como la posición y la forma, pero, además, posee una propiedad interna relacionada con su vitalidad. Una acción ligada al árbol sería la de ser talado. A través de la acción talar se modifica directamente su propiedad vitalidad. En cuanto esta vitalidad llegara a cero se produciría un evento físico (observable por otros agentes) que sería la caída del árbol.

- ***Inhabitant agents***

  Los *Inhabitant agents* representan las entidades autónomas presentes en el mundo virtual. Cada *Inhabitant agent* posee un *IVE artifact* como cuerpo, su *body artifact*, a través del cual realiza acciones en el entorno virtual y percibe eventos físicos. Este cuerpo le permite, además, situarse dentro de un *IVE workspace*.

  Al igual que un agente no virtual los *Inhabitant agents* poseerán un archivo .asl con código Jason que definirá el comportamiento del agente en el entorno.

Anotaciones sobre el meta-modelo MAM5

El meta-modelo MAM5 destaca por:

- Definir completamente un IVE de manera formal.
- Establecer una clara separación entre la parte de inteligencia artificial y la parte de visualización del entorno, conectándolas a través de un agente *Manager* que, por lo tanto, gestiona y representa el entorno.
- Definir la estructura y topología de un entorno (tanto su representanción como la parte computacional) de manera que evite la modificación de código del *Manager* a la hora de gestionar y representar IVEs completamente diferentes entre sí.
- Tratar la parte no virtual y la virtual al mismo nivel de abstracción utilizando el modelo A&A como herramienta de definición de elementos físicos en el entorno virtual (lo que llamamos *IVE artifacts* e *Inhabitant agents*). Los *Inhabitant agents* tendrán acceso al mundo virtual como si se tratara de un mundo físico real en el que aparecen situados proporcionando a la parte de IA como actuar y reaccionar al entorno.

- Ayudar a modelar la forma del entorno a través de sus elementos y su estructura y comportamiento independientemente de su simulación y representanción. Además ayuda a la realización de prototipos permitiendo definir diferentes entornos de forma escalable, y a través del uso del proceso *Manager*, gestionarlos de forma dinámica.

2.3. Representación y análisis de la información

Para la especificación de entornos virtuales inteligentes y la definición de comportamientos en sistemas multi-agente es común el uso de archivos implementados en lenguaje XML [22] [23].

El lenguaje XML (*eXtensible Markup Language*) es un subconjunto de SGML (*Standard Generalized Markup Language*) que define un formato de texto diseñado para el almacenamiento y la transmisión de datos estructurados entre diferentes plataformas.

Este tipo de lenguaje es de gran ayuda para la definición de SMAs debido a las relaciones estructurales que permite definir entre elementos. El modelo MAM5 se puede ver como un conjunto de relaciones entre agentes y artefactos con el entorno en sí. Un *workspace* estará formado por agentes, artefactos y otros elementos, que, a su vez, están formados por acciones, propiedades y/o variables.

Existen también otras ventajas a tener en cuenta:

- Permite, a partir de herramientas de edición, crear una estructura sintáctica que defina el tipo de archivo XML (en este caso establecer qué etiquetas XML son correctas para la creación de un SMA concreto).

- Ofrece, además, la posibilidad de poder generar automáticamente esqueletos XML para usar en procesos de test.

- Existen múltiples herramientas de lectura de XML en lenguaje Java que permiten el análisis de la información que contiene.

2.3.1. Herramientas de edición XML

Existen muchas y diversas herramientas de edición de archivos XML (EditiX [3], Oxygen [4], Notepad++ [5], XMLBlueprint [6] o el propio Eclipse). En este caso, el objetivo buscado en la edición del archivo era el poder definir su estructura sintáctica para una posterior validación léxica, además de poder generar archivos automáticamente. Esto lo conseguimos creando un documento esquema, XSD (*XML Schema definition*).

La herramienta elegida ha sido el editor por defecto de Eclipse (Eclipse XML Editors and Tools). Este editor, además de soportar la creación de archivos XSD y su posterior validación y generación autómatica, utiliza la licencia pública Eclipse (EPL), lo que lo converite en una herramienta de "software libre". Además, al implementar la totalidad del proyecto en Eclipse, se crea un entorno de trabajo perfecto para integrar las dos partes del proyecto: la inicialización y análisis de la información del IVE y la gestión del entorno por parte del *Manager*.

2.3.2. Herramientas de análisis

Existen diversas librerías en Java que nos ayudan a analizar un archivo XML, algunas de las más usadas pueden ser SAX [7], DOM [8] o JAXP [9]. En el caso que nos concierne se tuvieron en cuenta, básicamente, dos: SAX (*Simple API for XML*) y DOM (*Document Object Model*).

Las principales características de estas dos APIs son las siguientes:

- SAX. Esta API de análisis es secuencial y utiliza eventos. Un analizador sintáctico basado en SAX devolverá un evento cada vez que se lea una etiqueta de inicio o de fin. SAX, por tanto, nos permite tratar la información durante el análisis del documento de forma secuencial.

- DOM. Con DOM, en cambio, además de poder tratar otro tipo de documentos, como HTML o XHTML, analizamos, inicialmente, todo el documento por completo. En este análisis se crea una estructura de

[3] http://www.editix.com/
[4] http://www.oxygenxml.com/
[5] http://notepad-plus-plus.org/
[6] http://www.xmlblueprint.com/
[7] http://www.saxproject.org/
[8] http://www.w3schools.com/dom/
[9] http://jaxp.java.net/

árbol que guarda toda la información de las etiquetas del documento. Este árbol es devuelto por el analizador para su posterior navegación y acceso a la información originalmente almacenada en el XML.

El uso de SAX es adecuado cuando no se necesita acceder varias veces al documento XML. Al estar basado en eventos se puede decidir qué realizar en cada momento al encontrarse una etiqueta.

Si bien es cierto que el análisis de la información es más sencillo con DOM, en este caso se debería acceder varias veces a la información del documento, una para la creación del árbol inicial y otra para leer este árbol y crear las estructuras necesarias. Una posible ventaja es que cualquier información del documento puede ser accedida en cualquier orden.

Dado el gran tamaño de los archivos XML al definir un IVE (ver casos de estudio en el capítulo 4) y al ser el análisis de la información una parte necesaria en la ejecución de un IVE se consideró como mejor opción utilizar un analizador sintáctico basado en SAX. De esta forma se recorre el documento XML una sola vez priorizando el coste temporal y espacial respecto a la complejidad del análisis.

Capítulo 3

Representación y análisis de la información del IVE

3.1. Introducción

La representación y el análisis de la información del entorno virtual inteligente es solo el primer paso en la creación total de un IVE basado en MAM5 (ver figura 1.1).

Para poder realizar este paso el objetivo es generar un documento XSD, basado en los conceptos de MAM5, para que el desarrollador, fácilmente, pueda definir un IVE y generar el XML que lo represente. La traducción de los datos del archivo tendrá que realizarse en términos de Jason y CArtAgO para su posterior gestión y simulación gráfica por parte del proceso *Manager*. Cualquier IVE definido de esta manera tendrá una estructura basada en el modelo de A&A y, más concretamente, en el modelo MAM5.

La segunda sección de este capítulo (ver sección 3.2) se encarga de la especificación formal del modelo en el documento XSD. Para ello partimos de la base estructural y jerárquica (i.e. la ontología) de los elementos de un entorno según la teoría del meta-modelo MAM5. A partir de aquí y, una vez generado el XSD, analizamos como poder definir todos los elementos del IVE en el archivo XML. Además se expone en detalle la fase de generación y análisis sintáctico del documento XML gracias a la creación del archivo XSD.

Como hemos comentado anteriormente el *Manager* es el encargado de gestionar los recursos (creación de artefactos, agentes, etc.) y representar los diferentes elementos del entorno definidos en los archivos XML. Este pro-

ceso necesitará, además de la información del archivo XML en estructuras de datos, que algunos elementos del IVE estén representados en el ámbito CArtAgO a través de clases para su posterior uso en Jason. Esta generación de estructuras y creación de clases se realiza a través de un proceso que, además, tendrá implementado un análisis semántico del archivo XML (ver sección 3.3). La posterior integración del proceso generador de estructuras con el proceso *Manager* aparece en la sección 3.4.

Todo este proyecto ha sido creado utilizando el entorno de programación Eclipse IDE for Java EE Developers. El generador de estructuras está implementado en Java utilizando la librería SAX (Simple API for XML). Y el diseño del archivo XSD se ha realizado con el editor por defecto que proporciona Eclipse (Eclipse XML Editors and Tools) [1].

La figura 3.1 expone el proceso global de creación de un IVE centrándose en la integración y gestión del entorno por parte del *Manager*.

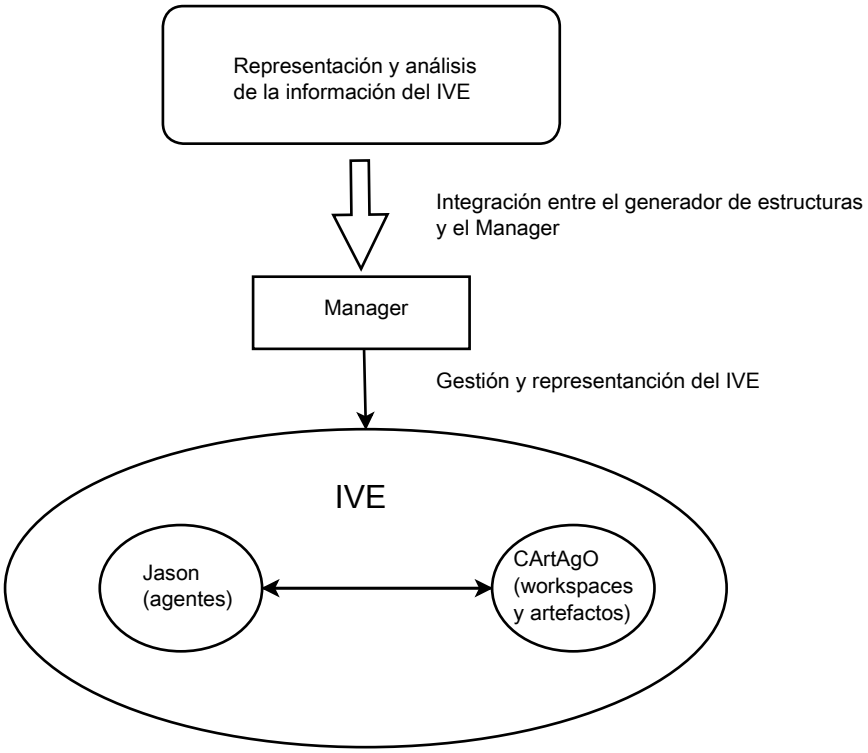

Figura 3.1: Proceso de integración y gestión del IVE

[1] http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/junosr2

3.2. Especificación de un documento XSD para la definición de IVEs

3.2.1. Ontología del modelo MAM5

Un IVE complejo, generalmente, posee una cantidad de elementos bastante extensa. Esto genera cierta problemática a la hora de plasmar los elementos en el archivo, sobretodo si se realiza manualmente. Es por esto que se decidió utilizar una herramienta de edición de archivos XML (editor por defecto de Eclipse *Eclipse XML Editors and Tools*) que permitiera al diseñador obtener una estructura básica del archivo a partir de un documento XSD. Lo primero que realizaría el diseñador de IVEs, a partir de esta estructura, sería rellenar aquellos datos de los elementos que vayan a aparecer en el entorno.

El utilizar una herramienta de edición y documentos XSD ayuda a analizar sintácticamente los archivos XML impidiendo escribir etiquetas incorrectas o erróneas. El análisis sintáctico se trata en profundidad en la sección 3.2.2 de esta memoria.

La estructura del archivo XSD toma como base el meta-modelo MAM5, englobándolo y extendiéndolo. Esta estructura, aunque muy similar, no coincide totalmente con la expuesta en MAM5 como se verá posteriormente.

La figura 3.2 expone la jerarquía de los elementos en el archivo XSD profundizando en algunos elementos más de como aparecen en el meta-modelo MAM5 2.1. Cada elemento se asocia con una etiqueta XML pero su orden es diferente que en el archivo, en este caso solo se muestra su relación a nivel jerárquico dentro del meta-modelo MAM5 y del entorno en sí.

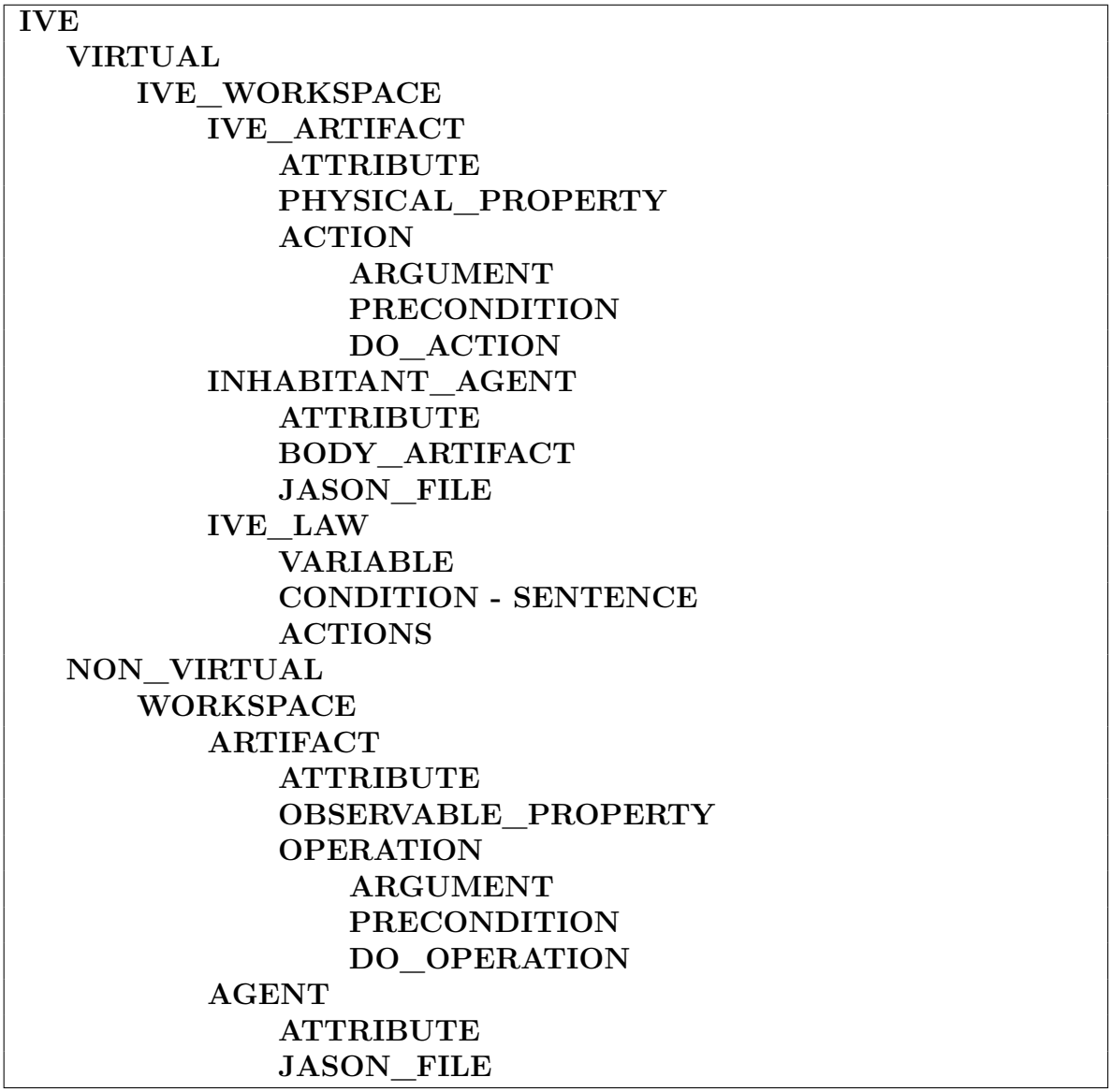

Figura 3.2: Ontología del archivo XSD

Dentro de cada archivo XML aparecerán estas etiquetas acompañadas de una serie de atributos XML (variables de etiquetas) que permiten definir las propiedades básicas de cada elemento, como el nombre, el tipo de una propiedad o cualquier identificador de otro elemento.

A continuación se realiza una pequeña descripción de cada concepto con los atributos XML que se necesiten, así como algún pequeño ejemplo de códi-

go.

- ***IVE*:** La especificación de un IVE contiene un atributo XML para el nombre del entorno virtual inteligente. Esta etiqueta es la primera que aparece y contiene a todas las demás, abre y cierra el archivo XML.

- ***IVE_WORKSPACE*:** En la definición de un *IVE workspace* debe aparecer el nombre del *IVE workspace*, así como tres listas con los nombres de: los *IVE artifacts*, los *Inhabitant agents* y las *IVE laws* que contiene. Cada artefacto, agente o ley diferente tendrá una definición posterior en el archivo.
  En un *IVE workspace*, además, pueden aparecer diferentes artefactos o agentes que posean una misma descripción o definición (si el entorno simulara, por ejemplo, una guerra, deberían poder existir varios artefactos "pistola" y varios agentes de tipo "soldado"), es por esto que la lista es sólo de nombres aunque posteriormente solo exista una definición por nombre de artefacto. El generador de estructuras se encarga de almacenar estos artefactos y agentes y de asignarles un identificador diferente a cada uno. Las *IVE laws*, en cambio, no pueden repetirse, es innecesario definir dos propiedades que haya de cumplir el entorno exactamente iguales.

  En la definición de un *IVE workspace* de trabajo es obligatorio incluir algún artefacto, algún agente y alguna ley.

  En general, cuando se habla de una lista de elementos se representa como una lista de elementos ITEM:

```
<ITEM NAME=""/>
```

  La figura 3.3 muestra como se definiría un *IVE workspace*.

```
<IVE_WORKSPACE NAME="IveWorkspaceExample">
    <IVE_ARTIFACTS>
        <ITEM NAME="IveArtifact1"/>
        <ITEM NAME="IveArtifact2"/>
    </IVE_ARTIFACTS>
    <INHABITANT_AGENTS>
        <ITEM NAME="InhabitantAgent1"/>
    </INHABITANT_AGENTS>
    <IVE_LAWS>
        <ITEM NAME="IveLaw1"/>
    </IVE_LAWS>
</IVE_WORKSPACE>
```

Figura 3.3: Ejemplo de *IVE workspace*

- ***IVE_ARTIFACT*:** La etiqueta de un *IVE artifact* contiene su nombre y si se puede conectar a otro artefacto o no (LINKEABLE). Esta conexión (en inglés *linking*) se realiza a través de acciones y está implementada de la misma manera que en la herramienta CArtAgO. Establecer una conexión inicialmente en el XML es independiente de las conexiones que realice el *Manager* dinámicamente en tiempo de ejecución, por tanto, esta conexión es estática y puede servir para crear artefactos que, por definición, deban estar conectados permanentemente.

  Un *IVE artifact* posee una lista de atributos (ATTRIBUTE), una lista de propiedades físicas (PHYSICAL_PROPERTIES) y una lista de acciones (ACTIONS).

  Como se ha comentado anteriormente, la definición de un artefacto es única. Podremos indicar, por ejemplo, que en un *IVE workspace* aparezcan tres artefactos de tipo "silla", pero solo existirá una definición de "silla"dentro del archivo XML. Al tener varios artefactos de un mismo tipo la misma definición, los valores de sus variables (atributos y propiedades físicas) serán los mismos inicialmente. En el caso de que se necesitara modificar alguna de estas variables antes de lanzar la simulación (por ejemplo, si no queremos que estén en la misma posición) sería tarea del *Manager* asignar los valores que se deseen para cada artefacto. La siguiente figura (3.4) ilustra la definición de un *IVE artifact* sin profundizar en sus elementos.

```
<IVE_ARTIFACT NAME="IveArtifact1" LINKEABLE="true">
    <ATTRIBUTES>
        ...
    </ATTRIBUTES>
    <PHYSICAL_PROPERTIES>
        ...
    </PHYSICAL_PROPERTIES>
    <ACTIONS>
        <ACTION NAME="action1">
            ...
        </ACTION>
        <ACTION NAME="action2">
            ...
        </ACTION>
    </ACTIONS>
</IVE_ARTIFACT>
```

Figura 3.4: Extracto de un *IVE artifact*

- ***ATTRIBUTE*:** Este elemento no aparece sólamente dentro de los artefactos (tanto virtuales como no virtuales), si no también dentro de los *Inhabitant agents* y de los *Agents*.

  Un *Attribute* contiene un nombre y un tipo como atributos XML, y, entre etiquetas, el valor que posee. Están asociados a variables simples y pueden ser: enteros, floats, doubles, cadenas de carácteres o booleanos. Los *Attributes* no aparecen en el meta-modelo MAM5 original y su función es, simplemente, la de servir como variables auxiliares dentro de la clase de cada artefacto o agente para usarse dentro de acciones/operaciones o funciones simples.

  Son diferentes a las propiedades físicas/observables de los artefactos en el sentido de que, simplemente, no son propiedades de éstos, si no variables internas que no tienen porqué tener un significado físico en el entorno. El uso de *Attributes* es totalmente opcional. La figura 3.5 muestra como sería un atributo booleano de nombre *booleanAttribute*.

```
<BOOLEAN NAME="booleanAttribute"> true </BOOLEAN>
```

Figura 3.5: Ejemplo de un *Attribute* booleano

- ***PHYSICAL_PROPERTY*:** Una especificación de una *physical property* debe poseer un nombre y un valor entre etiquetas. Pueden ser enteras, booleanas, floats, doubles, cadenas de carácteres o estructuras de tres variables de tipo double. Este último tipo, llamado VECTOR3D, ha sido creado para soportar la clase *Vector3D* que se usa en el caso de estudio CTF (Capture The Flag) de la asignatura de Ingenería Informática SIN (Sistemas Inteligentes) [26]. Estos vectores tendrán tres variables de nombres "x", "y" y "z"de tipo Double.

  Las *physical properties* se dividen en perceptibles o internas dependiendo de si otros artefactos del entorno pueden visualizar esas propiedades o no (un ejemplo sería la altura de un objeto, la altura es visible por otros elementos del entorno si se está viendo el objeto en cuestión).

  Como se ha comentado en la sección 3.2.1 de esta memoria, los *IVE artifacts* poseen una serie de propiedades físicas obligatorias que permiten al artefacto tener representanción en un mapa virtual. Las perceptibles por otros elementos son: *Position*, *Velocity* y *Orientation* de tipo *Vector3D*; *Shape* y *Sound* de tipo *String*; *Distance* y *Angle* de tipo *Double*. Las internas o no perceptibles por otros elementos son: *Acceleration* de tipo *Vector3D*; *Mass* y *Size* de tipo *Double*.

  Estas variables son comunes a cualquier *IVE artifact*. El diseñador, por tanto, debe darles un valor inicial en el archivo XML además de añadir las propiedades propias de la aplicación particular que se esté creando. En la figura 3.6 vemos la definición de una variable de tipo *Vector3D* y una de tipo entero.

```
<PHYSICAL_PROPERTIES>
    <PERCEIVABLE>
        <VECTOR3D NAME = "position">
            <DOUBLE NAME = "x"> 1.0 </DOUBLE>
            <DOUBLE NAME = "y"> 0.0 </DOUBLE>
            <DOUBLE NAME = "z"> 0.0 </DOUBLE>
        </VECTOR3D>
        ...
    </PERCEIVABLE>
    <INTERNAL>
       <DOUBLE NAME = "mass"> 500.0 </DOUBLE>
        ...
       <INTEGER NAME = "physicalPropertyExample"> 1 </INTEGER>
    </INTERNAL>
</PHYSICAL_PROPERTIES>
```

Figura 3.6: Ejemplo de una lista de *Physical properties*

- ***ACTION*:** La definición de una acción aparece dentro del *IVE artifact* que la posea. Debe tener un nombre, una serie de argumentos (ARGUMENTS), precondiciones (PRECONDITION), sentencias que realizen la acción (DO_ACTION) y un evento físico (PHYSICAL_EVENT) opcional. Las sentencias se ejecutarán dependiendo de si sus precondiciones asociadas se cumplen, modificando, entonces, propiedades de artefactos o elementos del entorno.

```
<ACTION NAME="actionExample">
    <ARGUMENTS>
        ...
     </ARGUMENTS>
    <PRECONDITION>
        ...
    </PRECONDITION>
    <DO_ACTION>
        ...
    </DO_ACTION>
    <PHYSICAL_EVENT NAME="actionPhysicalEvent">
        ...
    </PHYSICAL_EVENT>
</ACTION>
```

Figura 3.7: Estructura de una *Action*

La figura anterior (3.7) muestra los elementos que debe poseer una *action*.

Los atributos y propiedades físicas de un *IVE artifact* que vayan a utilizarse dentro de una acción se introducen mediante la etiqueta ELEMENT_ATT o ELEMENT_PROP, y deben tener el identificador del *IVE artifact* según el orden que tenga éste en el *IVE workspace* (etiqueta NAME) y el nombre de la variable en cuestión (etiqueta PROPERTY o ATTRIBUTE). Si la propiedad física o el atributo forman parte del artefacto que posee la acción el identificador se indica con la palabra SELF. Para acceder a una variable de tipo *Vector3D* debe indicarse el valor de la subvariable entre las etiquetas INDEX. Si, por ejemplo, se quisiera utilizar la subvariable x de la propiedad *position*, se realizaría de la siguiente manera:

```
<ELEMENT_PROP PROPERTY="position" NAME="SELF">
    <INDEX> "x" </INDEX>
</ELEMENT_PROP>
```

Una acción será única dentro de un artefacto y se traducirá en una @OPERATION en la clase del *IVE artifact* en CArtAgO.

- ***ARGUMENT*:** Los argumentos de una acción son variables que se introducirán cuando algún agente realice esa acción en el entorno de Jason. Existen dos tipos, parámetros y variables de retorno. Éstas últimas son las *OpFeedbackParam* de CArtAgO, variables que, después de ejecutarse la función, son devueltas con un valor asociado. Las dos etiquetas que puede tener un argumento en el archivo XML son PARAMETER y FEEDBACK (ver figura 3.8). Pueden ser de tipo INTEGER, BOOLEAN, FLOAT, DOUBLE o STRING.

```
<FEEDBACK NAME="feedbackArgument" TYPE="BOOLEAN" />
<PARAMETER NAME="parameterArgument" TYPE="INTEGER" />
```

Figura 3.8: Argumentos *Feedback* y *Parameter* en una *Action*

- ***PRECONDITION*:** Las precondiciones nos dicen cuando ejecutar las sentencias de una acción. Son variables conectadas por operadores booleanos que evalúan una condición. Estas variables pueden ser argumentos de la acción, propiedades físicas o atributos del artefacto o valores simples. Tienen un atributo XML asociado opcional OP que indica el operador booleano que ha de conectarla con la siguiente precondición (ver figura 3.9.

```
<PRECONDITION OP="and">
    <EQUAL>
        <OPERAND>
            <ELEMENT_PROP PROPERTY="position" NAME="SELF">
                <INDEX> "x" </INDEX>
            </ELEMENT_PROP>
        </OPERAND>
        <OPERAND>
            <PARAMETER NAME="parameter" TYPE="DOUBLE"/>
        </OPERAND>
    </EQUAL>
</PRECONDITION>

<PRECONDITION>
    <UNEQUAL>
        <OPERAND>
            <ELEMENT_ATT ATTRIBUTE="attribute" NAME="SELF"/>
        </OPERAND>
        <OPERAND>
            <PARAMETER NAME="parameter" TYPE="DOUBLE"/>
        </OPERAND>
    </UNEQUAL>
</PRECONDITION>
```

Figura 3.9: Extracto de dos *Preconditions* conectadas por una operación *And* en una *Action*

En pseudocódigo las anteriores precondiciones equivaldrían a: (this.position.x == parameter) && (this.attribute != parameter).

- ***DO_ACTION*:** Una DO_ACTION se asocia a una sentencia de ejecución. Estas sentencias siempre empiezan con la asignación a una variable, ya sea un argumento o cualquier otra propiedad o atributo. La sintaxis a nivel de XML es la misma que en las precondiciones.

Tanto para las DO_ACTION como para las PRECONDITION se tienen una serie de operadores matemáticos que ayudan a la evaluación y

ejecución de sentencias. Los operadores soportados son: EQUAL, UNEQUAL, GREATERTHAN, LESSTHAN, ADD, SUBSTRACT, MULTIPLY, DIVIDE, MOD, AND y OR.

Un ejemplo de la estructura de una acción podría ser la mostrada en la siguiente imagen:

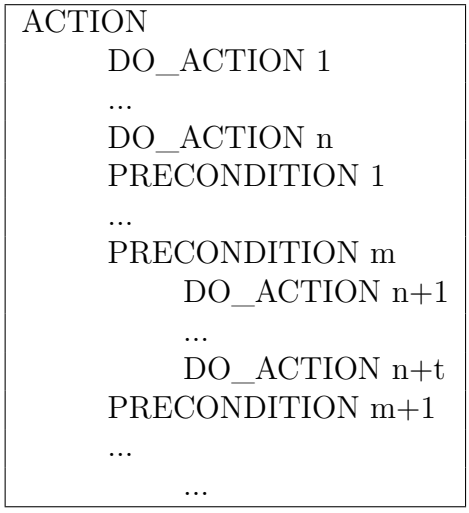

Figura 3.10: Estructura interna de una *Action* en el archivo XML

En este caso las DO_ACTION de 1 a n se ejecutarían sin ningún tipo de evaluación. En cambio las DO_ACTION de n+1 a n+t se ejecutarían cuando las PRECONDITION de 1 a m (conectadas entre sí con el operador OP que se incluya) se evaluasen a cierto.

De esta forma podemos conectar precondiciones entre sí para que se ejecuten una serie de sentencias o, simplemente, ejecutar sentencias sin necesidad de evaluar nada.

La figura 3.11 muestra una sentencia donde se asigna a un *feedback fall* el valor *true*.

```
<DO_ACTION>
    <ASSIGN>
        <OPERAND>
            <FEEDBACK NAME="fall" TYPE="BOOLEAN"/>
        </OPERAND>
        <OPERAND>
            <BOOLEAN_VAL> true </BOOLEAN_VAL>
        </OPERAND>
    </ASSIGN>
</DO_ACTION>
```

Figura 3.11: Ejemplo de una *Do action* dentro de una *Action*

A parte de realizar una asignación a una variable, en una DO_ACTION puede aparecer la ejecución de otra acción de otro *IVE artifact*. Esta ejecución realiza la conexión de un *IVE artifact* con otro. Si por ejemplo, hubieran dos artefactos conectados por una acción de movimiento, implicaría que al moverse un artefacto el otro se movería también. Esto se traduce en una conexión EXEC_LINKED_OP (ver figura 3.12) de la acción mover del primer artefacto con la acción mover del segundo. En el entorno de CArtAgO se hablaría de un *linking* a través de la sentencia *execLinkedOp* entre las dos acciones.

```
<DO_ACTION>
    <EXEC_LINKED_OP LINKED_ART_ID="1" LINKED_FUNCTION="linkedFunc">
        <LINKED_ARGUMENTS>
            <FEEDBACK NAME="feedback" TYPE="BOOLEAN"/>
            <ELEMENT_PROP PROPERTY="prop1" NAME="SELF"/>
        </LINKED_ARGUMENTS>
    </EXEC_LINKED_OP>
</DO_ACTION>
```

Figura 3.12: Ejemplo de una conexión estática dentro de una *Action*

En este ejemplo queremos que la función donde se encuentra la DO_ACTION ejecute la función *linkedFunction* del artefacto con id igual a 1 con los argumentos *feedback*, *prop1* y *prop2*.

- ***PHYSICAL_EVENT*:** Un evento físico necesita un nombre y una serie de parámetros opcionales. Estos eventos podrán ser observados por cualquier *Inhabitant agent* que se encuentre cercano, en el mapa virtual, al artefacto que haya realizado la acción asociada. En CArtAgO se traduce por una instrucción *signal*. Cuando un *Inhabitant agent* perciba una *signal* realizará el objetivo asociado al nombre del evento. Este objetivo deberá estar programado en el código Jason del agente. El nombre del evento físico, por tanto, indica el nombre de la función en Jason a ejecutar y los parámetros serán los argumentos de esta función.

  Los parámetros de un evento en el archivo XML tienen el mismo formato que los operandos de las acciones u operaciones: atributos o propiedades físicas de un artefacto, parámetros o *feedbacks* de una acción/operación y valores simples. Deben ir entre etiquetas de tipo PARAMETERS.

- ***INHABITANT_AGENT*:** La especificación de un *Inhabitant agent* necesita el nombre del agente. Además, cada *Inhabitant agent* debe poseer un *IVE artifact* que le sirva de cuerpo en el entorno virtual, en este caso se especifica el número (identificador) del artefacto en el *IVE workspace* a partir de la etiqueta BODY_ARTIFACT.

  Todo agente (tanto virtual como no virtual) debe tener un fichero .asl en código Jason que le permita realizar un razonamiento a base de objetivos. Esto se introduce con la etiqueta FILE, y debe contener el nombre del archivo. La siguiente imagen muestra un ejemplo de *Inhabitant agent* con un atributo y un artefacto como cuerpo.

```
<INHABITANT_AGENT NAME="agent">
    <ATTRIBUTES>
        <STRING NAME="stringAtt"> value </STRING>
    </ATTRIBUTES>
    <BODY_ARTIFACT>
        <ITEM ID="1"/>
    </BODY_ARTIFACT>
    <FILE NAME="agentJason.asl"/>
</INHABITANT_AGENT>
```

- ***IVE_LAW*:** Una *IVE law* ha de tener un nombre y puede tener dos tipos de definición. Para especificar una *IVE law* se puede establecer

Figura 3.13: Ejemplo de un *Inhabitant agent*

una variable con un valor o una condición asociada a una sentencia y, además, una serie de acciones a las que va a afectar la ley en concreto.

Si se quiere definir un elemento que se aplique al entorno bajo cualquier condición, se utilizará una variable con un valor con la etiqueta VARIABLE. Esta variable afectará a la lista de funciones que tenga asociada la *IVE law* cuando un *IVE artifact*, con esas acciones, se encuentre en el *IVE workspace* donde aparece esa ley (ver figura 3.14).

Otra forma de definir una *IVE law* es utilizando una condición, en este caso, para las acciones que tenga en la lista la ley, se evaluará primero una condición, y, cuando se cumpla, se ejecutará la sentencia que corresponda. En este caso se utilizan las etiquetas CONDITION y SENTENCE.

Un ejemplo de *IVE law* podría ser la densidad del medio (que variará, si, por ejemplo, un *IVE workspace* es un medio acuático) o cualquier ley física relacionada con la ejecución de acciones como la humedad o temperatura. A través de las condiciones-sentencias podemos definir que en un *IVE workspace* concreto no se puedan ejecutar una serie de acciones (por ejemplo, encender un fuego si estamos en un entorno húmedo o la velocidad de movimiento si la temperatura es muy baja).

```
<IVE_LAW NAME="mediumDensity">
    <DOUBLE NAME="density"> 1.0 </DOUBLE>
    <ACTIONS>
        <ITEM NAME="relatedAction"/>
    </ACTIONS>
</IVE_LAW>
```

Figura 3.14: Ejemplo de un *Ive law* de tipo variable

- ***WORKSPACE*:** En la definición de un *Workspace* debe aparecer el nombre del *workspace* y dos listas obligatorias con los nombres de: los *Artifacts* y los *Agents* que contiene. La estructura es la misma que para

definir un *IVE workspace*, se pueden establecer diversos artefactos o agentes del mismo tipo cuya definición se encontrará más adelante en el fichero XML. La figura 3.15 ilustra la definición de un *workspace*.

```
<WORKSPACE NAME="workspaceExample">
    <ARTIFACTS>
        <ITEM NAME="Artifact1"/>
    </ARTIFACTS>
    <AGENTS>
        <ITEM NAME="Agent1"/>
        <ITEM NAME="Agent2"/>
    </AGENTS>
</WORKSPACE>
```

Figura 3.15: Ejemplo de *workspace*

- ***ARTIFACT*:** La etiqueta de un *Artifact*, de forma similar a la de un *IVE artifact*, contiene su nombre y si se puede conectar a otro artefacto o no a través de alguna operación.
  Un *Artifact* posee una lista de atributos (ATTRIBUTE), una lista de propiedades observables (OBSERVABLE_PROPERTIES) y una lista de operaciones (OPERATIONS). La definición de un artefacto es única.

  La especificación de un *Artifact* es prácticamente la misma que un *IVE artifact*. En el ámbito de la definición la única diferencia es que, en este caso, no tenemos propidades físicas, si no, propiedades observables. Estas propiedades están directamente relacionadas con las *Observable properties* de CArtAgO y podrán ser observadas solamente por artefactos fuera del entorno de representanción, el mundo virtual.La figura 3.16 ilustra como sería la definición de un artefacto sin profundizar en sus elementos.

```
<ARTIFACT NAME="Artifact1" LINKEABLE="true">
    <ATTRIBUTES>
        ...
    </ATTRIBUTES>
    <OBSERVABLE_PROPERTIES>
        ...
    </OBSERVABLE_PROPERTIES>
    <OPERATIONS>
        <OPERATION NAME="operation1">
            ...
        </OPERATION>
        <OPERATION NAME="operation2">
            ...
        </OPERATION>
    </OPERATIONS>
</ARTIFACT>
```

Figura 3.16: Extracto de un *artifact*

- ***OBSERVABLE_PROPERTY*:** Las propiedades observables se especifican exactamente igual que las propiedades fisicas de los *IVE artifacts* teniendo en cuenta que no va a existir una separación entre aquellas perceptibles por otros agentes o aquellas internas. Todas podrán ser observables por otros artefactos en el workspace donde se encuentre.

- ***OPERATION*:** La definición de una operación aparece dentro del *Artifact* que la posea. Debe tener un nombre, una serie de argumentos (ARGUMENTS), precondiciones (PRECONDITIONS) y sentencias (DO_OPERATIONS). Todos estos elementos se definen exactamente igual que se definen para una acción (DO_ACTION pasa a ser DO_OPERATION por cuestión de sintaxis, su especificación es la misma). Al igual que ocurría con las *Observable properties* y *Physical properties*, la única diferencia entre *Actions* y *Operations* es que las primeras se ejecutan en el entorno virtual y las segundas en el entorno no virtual o no representable.

  Cada operación puede producir, tambíen, un evento (*Observable event*). El comportamiento es idéntico al los *Physical events* solo que los únicos que podrán percibir estos eventos observables son los agentes no

virtuales que se encuentren en el mismo *Workspace* donde ocurre el evento. Están totalmente separados de la parte virtual del entorno. Para su especificación se utiliza la etiqueta (OBSERVABLE_EVENT) dentro de una operación. Este tipo de eventos también se traducen en una función *signal* en CArtAgO.

- ***AGENT*:** Para un agente sin representanción virtual la definición se acorta en comparación con un *Inhabitant agent*. Como los agentes forman parte del entorno no virtual no necesitan un cuerpo o *Body artifact* que se asocie a su representanción visual. En este caso, por tanto, un agente deberá tener el nombre que lo defina y el nombre del archivo .asl de Jason donde esté implementado su razonamiento (etiqueta FILE). La figura 3.17 muestra la definición de un agente sin atributos.

```
<AGENT NAME="agentExample">
    <ATTRIBUTES/>
    <FILE NAME="agentExampleJason.asl"/>
</AGENT>
```

Figura 3.17: Ejemplo de un *Agent* sin atributos

Una vez definidos los elementos que se especifican en el documento XSD según el meta-modelo MAM5, se tratará en detalle en la siguiente subsección qué orden deben tener cada uno de estos elementos en las posibles instancias de IVEs en formato XML.

Estructura general del archivo XSD

La estructura general del archivo XSD está basada en la ontología definida por ELMS (Environment Description Language for Multi-Agent Simulation) [19]. Este lenguaje ha sido creado para el desarrollo de simulaciones basadas en agentes sociales (MAS-SOC) incorporando la especificación y ejecución de agentes cognitivos. Está íntimamente relacionado con el meta-modelo MAM5 ya que intenta establecer una serie de agentes con unas acciones dentro de un entorno definido por los recursos y objetos que se encuentren en él (en MAM5 artefactos). ELMS proporciona un archivo XML de inicialización del entorno, al igual que se está proporcionando en esta memoria para IVEs basados en

el meta-modelo MAM5.

En el archivo XSD, de entre los elementos principales: *workspaces*, artefactos y agentes, solo existe una definición por elemento único. Esto ahorra al diseñador el trabajo de definir varias veces un mismo elemento si quiere que éste aparezca más una vez en uno o más *workspaces*. De esta manera, reducimos la longitud y complejidad del archivo.

Es por esto que los *workspaces* (tanto virtuales como no virtuales) poseen listas de elementos sin definición. Esta definición se encuentra posteriormente en el archivo, y una vez leída, se completa la información, dentro del *workspace*, para ese elemento. Los únicos elementos definidos de esta manera son los *workspaces*, todos aquellos otros elementos dentro de agentes y artefactos (variables, acciones/operaciones, etc.) se definen de forma completa secuencialmente dentro de su elemento correspondiente. La siguiente imagen (3.18) muestra la estructura general establecida por el archivo XSD de la parte virtual.

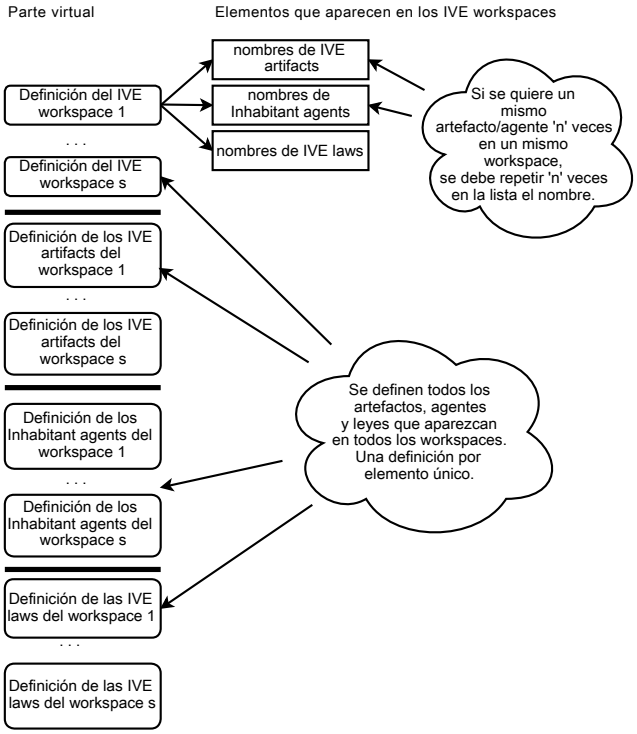

Figura 3.18: Estructura general de la parte virtual según el archivo XSD

La estructura de la parte no virtual es exactamente igual que la anterior, con la salvedad de que no existen *IVE laws*, los *IVE artifacts* serían *artifacts* y los *Inhabitant agents* simples *agents*.

La definición del archivo XSD extiende el meta-modelo MAM5 con el uso de los *Attributes*, como se aprecia en la imagen 3.19.

Una vez se ha expuesto la estructura del archivo XSD basado en el modelo MAM5 se procede a explicar una de las características que ofrece el utilizar este tipo de documentos para la edición de archivos XML, el análisis sintático.

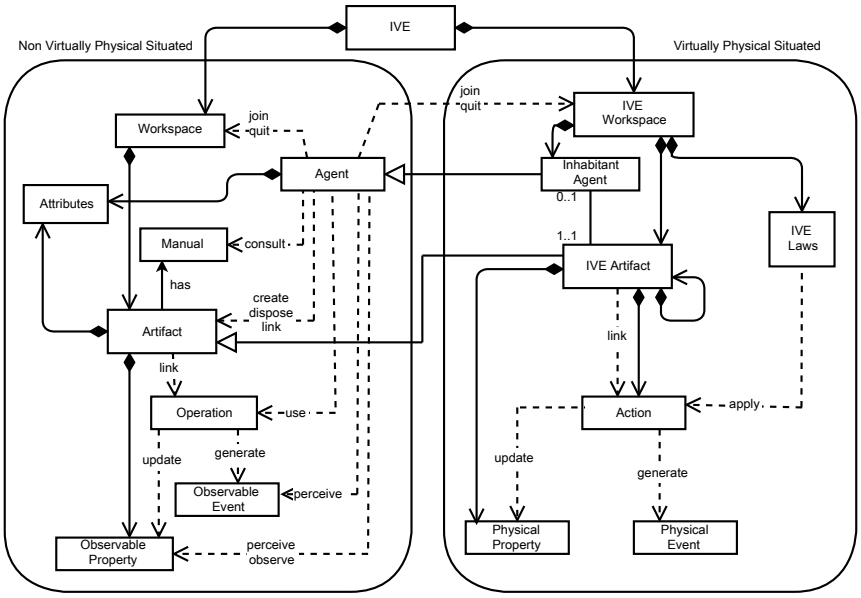

Figura 3.19: Meta-modelo MAM5 extendido

3.2.2. Análisis sintáctico

La creación de los archivos XML utilizan como base un documento esquema XSD (*XML Schema definition*). Con un documento XSD se define el orden y la sintaxis de las etiquetas requeridas por el XML y permite validar, respecto a su estructura, cualquier archivo.

El diseñador de entornos, por tanto, debería poder generar y validar el archivo XML del IVE que desee definir a partir del documento XSD antes

de enviárselo al proceso *Manager*.

El análisis sintáctico que proporciona el documento XSD a través del editor posee la siguientes características:

- Permite validar el nombre y el orden de las etiquetas y los atributos XML. Existe, además, una opción de autocompletado de etiquetas (ver figura 3.20).
- Señala como error los valores entre etiquetas de tipo incorrecto (una etiqueta de tipo STRING_VAL, por ejemplo, solo puede poseer como valor una cadena de carácteres).
- Indica aquellos atributos XML que son obligatorios. Por ejemplo, el nombre de cualquier elemento (NAME) o el tipo de las variables (TYPE). Los únicos atributos XML opcionales de todo el documento son los operadores de conexión de precondiciones (OP) y los identificadores de agente en un evento físico u observable (AGENT_ID).
- Advierte sobre etiquetas obligatorias que puedan no aparecer. Por ejemplo, cualquier *IVE workspace* debe poseer algún artefacto, agente o ley; los agentes deben poseer un fichero Jason; los *Inhabitant agents* siempre deben poseer un artefacto como cuerpo; los artefactos deben tener propiedades (físicas u observables) y las acciones/operaciones sentencias DO_ACTION o DO_OPERATION. Se pueden ver qué elementos son obligatorios en el archivo XSD.

El documento XSD permite, además, la generación automática de plantillas, que, en este caso, contendrán los elementos básicos para la creación de un IVE basado en MAM5. Para esta generación, una vez creado el archivo XSD, se pulsa con botón derecho y se selecciona *Generate XML file*.

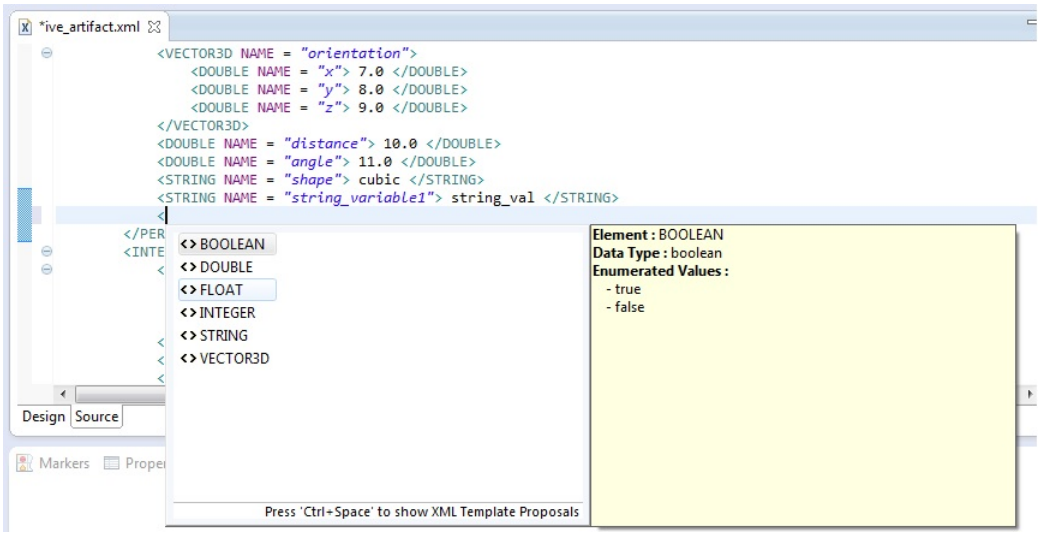

Figura 3.20: Función de autocompletado a partir de la estructura del archivo XSD

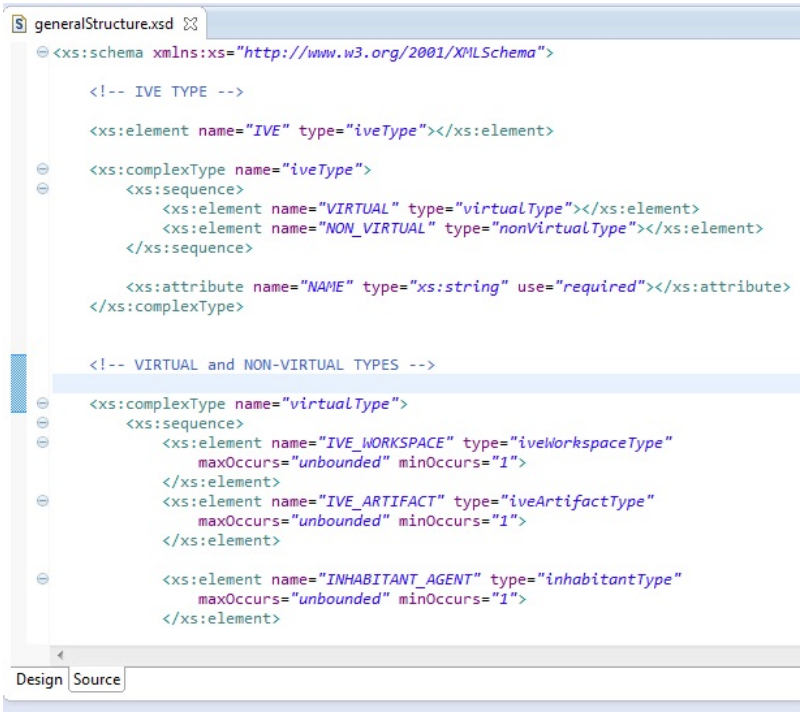

Figura 3.21: Extracto del archivo *generalStructure.xsd*

La herramienta utilizada ha sido el editor XML por defecto de Eclipse

(Eclipse XML Editors and Tools). Este editor permite modificar el archivo XSD directamente a nivel de código o a través de una interfaz gráfica.

El archivo XSD base de este proyecto se llama *generalStructure.xsd* (ver figuras 3.21 y 3.22) y se puede encontrar en el anexo de esta memoria.

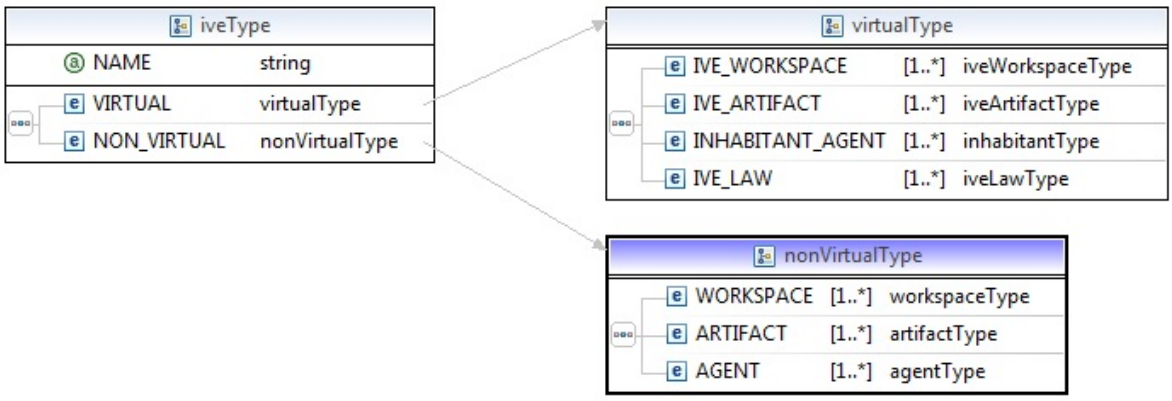

Figura 3.22: Extracto del archivo generalStructure.xsd desde el editor gráfico

3.2.3. Conclusiones

En esta parte de la memoria se ha presentado una estructura de un archivo XSD que permite definir y especificar entornos virtuales inteligentes basados en el meta-modelo MAM5. Como se ha podido comprobar, esta estructura no es fiel al cien por cien al meta-modelo que se especifica en MAM5 sino que lo engloba y extiende.

A diferencia del meta-modelo, laps *workspaces* y los *IVE workspaces* se definen al mismo nivel que los artefactos, agentes y leyes, todos cuelgan de la etiqueta VIRTUAL o NON_VIRTUAL, según corresponda. Esto permite establecer en un primer momento qué elementos van a aparecer en cada *workspace*, y, luego, qué definición poseen estos elementos. La ventaja de este método es que nos ahorra el tener que definir más de una vez un mismo elemento que vaya a aparecer en *workspaces* diferentes o varias veces en uno mismo. La desventaja aparece en el momento que el diseñador quiera establecer en el mapa virtual varios artefactos iguales con valores de propiedades y atributos diferentes, en este caso quedaría la opción de crear artefactos con nombre diferente (i.e. crear otro tipo de artefacto) o dejar para el *Manager*

la opción de modificar valores de variables antes de lanzar a ejecución la representación virtual.

3.3. Generación de estructuras de datos a partir del archivo XML

La generación de estructuras de datos es el segundo paso de la inicialización del IVE. Primero, se genera el archivo XML y, luego, se pasa como argumento al proceso *Manager*. El proceso que genera las estructuras de datos será ejecutado por el *Manager*. Este proceso posee una clase principal, llamada *Parser* que se encarga de la lectura de modelos o instancias de entornos generados a través del documento XSD y que se traducen en documentos XML. Esta clase, a través de su función principal, *parseXML*, almacena todas las estructuras de datos extraídas del XML, realiza un análisis semántico y crea el código necesario para el *Manager* en CArtAgO durante su ejecución.

La clase *Parser* realiza, por tanto, tres funciones diferentes:

- Generación de estructuras de datos. Su objetivo principal es transformar la entrada en formato XML a estructuras de datos entendibles por el proceso *Manager*.
- La compilación de las estructuras de datos que almacenan artefactos (virtuales y no virtuales) a clases Java que cumplan lo establecido en CArtAgO y en el meta-modelo MAM5 con respecto a la teoría de A&A.
- El análisis semántico. Este punto se realiza durante la lectura del XML y sirve para informar sobre posibles errores semánticos dentro del archivo.

3.3.1. Proceso generador de estructuras

Como se ha comentado anteriormente, la clase *Parser* es la clase principal creada para realizar tanto la generación de estructuras como el análisis semántico. Esta clase hereda de la clase *DefaultHandler*, implementada en SAX (Simple API for XML), que permite recorrer el archivo XML utilizando eventos.

En el estado del arte de esta memoria se ha definido el funcionamiento principal de SAX. Esta librería de análisis sintáctico de XML crea un evento cada vez que encuentra:

- Una etiqueta de inicio (función *startElement*).
- Una etiqueta de fin (función *endElement*).

- Un valor entre etiquetas (función *characters*).
- El inicio del archivo XML (función *startDocument*).
- El final del archivo XML (función *endDocument*).

Al utilizar una clase que hereda de *DefaultHandler* las funciones anteriores han de implementarse según el criterio de análisis que se quiera establecer. SAX, por tanto, solamente nos da funciones que se ejecutan cuando suceden una serie de eventos, quedándo en la parte del programador el realizar el análisis según estos eventos.

La estructura general de la clase *Parser* queda condicionada por el manejo de los eventos. En la función *startElement* se entra cada vez que se lee una etiqueta de inicio (por ejemplo <IVE_WORKSPACE> o <IVE_ARTIFACT>). Esta función se encarga de la inicialización del elemento correspondiente según etiqueta. Se tiene, por lo tanto, una función de lectura por cada etiqueta encontrada. Lo mismo ocurre con la función *endElement* pero con etiquetas de fin (del tipo </AGENT> o </ATTRIBUTE>).

Al realizarse el análisis secuencialmente con respecto al fichero XML, las estructuras de datos se van rellenando según se leen los elementos que aparezcan, sin posibilidad de volver atrás en el archivo. Esto aumenta la complejidad del análisis pero evita la pérdida de tiempo que se obtendría al recorrer varias veces todos los elementos del documento.

Existen dos partes bien diferenciadas en la clase *Parser*:

- El almacenamiento de la información de cada elemento del IVE en estructuras de datos (durante la lectura del archivo).
- La creación de las clases Java de los artefactos (al final de la lectura del archivo).

La creación de las clases se realiza al final de la lectura por si existen *IVE artifacts* o *artifacts* que puedan necesitar información de otros elementos del entorno. Por ejemplo si se realiza una conexión estática entre dos artefactos, donde un artefacto se conecta a otro que aperece posteriormente en el fichero o si la acción de un *IVE artifact* es afectada por una *IVE law*.

Clases de almacenamiento de elementos

Existen diversas clases que sirven de almacenamiento de los elementos del entorno. Estas clases son independientes de las que utiliza CArtAgO con respecto a artefactos y *workspaces*, son auxiliares y solo sirven para que el proceso *Manager* obtenga la información necesaria. A continuación se enumeran estas clases y se indican los principales componentes de cada una:

- **Parser**: clase principal de análisis. En esta clase se crean dos estructuras de datos que almacenan todos los *IVE workspaces* y *workspaces* diseñados en el XML, además guarda el nombre principal del IVE .
- **IveWorkspace**: existe una por cada *IVE workspace* definido en el XML. Cada *IVE workspace* guardará en tres estructuras de datos los *IVE artifacts*, *Inhabitant agents* e *IVE laws* que posea, tantos de cada uno como aparezcan en el XML.
- **Workspace**: esta clase es similar a la anterior pero en este caso solo almacena *Agents* y *Artifacts*.
- **IveArtifact**: la clase *IveArtifact* posee una estructura de datos por atributo, propiedad física y acción asociada según lo que se haya definido. Además guarda en variables si puede ser conectado o no por otro artefacto y el nombre de su puerto de salida. Este puerto de salida corresponde con la variable *outport* de CArtAgO. Que cada artefacto tenga un puerto es obligatorio en todo el entorno para realizar cualquier conexión en Jason de forma dinámica. Es decir, en cualquier momento un artefacto (que hayamos definido como *linkeable* en el XML) podrá ser conectado a otro durante la ejecución del *Manager*. En un *IveArtifact* guardamos también aquellos artefactos que tengan una conexión estática inicial con él.
- **Artifact**: en esta clase se tiene lo mismo que en la anterior con la diferencia de que estamos hablando de un elemento no virtual. En este caso, en vez de acciones, se tendrán operaciones y, en vez de propiedades físicas, propiedades observables. Al igual que para un *IVE artifact* podemos realizar conexiones estáticas y, por tanto, se guardan aquellos artefactos a los que se está conectado, además de el puerto de salida.
- **InhabitantAgent**: en un *Inhabitant agent* guardamos sus atributos, el nombre de su fichero Jason y el identificador del cuerpo artefacto que posea. En el caso de que tuviera más de un cuerpo (conectados por alguna acción) se guardarían todos los identificadores de sus artefactos cuerpo.

- **Agent**: la clase existente para un agente es la misma que para un *Inhabitant agent* solo que, en este caso, no existen ningún artefacto de tipo cuerpo ya que estamos tratando con un elemento no virtual.

- **IveLaw**: en una ley se tendrá la lista de acciones a las que afecta y una variable de tipo *IveSimpleVariable* o un par condición - sentencia de tipo *ActionSentence*.

- **Action/Operation**: las clases *Action* y *Operation* son estructuralmente iguales, difieren solo en su uso dentro del entorno (acciones para la parte virtual y operaciones para la no virtual). En una acción/-operación los principales elementos a almacenar son los argumentos y las sentencias. Las sentencias se guardan como listas de carácteres y representan precondiciones o sentencias de acción/operación (clase *ActionSentence*).

- **ActionSentence**: Las sentencias de acción/operación poseen una lista de carácteres con las variables y valores que indican la operación a realizar. Además se indica si la sentencia se ha de traducir en una precondición o en una operación. La asociatividad de los operandos que se haya establecido en el archivo XML se mantiene en la lista de carácteres de cada sentencia.

- **IvePhysicalProp**: esta clase es la que representa las propiedades físicas de cualquier *IVE artifact*. En una *IvePhysicalProp* se podrá establecer si es interna o perceptible y el valor introducido en el XML si no se trata de un *Vector3D*. En caso contrario poseerá una clase con tres subvariables con su tipo y nombre.

- **ObservableProp**: las propiedades observables de artefactos no virtuales se encuentran representadas por esta clase. En este caso, al igual que para las propiedades físicas, tendremos el tipo y valor de la variable o subvariables en caso de ser de tipo *Vector3D*.

- **IveSimpleVariable**: esta clase se encarga de soportar los atributos de los artefactos y agentes y los argumentos o *feedbacks* de las acciones y operaciones. Es más simple que las usadas para las propiedades físicas y observables y poseen un tipo y un valor.

En todos estas clases aparece siempre una variable identificadora y el nombre del elemento. El identificador simplemente enumera el orden de aparición de cada elemento dentro de donde esté incluido (por ejemplo, el primer artefacto de un *workspace* tendrá el identificador 0, el segundo el 1, etc.).

Cualquier uso de un identificador dentro del archivo XML debe hacer referencia a este tipo de identificadores.

Clases y funciones auxiliares

Dentro de la clase *Parser* se tienen dos funcionalidades principales: la primera, leer el archivo XML y almacenar su información y la segunda, crear las clases de los artefactos en CArtAgO.

De la primera funcionalidad se encarga la clase *ParsingFunctions*. Esta clase posee dos funciones de lectura de inicio y fin de etiquetas XML por elemento del entorno y, en algunos casos, una función de inicialización de las estructuras de datos. Es aquí donde se realiza el análisis semántico del archivo XML mediante el uso de la clase *Semantics*.

Existen otras funciones auxiliares que sirven de ayuda al diseñador y, a su vez, al proceso *Manager*:

- *PrintIVE*: Esta función imprime por pantalla todos los datos leídos por el analizador sintáctico. Sirve como validación de la información transmitida por el XML, se podrá ver, por tanto, qué información se ha leído y si es correcta.
- Funciones de obtención de datos: Estas funciones están, básicamente, enfocadas al proceso *Manager*. Permiten la obtención de la información de las estructuras de datos una vez leído el archivo XML (ver sección 3.4).

3.3.2. Creación de clases según CArtAgO y MAM5

La infraestructura CArtAgO posee una API propia de creación y gestión de artefactos a través de los *workspaces* donde éstos se encuentran situados. Esta API proporciona una clase para artefactos (*Artifact*) general con la funcionalidad básica que debe tener un artefacto con respecto a un entorno donde aparecen agentes. A partir de esta clase se pueden, por ejemplo, crear las propiedades observables de un artefacto, obtener y modificar su identificador dentro del entorno, generar eventos, etc. Inicialmente la clase *Artifact* no posee propiedades ni operaciones definidas, se hace, por tanto, necesario, el que cada tipo de artefacto tenga su propia clase con sus operaciones o propiedades ya especificadas.

Cualquier artefacto que aparezca definido en el archivo XML deberá tener su clase asociada extendiendo la clase *Artifact* original de CArtAgO. El objetivo de nuestro analizador sintáctico es generar estas clases dependiendo de las propiedades, operaciones y eventos que se hayan definido. Una vez estuviera la clase creada es trabajo del proceso *Manager*, dentro ya del SMA, el inicializar los artefactos a través de estas clases según los datos insertados en el XML. Estos datos los hallará a través de las estructuras de datos obtenidas del analizador sintáctico.

CArtAgO, además, está totalmente integrada con SMAs basados en Jason. Los agentes que formen parte de IVEs basados en MAM5 tendrán su código totalmente en Jason y gracias a las funciones que posee la clase *Artifact* podrán interactuar con los artefactos definidos en CArtAgO. De esta manera queda establecido el comportamiento del sistema. Una vez los artefactos hayan sido creados (en este caso tanto en lenguaje Jason directametne como con su versión en Java) los agentes podrán ejecutar acciones y operaciones de los artefactos a través de su código en Jason. El razonamiento cognitivo de los agentes, por tanto, depende de las acciones y operaciones de los artefactos del entorno.

En la sección 3.2.1 se hablaba sobre los elementos principales de MAM5. Se comentaba que un *IVE artifact* debía de tener una serie de propiedades físicas obligatorias para poder reproducirlo virtualmente en un entorno. Para poder realizar esto a través de CArtAgO se ha creado una nueva clase que hereda de *Artifact* llamada *IVEArtifact* (ver anexo IVEArtifact.java). En esta clase aquellas propiedades obligatorias que aparecen en todo artefacto virtual estan ya creadas con su tipo correspondiente (ver figura 3.23). Todos los artefactos virtuales heredarán de esta clase así como los artefactos no virtuales heredan de la clase *Artifact*. Es necesario que aparezcan en estas clases la importación de la librería de CArtAgO. Para poder utilizar toda la funcionalidad de CArtAgO integrado con Jason se utilizan los archivos *cartago.jar* y *c4jason.jar*.

```
class IVEArtifact extends Artifact {

        private ObsProperty position;
        private ObsProperty velocity;
        private ObsProperty orientation;
        private ObsProperty shape;
        private ObsProperty sound;
        private ObsProperty distance;
        private ObsProperty angle;
        private Vector3D acceleration;
        private double mass;
        private double size;
        ...
}
```

Figura 3.23: Extracto de declaración de variables en la clase *IVEArtifact*

La estructura de un artefacto que herede de la clase *Artifact* es muy similar a la de un artefacto virtual que herede de *IVEArtifact*. Básicamente tendremos los siguientes elementos:

- Declaraciones de atributos.
- Declaraciones de propiedades físicas/observables.
- Función de inicialización de atributos y propiedades físicas/observables.
- Acciones/Operaciones.
- Getters y setters de los atributos y propiedades físicas/observables.

Información del artefacto

El primer paso en la creación de la clase de un artefacto es establecer los puertos de conexión. Como se ha comentado en la sección 3.2.2 de esta memoria un artefacto puede conectarse dinámicamente en cualquier momento a otro. Esta conexión se realiza en el entorno de Jason utilizando un puerto. Todos los artefactos creados a partir de la clase *Parser* tendrán un puerto de la siguiente forma: " out-<nombre_artefacto>-Id<id_artefacto>".

Al nombre de la clase, además del nombre del artefacto, se añade la cadena *_class*.

Si hemos creado un artefacto de nombre *artifact1* el encabezado de la clase será de la siguiente manera:

```
@ARTIFACT_INFO(
outports = {
@OUTPORT(name = "out-artifact1-Id0")
}
) public class Artifact1_class ...
```

Atributos y propiedades físicas/observables

Una vez creado el encabezado de la clase se declaran los atributos y las propiedades físicas/observables. Para ello utilizamos la información almacenada a partir del XML declarando el tipo de la variable en cuestión. En este punto se puede proceder de dos formas:

- Declarando el tipo de las variables como *private ObsProperty*, el tipo de CArtAgO para las propiedades observables de los artefactos. Dentro de los artefactos no virtuales, por tanto, esta declaración se utiliza para las propiedades observables y dentro de los *IVE artifacts* se utilizará para las propiedades físicas perceptibles por otros artefactos. Una *ObsProperty* declara, en el entorno de CArtAgO, que un agente pueda observarla cuando se encuentra visualizando un artefacto.

- Declarando el tipo de las variables como *private <tipo_de_la_variable>*. Este caso se utiliza para cualquier atributo y para las propiedades físicas internas de un *IVE artifact*. No utilizamos *ObsProperty* porque no queremos que sean observables por otros agentes, son internas con respecto a la representación virtual. Los tipos en este caso pueden ser *int*, *boolean*, *float*, *double*, *String* o *Vector3D*.

Cogiendo como ejemplo el atributo de la figura 3.5 de la sección 3.2.2 se obtendría la siguiente declaración dentro de la clase:

```
private boolean booleanAttribute;
```

En el caso de una propiedad observable o física perceptible se cambiaría el tipo *boolean* por *ObsProperty*.

Función *void init*

Cada clase de un artefacto debe tener una método de incialización llamado *init*. Este método representa el constructor de la clase del artefacto y se utiliza en el momento de crearlo.

Dependiendo del tipo de propiedad física/observable o atributo la inicialización se realiza de una manera concreta. Teniendo en cuenta que a la función *init* se le pasarán como argumentos los valores iniciales correspondientes a cada propiedad o atributo del artefacto la inicialización quedaría de la siguiente manera:

- Atributos:

  this.atributo = valorAtributo;

  En este caso se está tratando con variables simples, por tanto, simplemente se establece que el atributo de dentro de la clase sea igual que el pasado como argumento al método init.

- Propiedades físicas internas:

  Para propiedades que no sean del tipo Vector3D se hace exactamente igual que para los atributos . En caso de ser de tipo Vector3D la inicialización sería de la siguiente manera.

  this.propiedad =
  new Vector3D(valorProp_x, valorProp_y, valorProp_z);

  En el método *init* para una variable Vector3D se han de pasar como argumento los valores de las tres subvariables.

- Propiedades físicas perceptibles y propiedades observables:

  En este caso hemos de inicializar la variable como si fuera una *ObsProperty*. Para ello existen dos pasos, primero definir una *ObsProperty* genérica que tenga por nombre y valor el nombre y el valor de la propiedad que se esté tratando, y, segundo, asignar a la propiedad declarada dentro del artefacto esta nueva *ObsPropery*.

  defineObsProperty("propiedad", valorPropiedad);

  this.propiedad = getObsProperty("propiedad");

En CArtAgO se puede definir una propiedad física con más de un valor, lo que permite definir propiedades físicas perceptibles y observables de tipo Vector3D:

defineObsProperty("propiedad", valorProp_x, valorProp_y, valorProp_z);

En los *IVE artifacts* al tener ya definidas en la clase padre las propiedades físicas obligatorias que debe poseer cualquier artefacto virtual se hace innecesario el inicializarlas individualmente. En estos casos en cada función *init* se llamará al método de inicialización de la clase IVEArtifact con los valores correspondientes de las propiedades obligatorias:

super.init(propiedades_fisicas);

Funciones de lectura y escritura de variables

Al definir los atributos y las propiedades físicas y observables de artefactos como privadas se hace necesario el uso de funciones de lectura y escritura de estas variables (funciones *get* y *set*). Las funciones de lectura devolverán directamente el tipo original de la variable, por ejemplo, en el caso de las perceptibles no se devolvería una *ObsProperty* si no su tipo tal como aparece definido en el archivo XML. Estas funciones no son operaciones ni acciones y, por tanto, no podrán ser ejecutadas dentro del entorno de Jason. Su función es la de tratar con las variables propias de los artefactos desde Java.

Acciones/Operaciones

A la hora de crear una acción/operación a partir del XML aparecen dos cuestiones importantes:

- Qué tipo de acción/operación es según CArtAgO. Existen diversos tipos en CArtAgO pero en este caso solo tenemos en cuenta dos: @OPERATION y @LINK. Si una acción/operación ejecuta a otra mediante una conexión, en esta segunda acción/operación debería aparecer la etiqueta @LINK, si no, simplemente la etiquetamos como @OPERATION. Solamente las funciones que tengan la etiqueta @OPERATION o @LINK pueden ser ejecutadas por agentes en el entorno de Jason.
- Cómo son los argumentos de la acción/operación. Como hemos visto en la sección 3.2.2 un argumento puede ser un parámetro de entrada

(PARAMETER) o de salida (FEEDBACK). Los parámetros de entrada se definen como cualquier variable en Java, primero el tipo y luego el nombre. Los *feedback* en cambio deben poseer el tipo *OpFeedbackParam<tipo>* de CArtAgO.

Usando Jason y CArtAgO los parámetros de salida de una acción/operación siempre serán del tipo *OpFeedbackParam*, nunca hará falta que la función retorne un valor y, por tanto, las acciones/operaciones serán siempre de tipo *void*.

Cogiendo de ejemplo la figura 3.8 de la sección 3.2.2, se generaría la siguiente acción en CArtAgO:

```
@OPERATION
void accionEjemplo(OpFeedbackParam<Boolean> feedbackArgument,
        int parameterArgument) {
       ...
}
```

Una vez creados el tipo de la acción y sus argumentos solamente quedaría traducir las precondiciones y las sentencias de acción/operación. En la estructura de datos que almacena el analizador este tipo de sentencias se almacenan como una cadena de carácteres y, además, se mantiene siempre la asociatividad de las operaciones matemáticas con respecto al XML. En este punto, por tanto, se imprime directamente la cadena en la función.

Hay que tener en cuenta que todas aquellas propiedades físicas u observables y atributos se escriben y leen a través de las funciones *get* y *set* explicadas en el punto anterior. Los parámetros se utilizarían directamente ya que son internos a la función y los *feedback* a través de los métodos *get* y *set* de la clase *OpFeedbackParam*.

Si tomamos como ejemplos las figuras 3.9 y 3.11 de la sección 3.2.2 y suponemos que van una detrás de otra en el archivo XML, la traducción a CArtAgO quedaría de la siguiente manera:

```
@OPERATION
void accionEjemplo(...) {
       try {
              ...
              if((getPositionValue().x == parameter)
                     && (getAttribute() != parameter)) {
                     fall.set(true);
              }
              ...
       } catch (Exception ex) {
              ex.printStackTrace();
       }
}
```

Para obtener una mejor visión de lo explicado se pueden ver diversos ejemplos, del archivo XML y de las posteriores clases creadas, en la sección 4 de esta memoria.

3.3.3. Análisis semántico

Una de las características principales de la clase *Parser* desarrollada es que realiza un análisis semántico del archivo XML. En un compilador la fase de análisis semántico utiliza como entrada el programa fuente revisado por el analizador sintáctico para encontrar restricciones de tipo y preparar la generación de código.

El análisis semántico que se realiza en este caso es similar al proceso que realiza un compilador. Primero, el programa fuente, es decir, el archivo XML, se analiza sintácticamente (a través de la herramienta de edición de archivos XML) para que no tenga errores de etiquetado, según el meta-modelo MAM5. Una vez realizado este análisis y, al enviar el archivo XML correcto al proceso *Manager*, la clase *Parser* guardará la información de los elementos en estructuras de datos y realizará, a su vez, a partir de la clase *Semantics*, el análisis semántico del archivo.

Por cada uno de los posibles errores de semántica que puedan aparecer existirá una función que se encargue de generar un evento del tipo *IllegalStateException* en Java. Estas funciones, al ejecutarse, pararán completamente el análisis del documento indicando donde se halla el error dentro del archivo

XML.

A continuación se exponen aquellos errores semánticos que es capaz de detectar la clase *Semantics*.

Errores en la especificación de *workspaces* e *IVE workspaces*

- Utilizar algún *IVE artifact*, *artifact*, *inhabitant agent*, *agent* o *IVE law* en la definición de un *IVE workspace* o *workspace* que no esté especificado posteriormente en el archivo.
- Definir más de un *IVE workspace* o *workspace* con el mismo nombre.

Errores en la especificación de *IVE artifacts* y *artifacts*

- Definir más de un *IVE artifact* o *artifact* con el mismo nombre.
- Definir más de un *attribute* con el mismo nombre en un *IVE artifact* o *artifact*.
- Definir más de una *physical property* con el mismo nombre en un *IVE artifact*.
- Definir más de una *observable property* con el mismo nombre en un *artifact*.
- Definir más de una *action* con el mismo nombre en un *IVE artifact*.
- Definir más de una *operation* con el mismo nombre en un *artifact*.

Errores en la especificación de *actions* y *operations*

- Definir más de un argumento (*parameter* o *feedback*) con el mismo nombre.
- Utilizar un argumento (*parameter* o *feedback*) que no esté definido en la *action*/*operation*.
- Utilizar un *IVE artifact* que no esté definido en el archivo en la conexión de una *action* con otra.
- Utilizar un *artifact* que no esté definido en el archivo en la conexión de una *operation* con otra.

- Conectar una *action/operation* a otra que no exista en el *IVE artifact* o *artifact* indicado.
- Utilizar una *physical property* o un *attribute*, tanto en la *action/operation* en sí como en una conexión, con un identificador de *IVE artifact* o *artifact* que no exista en el XML.
- Utilizar una *physical property* o un *attribute*, tanto en la *action/operation* en sí como en una conexión, que no exista en el *IVE artifact* o *artifact* identificado.

Errores en la especificación de *Inhabitant agents* y *agents*

- Definir más de un *Inhabitant agent* o *agent* con el mismo nombre.
- Definir más de un *attribute* con el mismo nombre en un *Inhabitant agent* o *agent*.
- Utilizar como *body artifact* de un *Inhabitant agent* un *IVE artifact* que no esté definido en el archivo.

Errores en la especificación de *IVE laws*

- Definir más de un *IVE law* con el mismo nombre.
- Utilizar en un *IVE workspace* más de una *IVE law* con el mismo nombre.

Errores generales

- Ningún atributo XML de tipo NAME, LINKEABLE, ATTRIBUTE, TYPE, PROPERTY, LINKED_ART_ID o LINKED_FUNCTION debe estar vacío.
- Cualquier *physical* o *observable property* de tipo *Vector3D* debe tener tres subvariables de nombres "x", "y" y "z" que sean de tipo DOUBLE.

3.3.4. Conclusiones

En este capítulo se ha expuesto la implementación de un generador de estructuras capaz de realizar análisis semánticos de archivos XML cuya estructura sea la definida en la sección 3. De esta manera completamos la funcionalidad previa al proceso *Manager* de creación y análisis de un IVE.

Para poder realizar el análisis se han creado una serie de clases que representan los posibles elementos de un entorno virtual según MAM5. Estas clases han sido implementadas para poder almacenar, por cada elemento, toda la información que aparece en el archivo XML. Esta información, sin embargo, se limita solo al archivo, no se contemplan aquellos datos que puedan poseer las clases de CArtAgO y MAM5 de artefactos y agentes una vez se esté realizado la ejecución del entorno. Es trabajo del *Manager* el coger aquella información relevante, en cada momento, de las clases del analizador y, utilizarla para gestionar el entorno.

Tanto el archivo XML como el analizador trabajan con identificadores para tratar con los elementos. Estos identificadores siempre forman parte del ámbito de lo definido en el archivo y, por tanto, no tienen porque coincidir con aquellos usados en la ejecución del entorno por parte de Jason y CArtAgO. Es trabajo del diseñador el utilizar correctamente estos identificadores, por ejemplo, en el caso de que se quiera realizar una conexión estática y trabajo del *Manager* el establecer una relación entre estos identificadores y los usados en Jason para poder saber con qué elemento estamos tratando en el momento de la ejecución del IVE.

3.4. Integración con el proceso *Manager*

En esta sección se abarcará la utilización de la clase *Parser* por parte del *Manager* enunciando aquellas peculiaridades necesarias para ejecutar correctamente del entorno. Además, se realizará un pequeño análisis de las funciones generales que debería realizar el *Manager* para gestionar el IVE.

El proceso de inicialización se entiende como paso previo a la creación, gestión y representación del entorno por parte del *Manager*. Esta inicialización la realiza el diseñador a través de la herramienta de edición y validación presentada en esta memoria. Sin embargo, debe existir en algún momento una integración del tratamiento del archivo XML (clase *Parser*) con su posterior ejecución como entorno virtual (*Manager*).

La primera tarea que realiza el proceso *Manager* es la lectura del archivo XML como argumento. El proceso crea e inicializa una variable de tipo *Parser* y ejecuta su método *parseXML* con el nombre del documento como parámetro. En este momento se realiza el análisis. La variable creada poseerá la información del XML dentro de sus estructuras de datos y se habrán creado

las clases necesarias de los artefactos. A partir de aquí se empieza a ejecutar la representanción y gestión del IVE.

El *Manager* posee toda la información introducida en el archivo XML. Consultando esta información el proceso inicialmente crea los agentes y *workspaces* virtuales y no virtuales. Una vez creados los agentes se ejecuta el código en Jason de cada uno. Los *Inhabitant agents* deben consultar con el *Manager* el identificador de su *body artifact* antes de realizar cualquier acción en el entorno. Cuando un agente realiza una acción u operación, el *Manager* debe ser informado para que pueda admitir o no su ejecución.

De este modo, el ciclo de vida en la especificación de un IVE queda cerrado y sería:

1. Creación del archivo XML, según el IVE que se quiera definir, apoyándose en el documento XSD para que sea correcto léxicamente.
2. Lectura del archivo por parte de la clase *Parser* a través de su ejecución por parte del *Manager*.
3. Análisis semántico del contenido del archivo a través de la clase *Semantics* durante la lectura.
4. Almacenamiento de la información de cada elemento en su clase correspondiente.
5. Creación de las clases en CArtAgO y MAM5 de *IVE artifacts* y *artifacts*.
6. Lectura del *Manager* de las estructuras de datos.

Capítulo 4

Casos de estudio

A continuación se exponen una serie de ejemplos de aplicación con sus posteriores clases creadas. Estos ejemplos servirán de caso de estudio y servirán de validación del proceso de creación y de la gestión por parte del *Manager*. En cada caso puede que no se contemple el IVE de manera completa, el generador de estructuras puede leer la parte virtual o no virtual por separado si solo se incluye información de una de ellas. De esta manera se acorta considerablemente la longitud de los ejemplos para su mejor entendimiento.

4.1. Robot ápodo

4.1.1. Descripción

En este caso de estudio tenemos inicialmente, un robot cuyo cuerpo está formado por dos objetos, tal como ilustra la figura 4.1. Este robot puede ir aumentando de tamaño conforme vaya asimilando más objetos, pasando por encima de ellos, como se puede obsevar en la imagen 4.2. El robot además posee un movimiento similar al de un gusano o una serpiente pues no posee patas o ruedas para desplazarse, es por esto que recibe el nombre de robot ápodo.

4.1.2. Diseño del caso de estudio

El robot ápodo se ve representado por el *Inhabitant agent* llamado *Robot* que posee un cuerpo compuesto por dos artefactos conectados entre sí. Estos dos artefactos forman el cuerpo o *body artifact* del agente dentro de la representación virtual.

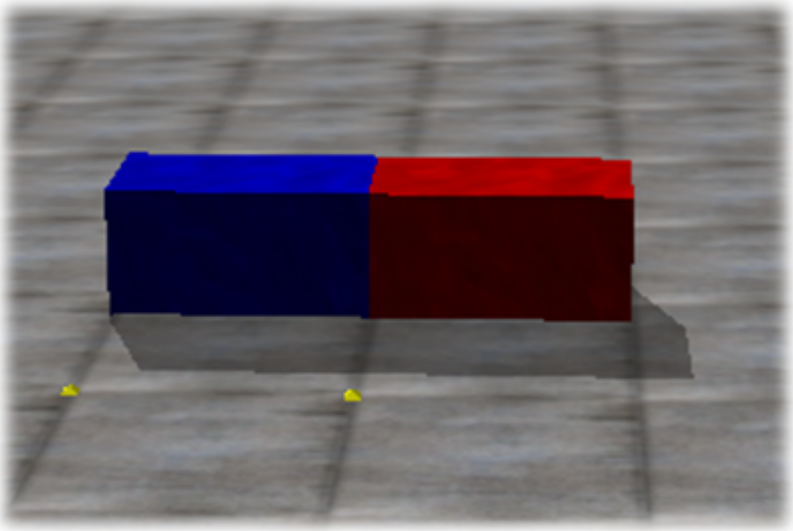

Figura 4.1: Detalle de la representación virtual del robot ápodo

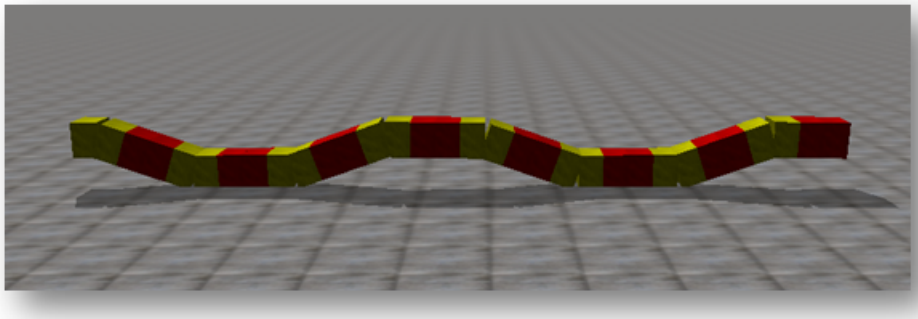

Figura 4.2: Robot ápodo en movimiento conectado a diversos *linkedArtifacts*

Este es un ejemplo simple de la creación de elementos en la parte virtual. El comportamiento del robot ápodo es simplemente evitar obstáculos en un mapa. Estos obstáculos serán artefactos totalmente estáticos. Además se plantea la posibilidad de que el robot se conecte a otros artefactos no considerados como obstáculos cuando pase por encima de ellos y que, por tanto, se conviertan en parte de su cuerpo.

Empezamos definiendo en el entorno los agentes y artefactos que quer-

amos situar virtualmente. En este caso introducimos los dos artefactos virtuales que formarán el cuerpo del agente principal (BodyRight y Bodyleft), un artefacto conectable al cuerpo del agente (linkedArtifact) y un artefacto estático que hará de obstáculo (unlinkedArtifact) al único *IVE workspace* que tendrá el entorno *apodRobot_workspace*.

```
<?xml version="1.0" encoding="UTF-8"?>
<IVE NAME="" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="generalStructure.xsd">

    <VIRTUAL>
        <IVE_WORKSPACE NAME="apodRobot_workspace">
            <IVE_ARTIFACTS>
                <ITEM NAME="BodyLeft"/>
                <ITEM NAME="BodyRight"/>
                <ITEM NAME="linkedArtifact"/>
                <ITEM NAME="unlinkedArtifact"/>
            </IVE_ARTIFACTS>
            <INHABITANT_AGENTS>
                <ITEM NAME="Robot"/>
            </INHABITANT_AGENTS>
            <IVE_LAWS>
                <ITEM NAME="Gravity"/>
            </IVE_LAWS>
        </IVE_WORKSPACE>
```

Una vez establecidos en lel *IVE workspace* todos los elementos, es hora de definirlos. Como tenemos cuatro tipos diferentes de artefactos realizaremos cuatro definiciones. A continuación se muestran todas las definiciones, explicando para el primer artefacto los elementos internos que posee la definición.

```
        <IVE_ARTIFACT NAME="BodyLeft" LINKEABLE="true">
            <ATTRIBUTES/>
            <PHYSICAL_PROPERTIES>
                <PERCEIVABLE>
                    <VECTOR3D NAME = "position">
                        <DOUBLE NAME = "x"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "z"> 0.0 </DOUBLE>
                    </VECTOR3D>
```

```
                <VECTOR3D NAME = "velocity">
                    <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "y"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "z"> 1.0 </DOUBLE>
                </VECTOR3D>
                <VECTOR3D NAME = "orientation">
                    <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                    <DOUBLE NAME = "z"> 0.0 </DOUBLE>
                </VECTOR3D>
                <DOUBLE NAME = "distance"> 5.0 </DOUBLE>
                <DOUBLE NAME = "angle"> 0.0 </DOUBLE>
                <STRING NAME = "shape"> cubic </STRING>
                <STRING NAME = "sound"> none </STRING>
            </PERCEIVABLE>
            <INTERNAL>
                <VECTOR3D NAME = "acceleration">
                    <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "y"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "z"> 1.0 </DOUBLE>
                </VECTOR3D>
                <DOUBLE NAME = "mass"> 20.0 </DOUBLE>
                <DOUBLE NAME = "size"> 5.0 </DOUBLE>
            </INTERNAL>
        </PHYSICAL_PROPERTIES>
```

Como podemos observar hemos definido las propiedades físicas que debe poseer todo *IVE artifact*. Al ser este elemento una parte del cuerpo del agente, debe tener una acción de movimiento. Además, lo consideramos como el cuerpo frontal, es el cuerpo que realiza el movimiento principal, luego tendrá que arrastrar al segundo cuerpo. Para ello se establece una conexión estática entre su acción de movimiento y la acción de movimiento del segundo cuerpo (BodyRig).

```
        <ACTIONS>
            <ACTION NAME="move">
                <ARGUMENTS>
                    <PARAMETER NAME="newX" TYPE="DOUBLE"/>
                    <PARAMETER NAME="newY" TYPE="DOUBLE"/>
                    <PARAMETER NAME="newZ" TYPE="DOUBLE"/>
                </ARGUMENTS>
```

```
                    <DO_ACTION>
                        <ASSIGN>
                            <OPERAND>
                                <ELEMENT_PROP NAME="SELF"
                                        PROPERTY="position">
                                    <INDEX> "x" </INDEX>
                                </ELEMENT_PROP>
                            </OPERAND>
                            <OPERAND>
                                <PARAMETER NAME="newX" TYPE="DOUBLE"/>
                            </OPERAND>
                        </ASSIGN>
                    </DO_ACTION>
                    <DO_ACTION>
                        <ASSIGN>
                            <OPERAND>
                                <ELEMENT_PROP NAME="SELF"
                                        PROPERTY="position">
                                    <INDEX> "y" </INDEX>
                                </ELEMENT_PROP>
                            </OPERAND>
                             <OPERAND>
                                <PARAMETER NAME="newY" TYPE="DOUBLE"/>
                            </OPERAND>
                        </ASSIGN>
                    </DO_ACTION>
                    <DO_ACTION>
                        <ASSIGN>
                            <OPERAND>
                                <ELEMENT_PROP NAME="SELF"
                                        PROPERTY="position">
                                    <INDEX> "z" </INDEX>
                                </ELEMENT_PROP>
                            </OPERAND>
                             <OPERAND>
                                <PARAMETER NAME="newZ" TYPE="DOUBLE"/>
                            </OPERAND>
                        </ASSIGN>
                    </DO_ACTION>
```

```
                    <DO_ACTION>
                        <EXEC_LINKED_OP LINKED_ART_ID="1"
                                LINKED_FUNCTION="move">
                            <LINKED_ARGUMENTS>
                                <ELEMENT_PROP NAME="SELF"
                                        PROPERTY="position">
                                    <INDEX> "x" </INDEX>
                                </ELEMENT_PROP>
                                <ELEMENT_PROP NAME="SELF"
                                        PROPERTY="position">
                                    <INDEX> "y" </INDEX>
                                </ELEMENT_PROP>
                                <ELEMENT_PROP NAME="SELF"
                                        PROPERTY="position">
                                    <INDEX> "z" </INDEX>
                                </ELEMENT_PROP>
                            </LINKED_ARGUMENTS>
                        </EXEC_LINKED_OP>
                    </DO_ACTION>
                    <PHYSICAL_EVENT NAME=""/>
                </ACTION>
            </ACTIONS>
        </IVE_ARTIFACT>
```

A continuación el artefacto BodyRight. Éste artefacto solo tiene una acción, la de movimiento, que aparecerá conectada a la de BodyLeft en su clase en CArtAgO. Inicialmente se encuentra situado al lado de BodyLeft.

```
        <IVE_ARTIFACT NAME="BodyRight" LINKEABLE="true">
            <ATTRIBUTES/>
            <PHYSICAL_PROPERTIES>
                <PERCEIVABLE>
                    <VECTOR3D NAME = "position">
                        <DOUBLE NAME = "x"> 5.0 </DOUBLE>
                        <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "z"> 0.0 </DOUBLE>
                    </VECTOR3D>
```

```
                <VECTOR3D NAME = "velocity">
                    <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "y"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "z"> 1.0 </DOUBLE>
                </VECTOR3D>
                <VECTOR3D NAME = "orientation">
                    <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                    <DOUBLE NAME = "z"> 0.0 </DOUBLE>
                </VECTOR3D>
                <DOUBLE NAME = "distance"> 5.0 </DOUBLE>
                <DOUBLE NAME = "angle"> 0.0 </DOUBLE>
                <STRING NAME = "shape"> cubic </STRING>
                <STRING NAME = "sound"> none </STRING>
            </PERCEIVABLE>
            <INTERNAL>
                <VECTOR3D NAME = "acceleration">
                    <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "y"> 1.0 </DOUBLE>
                    <DOUBLE NAME = "z"> 1.0 </DOUBLE>
                </VECTOR3D>
                <DOUBLE NAME = "mass"> 20.0 </DOUBLE>
                <DOUBLE NAME = "size"> 5.0 </DOUBLE>
            </INTERNAL>
        </PHYSICAL_PROPERTIES>
        <ACTIONS>
            <ACTION NAME="move">
                <ARGUMENTS>
                    <PARAMETER NAME="newX" TYPE="DOUBLE"/>
                    <PARAMETER NAME="newY" TYPE="DOUBLE"/>
                    <PARAMETER NAME="newZ" TYPE="DOUBLE"/>
                </ARGUMENTS>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <ELEMENT_PROP NAME="SELF"
                                    PROPERTY="position">
                                <INDEX> "x" </INDEX>
                            </ELEMENT_PROP>
```

```
                        </OPERAND>
                        <OPERAND>
                            <PARAMETER NAME="newX"
                                TYPE="DOUBLE"/>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <ELEMENT_PROP NAME="SELF"
                                PROPERTY="position">
                                <INDEX> "y" </INDEX>
                            </ELEMENT_PROP>
                        </OPERAND>
                        <OPERAND>
                            <PARAMETER NAME="newY"
                                TYPE="DOUBLE"/>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <ELEMENT_PROP NAME="SELF"
                                PROPERTY="position">
                                <INDEX> "z" </INDEX>
                            </ELEMENT_PROP>
                        </OPERAND>
                        <OPERAND>
                            <PARAMETER NAME="newZ"
                                TYPE="DOUBLE"/>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <PHYSICAL_EVENT NAME=""/>
            </ACTION>
        </ACTIONS>
    </IVE_ARTIFACT>
```

El siguiente artefacto, situado alejado del cuerpo del agente, debe dar la

oportunidad de ser conectado cuando el robot ápodo pase por su posición y además, poseer una acción de movimiento. Para ello establecemos la variable LINKEABLE a true. El proceso *Manager* será el encargado de realizar esta conexión dinámica durante la ejecución.

```
<IVE_ARTIFACT NAME="linkedArtifact" LINKEABLE="true">
    <ATTRIBUTES/>
    <PHYSICAL_PROPERTIES>
        <PERCEIVABLE>
            <VECTOR3D NAME = "position">
                <DOUBLE NAME = "x"> 10.0 </DOUBLE>
                <DOUBLE NAME = "y"> 10.0 </DOUBLE>
        <DOUBLE NAME = "z"> 10.0 </DOUBLE>
            </VECTOR3D>
            <VECTOR3D NAME = "velocity">
                <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                <DOUBLE NAME = "y"> 1.0 </DOUBLE>
                <DOUBLE NAME = "z"> 1.0 </DOUBLE>
            </VECTOR3D>
            <VECTOR3D NAME = "orientation">
                <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 0.0 </DOUBLE>
            </VECTOR3D>
            <DOUBLE NAME = "distance"> 5.0 </DOUBLE>
            <DOUBLE NAME = "angle"> 0.0 </DOUBLE>
            <STRING NAME = "shape"> cubic </STRING>
            <STRING NAME = "sound"> none </STRING>
        </PERCEIVABLE>
        <INTERNAL>
            <VECTOR3D NAME = "acceleration">
                <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                <DOUBLE NAME = "y"> 1.0 </DOUBLE>
                <DOUBLE NAME = "z"> 1.0 </DOUBLE>
            </VECTOR3D>
            <DOUBLE NAME = "mass"> 20.0 </DOUBLE>
            <DOUBLE NAME = "size"> 5.0 </DOUBLE>
        </INTERNAL>
    </PHYSICAL_PROPERTIES>
```

```
<ACTIONS>
    <ACTION NAME="move">
        <ARGUMENTS>
            <PARAMETER NAME="newX" TYPE="DOUBLE"/>
            <PARAMETER NAME="newY" TYPE="DOUBLE"/>
            <PARAMETER NAME="newZ" TYPE="DOUBLE"/>
        </ARGUMENTS>
        <DO_ACTION>
            <ASSIGN>
                <OPERAND>
                    <ELEMENT_PROP NAME="SELF"
                                  PROPERTY="position">
                        <INDEX> "x" </INDEX>
                    </ELEMENT_PROP>
                </OPERAND>
                <OPERAND>
                     <PARAMETER NAME="newX"
                          TYPE="DOUBLE"/>
                </OPERAND>
            </ASSIGN>
        </DO_ACTION>
        <DO_ACTION>
            <ASSIGN>
                <OPERAND>
                     <ELEMENT_PROP NAME="SELF"
                                   PROPERTY="position">
                        <INDEX> "y" </INDEX>
                    </ELEMENT_PROP>
                </OPERAND>
                <OPERAND>
                     <PARAMETER NAME="newY" TYPE="DOUBLE"/>
                </OPERAND>
            </ASSIGN>
        </DO_ACTION>
```

```
                    <DO_ACTION>
                        <ASSIGN>
                            <OPERAND>
                                <ELEMENT_PROP NAME="SELF"
                                        PROPERTY="position">
                                    <INDEX> "z" </INDEX>
                                </ELEMENT_PROP>

                            </OPERAND>
                            <OPERAND>
                                <PARAMETER NAME="newZ" TYPE="DOUBLE"/>
                            </OPERAND>
                        </ASSIGN>
                    </DO_ACTION>
                    <PHYSICAL_EVENT NAME=""/>
                </ACTION>
            </ACTIONS>
        </IVE_ARTIFACT>
```

El último artefacto representa un obstáculo, por tanto, no puede ser conectado por el robot (LINKEABLE = false) y además, no tiene ningún tipo de acción a realizar en el entorno.

```
        <IVE_ARTIFACT NAME="unlinkedArtifact" LINKEABLE="false">
            <ATTRIBUTES/>
            <PHYSICAL_PROPERTIES>
                <PERCEIVABLE>
                    <VECTOR3D NAME = "position">
                        <DOUBLE NAME = "x"> 20.0 </DOUBLE>
                        <DOUBLE NAME = "y"> 20.0 </DOUBLE>
                        <DOUBLE NAME = "z"> 20.0 </DOUBLE>
                    </VECTOR3D>
```

```
                    <VECTOR3D NAME = "velocity">
                        <DOUBLE NAME = "x"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "z"> 0.0 </DOUBLE>
                    </VECTOR3D>
                    <VECTOR3D NAME = "orientation">
                        <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                        <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "z"> 0.0 </DOUBLE>
                    </VECTOR3D>
                    <DOUBLE NAME = "distance"> 10.0 </DOUBLE>
                    <DOUBLE NAME = "angle"> 0.0 </DOUBLE>
                    <STRING NAME = "shape"> cubic </STRING>
                    <STRING NAME = "sound"> none </STRING>
                </PERCEIVABLE>
                <INTERNAL>
                    <VECTOR3D NAME = "acceleration">
                        <DOUBLE NAME = "x"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                        <DOUBLE NAME = "z"> 0.0 </DOUBLE>
                    </VECTOR3D>
                    <DOUBLE NAME = "mass"> 100.0 </DOUBLE>
                    <DOUBLE NAME = "size"> 10.0 </DOUBLE>
                </INTERNAL>
            </PHYSICAL_PROPERTIES>
            <ACTIONS/>
        </IVE_ARTIFACT>
```

Para acabar definimos el agente robot especificando (según el identificador de los artefactos en el archivo XML) sus cuerpos. Además indicamos donde se encuentra su comportamiento implementado (FILE). La *IVE law* en este caso de estudio no se aplica a ninguna acción.

```
        <INHABITANT_AGENT NAME="Robot">
            <ATTRIBUTES/>
            <BODY_ARTIFACT>
                <ITEM ID="0"/>
                <ITEM ID="1"/>
            </BODY_ARTIFACT>
            <FILE NAME="apodRobotJason.asl"/>
        </INHABITANT_AGENT>

        <IVE_LAW NAME="Gravity">
            <VECTOR3D NAME="gravity">
                <DOUBLE NAME="x"> 0.0 </DOUBLE>
                <DOUBLE NAME="y"> 0.0 </DOUBLE>
                <DOUBLE NAME="z"> 0.0 </DOUBLE>
            </VECTOR3D>
            <ACTIONS/>
        </IVE_LAW>
    </VIRTUAL>
</IVE>
```

Una vez definido el XML, se lo enviaríamos al *Manager* como argumento. Durante el proceso de lectura el generador de estructuras crearía las clases necesarias para la utilización de los artefactos en la integración Jason - CArtAgO.

A continuación mostramos las clase generadas para los artefactos BodyLeft y LinkedArtifact.

```
import cartago.*;

@ARTIFACT_INFO(
outports = {
@OUTPORT(name = "out-BodyLeft-Id0")
}
) public class BodyLeft_class extends IVEArtifact {

        // attributes and physical properties initialization
        void init(double xPosition, double yPosition,
                double zPosition, double xVelocity,
                double yVelocity, double zVelocity,
                double xOrientation, double yOrientation,
                double zOrientation, double distance, double angle,
                String shape, String sound, double xAcceleration,
                double yAcceleration, double zAcceleration,
                double mass, double size) {

                super.init(xPosition, yPosition, zPosition,
                        xVelocity, yVelocity, zVelocity,
                        xOrientation, yOrientation, zOrientation,
                        shape, sound, distance, angle,
                        xAcceleration, yAcceleration, zAcceleration,
                        mass, size);
        }
```

```
        // acciones
        @LINK void move(double newX, double newY,
                                            double newZ) {
                try {
                        setPositionValue(0, newX);
                        setPositionValue(1, newY);
                        setPositionValue(2, newZ);
                        execLinkedOp("out-BodyLeft-Id0",
                                "move", getPositionValue().x,
                                 getPositionValue().y,
                                getPositionValue().z);
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
}
```

```
import cartago.*;

@ARTIFACT_INFO(
outports = {
@OUTPORT(name = "out-linkedArtifact-Id2")
}
) public class LinkedArtifact_class extends IVEArtifact {

	// attributes and physical properties initialization
	void init(double xPosition, double yPosition,
		double zPosition, double xVelocity,
		double yVelocity, double zVelocity,
		double xOrientation, double yOrientation,
		double zOrientation, double distance, double angle,
		String shape, String sound, double xAcceleration,
		double yAcceleration, double zAcceleration,
		double mass, double size) {

		super.init(xPosition, yPosition, zPosition,
			xVelocity, yVelocity, zVelocity,
			xOrientation, yOrientation, zOrientation,
			shape, sound, distance, angle,
			xAcceleration, yAcceleration, zAcceleration,
			mass, size);
	}
	// acciones
	@OPERATION void move(double newX, double newY,
					double newZ) {
		try {
			setPositionValue(0, newX);
			setPositionValue(1, newY);
			setPositionValue(2, newZ);
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}
}
```

4.2. Simulación de un juego de estrategia en tiempo real

4.2.1. Descripción

En este caso de estudio se intenta emular el comportamiento de un juego de estrategia en tiempo real (RTS, *real strategy game*) simplificado basado en la recolección de materias primas.

En este tipo de juegos el tiempo transcurre de forma continua para el jugador y, normalmente, basan su forma de juego en acciones de estrategia militar. Se tendrán una serie de personajes, unidades o grupos que se dediquen a la recolección de materiales para la posterior construcción de edificios. Usualmente estos edificios aumentarán las posibilidades de acción dentro del juego, permitiendo la creación de nuevas unidades (por ejemplo de ataque) o para aumentar la velocidad de recolección de recursos. Ejemplos clásicos de este tipo de juegos pueden ser *Warcraft* [1], *Starcraft* [2] o *Age of empires* [3].

En este caso se ha simplificado la funcionalidad del juego. En el entorno podremos definir recolectores de recursos, en este caso de madera, que podrán talar objetos de tipo árbol para obternerlos. Para completar esta simulación quedaría por definir alguna unidad de ataque que pueda crearse a través de la construcción de algún edificio (a partir de madera). Los recolectores, tal como están definidos, no podrán atacar, pero sí quedar heridos.

4.2.2. Diseño del caso de estudio

En el entorno se crearán tres *IVE artifacts*: dos árboles (*tree*) y el cuerpo de un agente que recoge madera (*woodCollectorBody*) y un *artifact*: el reloj que controla el tiempo de juego. El número de artefactos a situar en el *IVE workspace* pueden ser tantos como queramos, podremos tener, por ejemplo, diez árboles y tres recolectores de madera, depende del diseño del juego en sí.

Los árboles pueden ser cortados por el agente, es por esto que realizamos una conexión estática inicialmente entre la acción *cutTree* del agente con la

[1] http://us.blizzard.com/es-mx/games/war3/
[2] http://us.blizzard.com/es-mx/games/sc/
[3] www.ageofempires.com/

acción *cut* del árbol. En los dos artefactos existen eventos físicos relacionados con la caída del árbol y con herir al agente recolector.

En este caso sí utilizamos la parte no virtual. En un *workspace* se crea un agente que controla el tiempo de juego, pudiéndolo incrementar y acabar la partida, a través del artefacto que representa el reloj del juego.

El mapa queda representado por el *IVE workspace terrain*. Éste *IVE workspace* representa lo que sería un terreno estándar en cualquier videojuego. Le introducimos dos artefactos de tipo árbol y el cuerpo del recolector de madera. La *IVE law* representa la densidad del medio, que en este caso es 1, y, por tanto, no afectaría a las acciones del entorno.

```
<?xml version="1.0" encoding="UTF-8"?>
<IVE NAME="rstGame"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="generalStructure.xsd">

    <VIRTUAL>

        <IVE_WORKSPACE NAME="terrain">
            <IVE_ARTIFACTS>
                <ITEM NAME="tree" />
                <ITEM NAME="tree" />
                <ITEM NAME="woodCollectorBody"/>
            </IVE_ARTIFACTS>
            <INHABITANT_AGENTS>
                <ITEM NAME="woodCollector"/>
            </INHABITANT_AGENTS>
            <IVE_LAWS>
                <ITEM NAME="density"/>
            </IVE_LAWS>
        </IVE_WORKSPACE>
```

Los artefactos de tipo árbol poseerán una acción que les permita ser cortados. Cuando un agente corte un árbol disminuirá su vitalidad y el agente obtendrá madera. Cuando la vitalidad llegue a cero, caerá y provocará el evento físico *fall*.

```
<IVE_ARTIFACT LINKEABLE="true" NAME="tree">
    <ATTRIBUTES/>
    <PHYSICAL_PROPERTIES>
        <PERCEIVABLE>
            <VECTOR3D NAME = "position">
                <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 0.0 </DOUBLE>
            </VECTOR3D>
            <VECTOR3D NAME = "velocity">
                <DOUBLE NAME = "x"> 0.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 0.0 </DOUBLE>
            </VECTOR3D>
            <VECTOR3D NAME = "orientation">
                <DOUBLE NAME = "x"> 0.0 </DOUBLE>
                <DOUBLE NAME = "y"> 1.0 </DOUBLE>
                <DOUBLE NAME = "z"> 0.0 </DOUBLE>
            </VECTOR3D>
            <DOUBLE NAME = "distance"> 100.0 </DOUBLE>
            <DOUBLE NAME = "angle"> 0.0 </DOUBLE>
            <STRING NAME = "shape"> tree </STRING>
        </PERCEIVABLE>
        <INTERNAL>
            <VECTOR3D NAME = "acceleration">
                <DOUBLE NAME = "x"> 0.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 0.0 </DOUBLE>
            </VECTOR3D>
            <DOUBLE NAME = "mass"> 500.0 </DOUBLE>
            <DOUBLE NAME = "size"> 3.0 </DOUBLE>
            <INTEGER NAME="vitality"> 50 </INTEGER>
        </INTERNAL>
    </PHYSICAL_PROPERTIES>
```

```
        <ACTIONS>
            <ACTION NAME="cut">
                    <ARGUMENTS>
                        <FEEDBACK NAME="fall" TYPE="BOOLEAN" />
                        <FEEDBACK NAME="woodObtained" TYPE="INTEGER" />
                        <PARAMETER NAME="attackValue" TYPE="INTEGER" />
                    </ARGUMENTS>
                    <DO_ACTION>
                        <ASSIGN>
                            <OPERAND>
                                <FEEDBACK NAME="fall" TYPE="BOOLEAN"/>
                            </OPERAND>
                            <OPERAND>
                                <BOOLEAN_VAL> false </BOOLEAN_VAL>
                            </OPERAND>
                        </ASSIGN>
                    </DO_ACTION>
                    <DO_ACTION>
                        <ASSIGN>
                            <OPERAND>
                                <ELEMENT_PROP PROPERTY="vitality"
                                        NAME="SELF"/>
                            </OPERAND>
                            <OPERAND>
                                <SUBSTRACT>
                                    <OPERAND>
                                        <ELEMENT_PROP
                                            PROPERTY="vitality"
                                            NAME="SELF"/>
                                    </OPERAND>
                                    <OPERAND>
                                        <PARAMETER NAME="attackValue"
                                                TYPE="INTEGER"/>
                                    </OPERAND>
                                </SUBSTRACT>
                            </OPERAND>
                        </ASSIGN>
                    </DO_ACTION>
```

```
                <PRECONDITION>
                    <LESSTHAN>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="vitality"
                                NAME="SELF"/>
                        </OPERAND>
                        <OPERAND>
                            <INT_VAL> 0 </INT_VAL>
                        </OPERAND>
                    </LESSTHAN>
                </PRECONDITION>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <FEEDBACK NAME="fall"
                                      TYPE="BOOLEAN"/>
                        </OPERAND>
                        <OPERAND>
                            <BOOLEAN_VAL> true </BOOLEAN_VAL>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <FEEDBACK NAME="woodObtained"
                                      TYPE="INTEGER"/>
                        </OPERAND>
                        <OPERAND>
                            <INT_VAL> 10 </INT_VAL>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <PHYSICAL_EVENT NAME="treeFall"/>
            </ACTION>
        </ACTIONS>
    </IVE_ARTIFACT>
```

El *IVE artifact* que define el cuerpo del agente recolector posee tres funciones. Una de movimiento, una para cuando es atacado por otro agente y una para cortar árboles. En esta última se realiza una conexión estática desde

el inicio, en este caso el *Manager* debería avisar al agente de que solo puede cortar un árbol si se encuentra cerca de él (según la propiedad *distance*). Se podría optar por una conexión dinámica y dejarlo todo en manos del *Manager*.

Cuando un agente es herido, puede llegar a morir. Para plasmar esto en el XML utilizamos una variable de tipo *OpFeedbackParam* llamada *die*. Se pondrá a true cuando la vitalidad del agente llegue a 0.

```
<IVE_ARTIFACT LINKEABLE="true" NAME="woodCollectorBody">
    <ATTRIBUTES />
    <PHYSICAL_PROPERTIES>
        <PERCEIVABLE>
            <VECTOR3D NAME = "position">
                <DOUBLE NAME = "x"> 10.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 10.0 </DOUBLE>
            </VECTOR3D>
            <VECTOR3D NAME = "velocity">
                <DOUBLE NAME = "x"> 2.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 2.0 </DOUBLE>
            </VECTOR3D>
            <VECTOR3D NAME = "orientation">
                <DOUBLE NAME = "x"> 1.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 0.0 </DOUBLE>
            </VECTOR3D>
            <DOUBLE NAME = "distance"> 20.0 </DOUBLE>
            <DOUBLE NAME = "angle"> 0.0 </DOUBLE>
            <STRING NAME = "shape"> woodCollector </STRING>
        </PERCEIVABLE>
        <INTERNAL>
            <VECTOR3D NAME = "acceleration">
                <DOUBLE NAME = "x"> 0.0 </DOUBLE>
                <DOUBLE NAME = "y"> 0.0 </DOUBLE>
                <DOUBLE NAME = "z"> 0.0 </DOUBLE>
            </VECTOR3D>
```

```
                <DOUBLE NAME = "mass"> 70.0 </DOUBLE>
                <DOUBLE NAME = "size"> 1.0 </DOUBLE>
                <INTEGER NAME="vitality"> 100 </INTEGER>
                <INTEGER NAME="attackValue"> 20 </INTEGER>
                <INTEGER NAME="woodCollected"> 0 </INTEGER>
            </INTERNAL>
        </PHYSICAL_PROPERTIES>

        <ACTIONS>
            <ACTION NAME="move">
                <ARGUMENTS>
                    <PARAMETER NAME="newX" TYPE="DOUBLE"/>
                    <PARAMETER NAME="newY" TYPE="DOUBLE"/>
                    <PARAMETER NAME="newZ" TYPE="DOUBLE"/>
                </ARGUMENTS>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="position"
                                    NAME="SELF">
                                <INDEX> "x" </INDEX>
                            </ELEMENT_PROP>
                        </OPERAND>
                        <OPERAND>
                            <PARAMETER NAME="newX" TYPE="DOUBLE"/>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="position"
                                    NAME="SELF">
                                <INDEX> "y" </INDEX>
                            </ELEMENT_PROP>
                        </OPERAND>
                        <OPERAND>
                            <PARAMETER NAME="newY" TYPE="DOUBLE"/>
                        </OPERAND>
```

```
                    </ASSIGN>
                </DO_ACTION>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="position"
                                  NAME="SELF">
                                <INDEX> "z" </INDEX>
                            </ELEMENT_PROP>
                        </OPERAND>
                        <OPERAND>
                            <PARAMETER NAME="newZ" TYPE="DOUBLE"/>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
            </ACTION>

            <ACTION NAME="wounded">
                <ARGUMENTS>
                    <FEEDBACK NAME="die" TYPE="BOOLEAN" />
                    <PARAMETER NAME="attackValue" TYPE="INTEGER" />
                </ARGUMENTS>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <FEEDBACK NAME="die" TYPE="BOOLEAN"/>
                        </OPERAND>
                        <OPERAND>
                            <BOOLEAN_VAL> false </BOOLEAN_VAL>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="vitality"
                                  NAME="SELF"/>
                        </OPERAND>
```

```
                        <OPERAND>
                            <SUBSTRACT>
                                <OPERAND>
                                    <ELEMENT_PROP PROPERTY="vitality"
                                        NAME="SELF"/>
                                </OPERAND>
                                <OPERAND>
                                    <PARAMETER NAME="attackValue"
                                        TYPE="INTEGER"/>
                                </OPERAND>
                            </SUBSTRACT>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <PRECONDITION>
                    <LESSTHAN>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="vitality"
                                NAME="SELF"/>
                        </OPERAND>
                        <OPERAND>
                            <INT_VAL> 0 </INT_VAL>
                        </OPERAND>
                    </LESSTHAN>
                </PRECONDITION>

                <DO_ACTION>
                    <ASSIGN>
                        <OPERAND>
                            <FEEDBACK NAME="die" TYPE="BOOLEAN"/>
                        </OPERAND>
                        <OPERAND>
                            <BOOLEAN_VAL> true </BOOLEAN_VAL>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
                <PHYSICAL_EVENT NAME="attacked"/>
            </ACTION>
            <ACTION NAME="cutTree">
                <ARGUMENTS>
```

```
            <FEEDBACK NAME="fall" TYPE="BOOLEAN"/>
            <PARAMETER NAME="xPos" TYPE="DOUBLE"/>
            <PARAMETER NAME="yPos" TYPE="DOUBLE"/>
            <PARAMETER NAME="zPos" TYPE="DOUBLE"/>
            <PARAMETER NAME="auxWoodCollected" TYPE="INTEGER"/>
        </ARGUMENTS>

        <PRECONDITION OP="and">
            <EQUAL>
                <OPERAND>
                    <ADD>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="position"
                                    NAME="SELF">
                                <INDEX> "x" </INDEX>
                            </ELEMENT_PROP>
                        </OPERAND>
                        <OPERAND>
                            <INT_VAL> 1 </INT_VAL>
                        </OPERAND>
                    </ADD>
                </OPERAND>
                <OPERAND>
                    <PARAMETER NAME="xPos" TYPE="DOUBLE"/>
                </OPERAND>
            </EQUAL>
        </PRECONDITION>

        <PRECONDITION OP="and">
            <EQUAL>
                <OPERAND>
                    <ELEMENT_PROP PROPERTY="position"
                            NAME="SELF">
                        <INDEX> "y" </INDEX>
                    </ELEMENT_PROP>
                </OPERAND>
                <OPERAND>
                    <PARAMETER NAME="yPos"
                        TYPE="DOUBLE"/>
                </OPERAND>
            </EQUAL>
```

```
                    </PRECONDITION>
                    <PRECONDITION>
                        <EQUAL>
                            <OPERAND>
                                <ELEMENT_PROP PROPERTY="position"
                                        NAME="SELF">
                                    <INDEX> "z" </INDEX>
                                </ELEMENT_PROP>
                            </OPERAND>
                            <OPERAND>
                                <PARAMETER NAME="zPos" TYPE="DOUBLE"/>
                            </OPERAND>
                        </EQUAL>
                    </PRECONDITION>
                    <DO_ACTION>
                        <EXEC_LINKED_OP LINKED_ART_ID="1"
                                LINKED_FUNCTION="cut">
                            <LINKED_ARGUMENTS>
                                <FEEDBACK NAME="fall" TYPE="BOOLEAN"/>
                                <PARAMETER NAME="auxWoodCollected"
                                    TYPE="INTEGER"/>
                                 <ELEMENT_PROP PROPERTY="attackValue"
                                        NAME="SELF"/>
                           </LINKED_ARGUMENTS>
                        </EXEC_LINKED_OP>
                    </DO_ACTION>
                    <DO_ACTION>
                        <ASSIGN>
                            <OPERAND>
                                <ELEMENT_PROP NAME="SELF"
                                    PROPERTY="woodCollected"/>
```

```
                        </OPERAND>
                        <OPERAND>
                            <PARAMETER NAME="auxWoodCollected"
                               TYPE="INTEGER"/>
                        </OPERAND>
                    </ASSIGN>
                </DO_ACTION>
            </ACTION>
        </ACTIONS>
    </IVE_ARTIFACT>
```

Una vez definido el cuerpo del agente, especificamos el *Inhabitant agent* del agente recolector. En este caso su cuerpo es el *IVE artifact woodCollectorBody*.

```
    <INHABITANT_AGENT NAME="woodCollector">
        <ATTRIBUTES />
            <BODY_ARTIFACT>
                <ITEM ID="1"/>
            </BODY_ARTIFACT>
            <FILE NAME="woodCollector.asl" />
    </INHABITANT_AGENT>

    <IVE_LAW NAME="density">
        <DOUBLE NAME="density"> 1.0 </DOUBLE>
            <ACTIONS>
                <ITEM NAME="move"/>
            </ACTIONS>
    </IVE_LAW>
</VIRTUAL>
```

Hasta aquí aparece definida la parte virtual. En la parte no virtual procederemos del mismo modo. Primero la incialización del *workspace*, luego la definición de artefactos y agentes. Como ejemplo se ha creado un agente que esté a cargo del reloj del juego. En su archivo Jason debería poder ejecutar la operación del artefacto *gameClock increaseTime*. Cuando se llega al final del juego, se produce un evento de fin llamado *end*.

```
    <NON_VIRTUAL>
        <WORKSPACE NAME="environmentManagement">
            <ARTIFACTS>
                <ITEM NAME="gameClock"/>
            </ARTIFACTS>
            <AGENTS>
                <ITEM NAME="clockManager"/>
            </AGENTS>
        </WORKSPACE>

        <ARTIFACT NAME="gameClock" LINKEABLE="true">
            <ATTRIBUTES/>
            <OBSERVABLE_PROPERTIES>
                <DOUBLE NAME="time"> 0.0 </DOUBLE>
            </OBSERVABLE_PROPERTIES>
            <OPERATIONS>
                <OPERATION NAME="increaseTime">
                    <ARGUMENTS />
                    <DO_OPERATION>
                        <ASSIGN>
                            <OPERAND>
                                <ELEMENT_PROP PROPERTY="time"
                                              NAME="SELF"/>
                            </OPERAND>
                            <OPERAND>
                                <ADD>
                                    <OPERAND>
                                        <ELEMENT_PROP PROPERTY="time"
                                                      NAME="SELF"/>
                                    </OPERAND>
```

```
                                <OPERAND>
                                    <DOUBLE_VAL> 1.0 </DOUBLE_VAL>
                                </OPERAND>
                            </ADD>
                        </OPERAND>
                    </ASSIGN>
                </DO_OPERATION>
            </OPERATION>

            <OPERATION NAME="gameEnd">
                <ARGUMENTS>
                    <FEEDBACK NAME="end" TYPE="BOOLEAN"/>
                </ARGUMENTS>

                <PRECONDITION>
                    <GREATERTHAN>
                        <OPERAND>
                            <ELEMENT_PROP PROPERTY="time"
                                    NAME="SELF"/>
                        </OPERAND>
                        <OPERAND>
                            <DOUBLE_VAL> 10000.0 </DOUBLE_VAL>
                        </OPERAND>
                    </GREATERTHAN>
                </PRECONDITION>
                <DO_OPERATION>
                    <ASSIGN>
                    <OPERAND>
                            <FEEDBACK NAME="end" TYPE="BOOLEAN"/>
                        </OPERAND>
                        <OPERAND>
                            <BOOLEAN_VAL> true </BOOLEAN_VAL>
                        </OPERAND>
                    </ASSIGN>
                </DO_OPERATION>
```

```
                </OPERATION>
            </OPERATIONS>
            <MANUAL> Artifact description </MANUAL>
            </ARTIFACT>

            <AGENT NAME="clockManager">
                <ATTRIBUTES/>
                    <FILE NAME="clockManager.asl"/>
            </AGENT>
        </NON_VIRTUAL>
</IVE>
```

Al igual que en el primer caso de estudio, el generador de estructuras crearía las siguientes clases para los artefactos *WoodCollectorBody* y *GameClock*. Cabe destacar el uso de la clase *IVEArtifact* por parte del cuerpo del recolector de madera y el uso de la clase *Artifact* por parte del reloj de juego.

```
import cartago.*;

@ARTIFACT_INFO(
outports = {
@OUTPORT(name = "out-woodCollectorBody-Id2")
}
) public class WoodCollectorBody_class extends IVEArtifact {

        // physical properties
        private int vitality;
        private int attackValue;
        private int woodCollected;

        // attributes and physical properties initialization
        void init(double xPosition, double yPosition,
                double zPosition, double xVelocity,
                double yVelocity, double zVelocity,
                double xOrientation, double yOrientation,
                double zOrientation, double distance, double angle,
                String shape, String sound, double xAcceleration,
                double yAcceleration, double zAcceleration,
                double mass, double size) {

                super.init(xPosition, yPosition, zPosition,
                        xVelocity, yVelocity, zVelocity,
                        xOrientation, yOrientation, zOrientation,
                        shape, sound, distance, angle,
                        xAcceleration, yAcceleration, zAcceleration,
                        mass, size);

                // physical properties
                this.vitality = vitality;
                this.attackValue = attackValue;
                this.woodCollected = woodCollected;
        }
```

```
        // acciones
        @OPERATION void move(double newX, double newY,
                                        double newZ) {
                try {
                        setPositionValue(0, newX);
                        setPositionValue(1, newY);
                        setPositionValue(2, newZ);
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }

        @OPERATION void wounded(OpFeedbackParam<Boolean> die,
                                int attackValue) {
                try {
                        die.set(false);
                        setVitality((getVitality() - attackValue));

                        if(getVitality() < 0) {

                                die.set(true);
                        }
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
        @OPERATION void cutTree(OpFeedbackParam<Boolean> fall,
                        double xPos, double yPos, double zPos,
                        int auxWoodCollected) {
                try {
                        if((getPositionValue().x + 1) == xPos
                                && getPositionValue().y == yPos
                                && getPositionValue().z == zPos) {

                                execLinkedOp(
                                        "out-woodCollectorBody-Id2",
                                        "cut", fall,
                                        auxWoodCollected,
                                        getAttackValue());
                                setWoodCollected(auxWoodCollected);
```

```
                        }
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
        }
        // getters & setters
        public void setVitality(int vitality) {
                this.vitality = vitality;
        }
        public int getVitality() {
                return this.vitality;
        }

        public void setAttackValue(int attackValue) {
                this.attackValue = attackValue;
        }
        public int getAttackValue() {
                return this.attackValue;
        }

        public void setWoodCollected(int woodCollected) {
                this.woodCollected = woodCollected;
        }
        public int getWoodCollected() {
                return this.woodCollected;
        }
}
```

```
import cartago.*;
import java.util.*;

@ARTIFACT_INFO(
outports = {
@OUTPORT(name = "out-gameClock-Id0")
}
) public class GameClock_class extends Artifact {

	// observable properties
	private ObsProperty time;

	// attributes and observable_properties initialization
	void init(double time) {

		// observable properties
		defineObsProperty("time", time);
		this.time = getObsProperty("time");
	}
	// operations
	@OPERATION void increaseTime() {
		try {
			setTime((getTime() + 1.0));
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}
	@OPERATION void gameEnd(OpFeedbackParam<Boolean> end) {
		try {

			if(getTime() > 10000.0) {

				end.set(true);
			}
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}
```

```
        // getters & setters
        public void setTime(double time) {
                this.time.updateValue(time);
        }
        public double getTime() {
                return this.time.doubleValues();
        }
}
```

Capítulo 5

Conclusiones y trabajos futuros

A través de la realización de este proyecto se han abordado dos objetivos principales: en primer lugar, la creación de una serie de herramientas de diseño de entornos virtuales inteligentes basados en el meta-modelo MAM5. En segundo lugar la creación de clases implementadas en CArtAgO para la ejecución del entorno en la plataforma Jason. Estos objetivos formaban parte de uno mayor: el establecer un sistema de ayuda al diseñador de entornos, desde la especificación y creación de los entornos virtuales inteligentes hasta su posterior simulación gráfica y gestión por parte de un proceso *Manager*. Los resultados más detacables de este proyecto han sido:

- La especificación formal del meta-modelo MAM5 a través de un documento esquema XSD. Esto permite que el desarrollador sea capaz de generar sus propios IVEs basados en lo establecido por MAM5, inicializándolos con los elementos que desee el propio desarrollador. La utilización de un documento XSD para la especificación del modelo permite la generación automática de instancias de IVEs en formato XML y el análisis sintáctico de ayuda a la corrección de errores dentro del archivo.

- La implementación de un proceso generador de estructuras que, a partir del archivo XML que define el entorno, se encarga de: almacenar su información en estructuras de datos, realizarle un análisis semántico y generar aquellas clases de artefactos necesarias para la simulación y gestión del entorno del proceso *Manager* del IVE.

- El desarrollo de un proceso de integración del generador de estructuras y del *Manager* del IVE mencionado anteriormente.

Líneas futuras de trabajo

Gracias a la propuesta planteada es posible definir entornos virtuales completamente desde cero. Sin embargo, cabe la posibilidad de que un diseñador de entornos esté interesado solamente en una de las partes que conforman un IVE. En este caso se podría plantear la creación de bibliotecas reutilizables e instanciables tanto de *workspaces* como de *IVE workspaces*. De esta manera obtendríamos entornos predefinidos donde se podrían personalizar los agentes y artefactos u otros elementos del entorno a gusto del usuario, ahorrando tiempo en la definición general del entorno y centrándose solamente en aquellos elementos en los que se tenga especial interés.

Como se ha comentado en la sección 3.1 de esta memoria, todo el proyecto ha sido realizado a través del IDE Eclipse. El uso de esta herramienta en todas las facetas de la implementación ayuda a la integración de los diferentes elementos que conforman el proyecto. Gracias al amplio rango de funcionalidad de Eclipse resulta relativamente simple el añadir nuevas características a las herramientas creadas. Dentro de este ámbito podría plantearse la creación, a través de plugins o herramientas CASE en Eclipse, de una interfaz gráfica de usuario previa a la creación del archivo XML, añadiendo recursos de ayuda gráfica al diseñador de entornos.

Apéndice

Esquema XSD generado para la ontología MAM5 *generalStructure.xsd*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

     <!-- IVE TYPE -->

    <xs:element name="IVE" type="iveType"></xs:element>

    <xs:complexType name="iveType">
        <xs:sequence>
                <xs:element name="VIRTUAL" type="virtualType">
                </xs:element>
                <xs:element name="NON_VIRTUAL"
                        type="nonVirtualType">
                </xs:element>
        </xs:sequence>

        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
    </xs:complexType>


        <!-- VIRTUAL and NON-VIRTUAL TYPES -->

    <xs:complexType name="virtualType">
        <xs:sequence>
                <xs:element name="IVE_WORKSPACE"
                        type="iveWorkspaceType"
                        maxOccurs="unbounded" minOccurs="1">
                </xs:element>
                <xs:element name="IVE_ARTIFACT"
                        type="iveArtifactType"
                        maxOccurs="unbounded" minOccurs="1">
                </xs:element>

            <xs:element name="INHABITANT_AGENT"
                        type="inhabitantType"
                        maxOccurs="unbounded" minOccurs="1">
                </xs:element>
```

```
            <xs:element name="IVE_LAW" type="iveLawType"
                    maxOccurs="unbounded" minOccurs="1">
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="nonVirtualType">
        <xs:sequence>
            <xs:element name="WORKSPACE" type="workspaceType"
                    maxOccurs="unbounded" minOccurs="1">
            </xs:element>
            <xs:element name="ARTIFACT" type="artifactType"
                    maxOccurs="unbounded" minOccurs="1">
            </xs:element>
            <xs:element name="AGENT" type="agentType"
                    maxOccurs="unbounded" minOccurs="1">
            </xs:element>

        </xs:sequence>
    </xs:complexType>


    <!-- VIRTUAL INSIDE TYPES -->

    <xs:complexType name="iveWorkspaceType">
        <xs:sequence>
            <xs:element name="IVE_ARTIFACTS" type="itemsList">
            </xs:element>
            <xs:element name="INHABITANT_AGENTS" type="itemsList">
            </xs:element>
            <xs:element name="IVE_LAWS" type="itemsList">
            </xs:element>
        </xs:sequence>
        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
    </xs:complexType>
    <xs:complexType name="iveArtifactType">
        <xs:sequence>
```

```
            <xs:element name="ATTRIBUTES" type="attributesList">
            </xs:element>
            <xs:element name="PHYSICAL_PROPERTIES"
                   type="physicalPropertiesType">
            </xs:element>
            <xs:element name="ACTIONS" type="actionsList">
            </xs:element>
        </xs:sequence>
        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
        <xs:attribute name="LINKEABLE" type="xs:string"
               use="required"></xs:attribute>
    </xs:complexType>
    <xs:complexType name="inhabitantType">
        <xs:sequence>
               <xs:element name="ATTRIBUTES"
                      type="attributesList"></xs:element>
               <xs:element name="BODY_ARTIFACT" type="idsList"
               maxOccurs="1" minOccurs="1"></xs:element>
               <xs:element name="FILE" type="fileType">
               </xs:element>
               </xs:sequence>
        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
    </xs:complexType>

    <xs:complexType name="iveLawType">
        <xs:sequence>
               <xs:choice>
                      <xs:choice>
                             <xs:element name="BOOLEAN"
                             type="booleanAtt"></xs:element>
                             <xs:element name="INTEGER"
                             type="integerAtt"></xs:element>
                             <xs:element name="FLOAT"
                             type="floatAtt"></xs:element>
                             <xs:element name="DOUBLE"
                             type="doubleAtt"></xs:element>
                             <xs:element name="STRING"
                             type="stringAtt"></xs:element>
                             <xs:element name="VECTOR3D"
```

```
                                type="vector3DType"></xs:element>
                        </xs:choice>
                        <xs:sequence>
                                <xs:element name="CONDITION"
                                        type="xs:string"></xs:element>
                                <xs:element name="SENTENCE"
                                        type="xs:string"></xs:element>
                        </xs:sequence>
                        </xs:choice>
                <xs:element name="ACTIONS" type="itemsList">
        </xs:element>
        </xs:sequence>
        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
    </xs:complexType>
    <!-- NON-VIRTUAL INSIDE TYPES -->

    <xs:complexType name="workspaceType">
        <xs:sequence>
                <xs:element name="ARTIFACTS" type="itemsList"
                        maxOccurs="unbounded" minOccurs="1">
                </xs:element>
                <xs:element name="AGENTS" type="itemsList"
                        maxOccurs="unbounded" minOccurs="1">
                </xs:element>
        </xs:sequence>
        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
    </xs:complexType>

    <xs:complexType name="artifactType">
        <xs:sequence>
                <xs:element name="ATTRIBUTES" type="attributesList">
                </xs:element>
                <xs:element name="OBSERVABLE_PROPERTIES"
                        type="propertiesList">
                </xs:element>
                <xs:element name="OPERATIONS" type="nonVirtualOpList">
                </xs:element>
```

```
                <xs:element name="MANUAL" type="xs:string">
                </xs:element>
        </xs:sequence>

        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
        <xs:attribute name="LINKEABLE" type="xs:string"
                use="required"></xs:attribute>
    </xs:complexType>
    <xs:complexType name="agentType">
        <xs:sequence>
                <xs:element name="ATTRIBUTES" type="attributesList"
                        maxOccurs="unbounded" minOccurs="0">
                </xs:element>
                <xs:element name="FILE" type="fileType"
                        maxOccurs="1" minOccurs="1">
                </xs:element>
        </xs:sequence>
        <xs:attribute name="NAME" type="xs:string"
                use="required"></xs:attribute>
    </xs:complexType>
    <!-- IVE_ARTIFACT INSIDE TYPES -->

        <xs:complexType name="physicalPropertiesType">
        <xs:sequence>
                <xs:element name="PERCEIVABLE"
                        type="propertiesList"></xs:element>
                <xs:element name="INTERNAL"
                        type="propertiesList"></xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="actionsList">
        <xs:sequence>
                <xs:element name="ACTION" type="actionType"
                        maxOccurs="unbounded" minOccurs="0">
                </xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="actionType">
        <xs:sequence>
                <xs:element name="ARGUMENTS" type="argsType">
```

```
                </xs:element>
                <xs:sequence maxOccurs="unbounded" minOccurs="1">
                        <xs:element name="PRECONDITION"
                                type="preconditionType"
                                maxOccurs="unbounded" minOccurs="0">
                        </xs:element>
                        <xs:element name="DO_ACTION"
                                type="doActionType"
                                maxOccurs="unbounded" minOccurs="1">
                        </xs:element>
                </xs:sequence>
                <xs:element name="PHYSICAL_EVENT" type="eventType"
                maxOccurs="unbounded" minOccurs="0"></xs:element>
        </xs:sequence>
        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
    </xs:complexType>


        <!-- ARTIFACT INSIDE TYPES -->

        <xs:complexType name="nonVirtualOpList">
        <xs:sequence>
                <xs:element name="OPERATION"
                        type="nonVirtualOperationType"
                        maxOccurs="unbounded" minOccurs="1">
                </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="nonVirtualOperationType">
        <xs:sequence>
                <xs:element name="ARGUMENTS" type="argsType">
                </xs:element>
                <xs:element name="PRECONDITION"
                        type="preconditionType"
                        maxOccurs="unbounded" minOccurs="0">
                </xs:element>
                <xs:element name="DO_OPERATION"
                        type="doActionType"
```

```
                    maxOccurs="unbounded" minOccurs="1">
            </xs:element>
            <xs:element name="OBSERVABLE_EVENT"
                    type="eventType"
                    maxOccurs="unbounded" minOccurs="0">
            </xs:element>
        </xs:sequence>
        <xs:attribute name="NAME" type="xs:string"
                use="required"></xs:attribute>
    </xs:complexType>


    <xs:complexType name="linkedArtifact">
        <xs:sequence>
                <xs:element name="LINKED_ARGUMENTS"
                type="linkedArgumentType"></xs:element>
        </xs:sequence>
        <xs:attribute name="LINKED_ART_ID" type="xs:string"
                use="required">
        </xs:attribute>
        <xs:attribute name="LINKED_FUNCTION"
        type="xs:string" use="required"></xs:attribute>
    </xs:complexType>

    <xs:complexType name="linkedArgumentType">
        <xs:choice maxOccurs="unbounded" minOccurs="0">
                <xs:element name="ELEMENT_ATT"
                        type="elementAttType">
                </xs:element>
                <xs:element name="ELEMENT_PROP"
                        type="elementPhyPropType">
                </xs:element>
                <xs:element name="PARAMETER"
                        type="argumentType">
                </xs:element>
                <xs:element name="FEEDBACK"
                        type="argumentType">
                </xs:element>
        </xs:choice>
    </xs:complexType>
```

```
<!-- INHABITANT_AGENT and AGENT INSIDE TYPES -->

    <xs:complexType name="idsList">
    <xs:sequence>
           <xs:element name="ITEM" type="idType"
                  maxOccurs="unbounded" minOccurs="1">
           </xs:element>
    </xs:sequence>
</xs:complexType>

    <xs:complexType name="idType">
           <xs:attribute name="ID" type="xs:string"
                  use="required">
           </xs:attribute>
</xs:complexType>

    <xs:complexType name="fileType">
    <xs:attribute name="NAME" type="xs:string"
           use="required">
    </xs:attribute>
</xs:complexType>


    <!-- GENERAL LISTS -->

    <xs:complexType name="itemsList">
    <xs:sequence>
           <xs:element name="ITEM" type="itemType"
                  maxOccurs="unbounded" minOccurs="1">
           </xs:element>
    </xs:sequence>
</xs:complexType>

    <xs:complexType name="itemType">
           <xs:attribute name="NAME" type="xs:string"
                  use="required">
           </xs:attribute>
</xs:complexType>
```

```
    <xs:complexType name="propertiesList">
        <xs:choice maxOccurs="unbounded" minOccurs="1">
                <xs:element name="BOOLEAN" type="booleanAtt">
                </xs:element>
                <xs:element name="INTEGER" type="integerAtt">
                </xs:element>
                <xs:element name="FLOAT" type="floatAtt">
                </xs:element>
                <xs:element name="DOUBLE" type="doubleAtt">
                </xs:element>
                <xs:element name="STRING" type="stringAtt">
                </xs:element>
                <xs:element name="VECTOR3D" type="vector3DType">
                </xs:element>
        </xs:choice>
    </xs:complexType>


        <xs:complexType name="attributesList">
        <xs:choice maxOccurs="unbounded" minOccurs="0">
                <xs:element name="BOOLEAN" type="booleanAtt">
                </xs:element>
                <xs:element name="INTEGER" type="integerAtt">
                </xs:element>
                <xs:element name="FLOAT" type="floatAtt">
                </xs:element>
                <xs:element name="DOUBLE" type="doubleAtt">
                </xs:element>
                <xs:element name="STRING" type="stringAtt">
                </xs:element>
        </xs:choice>
    </xs:complexType>

   <xs:complexType name="booleanAtt">
        <xs:simpleContent>
                <xs:extension base="xs:boolean">
                <xs:attribute name="NAME" type="xs:string"
                        use="required">
                </xs:attribute>
                </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
```

```
<xs:complexType name="integerAtt">
    <xs:simpleContent>
            <xs:extension base="xs:int">
            <xs:attribute name="NAME" type="xs:string"
             use="required">
            </xs:attribute>
            </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="floatAtt">
    <xs:simpleContent>
            <xs:extension base="xs:float">
            <xs:attribute name="NAME" type="xs:string"
            use="required">
            </xs:attribute>
            </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="doubleAtt">
    <xs:simpleContent>
            <xs:extension base="xs:double">
            <xs:attribute name="NAME" type="xs:string"
            use="required">
            </xs:attribute>
            </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="stringAtt">
    <xs:simpleContent>
            <xs:extension base="xs:string">
            <xs:attribute name="NAME" type="xs:string"
             use="required">
            </xs:attribute>
            </xs:extension>
    </xs:simpleContent>
</xs:complexType>
```

```
<xs:complexType name="vector3DType">
    <xs:choice>
           <xs:sequence>
                  <xs:element name="DOUBLE" type="doubleAtt">
                  </xs:element>
                  <xs:element name="DOUBLE" type="doubleAtt">
                  </xs:element>
                  <xs:element name="DOUBLE" type="doubleAtt">
                  </xs:element>
           </xs:sequence>
    </xs:choice>
    <xs:attribute name="NAME" type="xs:string" use="required">
    </xs:attribute>
</xs:complexType>


<!-- ACTION and OPERATION INSIDE TYPES -->

    <xs:complexType name="argsType">
    <xs:choice maxOccurs="unbounded" minOccurs="0">
           <xs:element name="FEEDBACK" type="argumentType"
                  maxOccurs="unbounded" minOccurs="0">
           </xs:element>
           <xs:element name="PARAMETER" type="argumentType"
                  maxOccurs="unbounded" minOccurs="0">
           </xs:element>
    </xs:choice>
</xs:complexType>

<xs:complexType name="argumentType">
    <xs:attribute name="NAME" type="xs:string"
           use="required">
    </xs:attribute>
    <xs:attribute name="TYPE" type="xs:string"
            use="required">
    </xs:attribute>
</xs:complexType>

    <xs:complexType name="preconditionType">
    <xs:choice maxOccurs="1" minOccurs="1">
           <xs:element name="EQUAL" type="operationType">
```

```
                </xs:element>
                <xs:element name="UNEQUAL" type="operationType">
                </xs:element>
                <xs:element name="GREATERTHAN" type="operationType">
                </xs:element>
                <xs:element name="LESSTHAN" type="operationType">
                </xs:element>
        </xs:choice>
        <xs:attribute name="OP" type="xs:string" use="optional">
        </xs:attribute>
    </xs:complexType>

    <xs:complexType name="operationType">
        <xs:sequence>
                <xs:element name="OPERAND" type="operandType"
                maxOccurs="1" minOccurs="1"/>
                <xs:element name="OPERAND" type="operandType"
                maxOccurs="1" minOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="operandType">
        <xs:choice>
                <xs:element name="EQUAL" type="operationType">
                </xs:element>
                <xs:element name="UNEQUAL" type="operationType">
                </xs:element>
                <xs:element name="GREATERTHAN" type="operationType">
                </xs:element>
                <xs:element name="LESSTHAN" type="operationType">
                </xs:element>
                <xs:element name="ADD" type="operationType">
                </xs:element>
                <xs:element name="SUBSTRACT" type="operationType">
                </xs:element>
                <xs:element name="MULTIPLY" type="operationType">
                </xs:element>
                <xs:element name="DIVIDE" type="operationType">
                </xs:element>
```

```
            <xs:element name="MOD" type="operationType">
            </xs:element>
            <xs:element name="ELEMENT_ATT"
                   type="elementAttType">
            </xs:element>
            <xs:element name="ELEMENT_PROP"
                   type="elementPhyPropType">
            </xs:element>
            <xs:element name="PARAMETER" type="argumentType">
            </xs:element>
            <xs:element name="AND" type="operationType">
            </xs:element>
            <xs:element name="OR" type="operationType">
            </xs:element>
            <xs:element name="BOOLEAN_VAL" type="xs:boolean">
            </xs:element>
            <xs:element name="INT_VAL" type="xs:integer">
            </xs:element>
            <xs:element name="FLOAT_VAL" type="xs:float">
            </xs:element>
            <xs:element name="DOUBLE_VAL" type="xs:double">
            </xs:element>
            <xs:element name="STRING_VAL" type="xs:string">
            </xs:element>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="elementAttType">
        <xs:attribute name="NAME" type="xs:string"
               use="required">
        </xs:attribute>
        <xs:attribute name="ATTRIBUTE" type="xs:string"
               use="required">
        </xs:attribute>
    </xs:complexType>

    <xs:complexType name="elementPhyPropType">
        <xs:sequence>
               <xs:element name="INDEX" type="xs:string"
               minOccurs="0" maxOccurs="1"></xs:element>
        </xs:sequence>
```

```
        <xs:attribute name="NAME" type="xs:string"
                use="required">
        </xs:attribute>
        <xs:attribute name="PROPERTY" type="xs:string"
                use="required">
        </xs:attribute>
    </xs:complexType>

    <xs:complexType name="doActionType">
        <xs:choice>
                <xs:element name="ASSIGN" type="assignType">
                </xs:element>
                <xs:element name="EXEC_LINKED_OP"
                        type="linkedArtifact"
                        maxOccurs="1" minOccurs="0">
                </xs:element>
        </xs:choice>
    </xs:complexType>
    <xs:complexType name="assignType">
        <xs:sequence>
                <xs:element name="OPERAND"
                        type="firstAssignOperand"
                        maxOccurs="1" minOccurs="1">
                </xs:element>
                <xs:element name="OPERAND"
                        type="scndAssignOperand"
                        maxOccurs="1" minOccurs="1">
                </xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="firstAssignOperand">
        <xs:choice maxOccurs="1" minOccurs="1">
                <xs:element name="FEEDBACK"
                        type="argumentType">
                </xs:element>
                <xs:element name="ELEMENT_PROP"
                        type="elementPhyPropType">
                </xs:element>
                <xs:element name="ELEMENT_ATT"
                        type="elementAttType">
                </xs:element>
```

```
                <xs:element name="PARAMETER"
                        type="argumentType">
                </xs:element>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="scndAssignOperand">
        <xs:choice maxOccurs="1" minOccurs="1">
                <xs:element name="EQUAL"
                        type="scndOperationType">
                </xs:element>
                <xs:element name="UNEQUAL"
                        type="scndOperationType">
                </xs:element>
                <xs:element name="GREATERTHAN"
                        type="scndOperationType">
                </xs:element>
                <xs:element name="LESSTHAN"
                        type="scndOperationType">
                </xs:element>
                <xs:element name="ADD"
                        type="scndOperationType">
                </xs:element>
                <xs:element name="SUBSTRACT"
                        type="scndOperationType">
                </xs:element>
                <xs:element name="MULTIPLY" type="scndOperationType">
                </xs:element>
                <xs:element name="DIVIDE" type="scndOperationType">
                </xs:element>
                <xs:element name="MOD" type="scndOperationType">
                </xs:element>
                <xs:element name="AND" type="operationType">
                </xs:element>
                <xs:element name="OR" type="operationType">
                </xs:element>
                <xs:element name="ELEMENT_ATT"
                        type="elementAttType">
                </xs:element>
```

```
            <xs:element name="ELEMENT_PROP"
                    type="elementPhyPropType">
            </xs:element>
            <xs:element name="PARAMETER" type="argumentType">
            </xs:element>
            <xs:element name="FEEDBACK" type="argumentType">
            </xs:element>
            <xs:element name="BOOLEAN_VAL" type="xs:boolean">
            </xs:element>
            <xs:element name="INT_VAL" type="xs:integer">
            </xs:element>
            <xs:element name="FLOAT_VAL" type="xs:float">
            </xs:element>
            <xs:element name="DOUBLE_VAL" type="xs:double">
            </xs:element>
            <xs:element name="STRING_VAL" type="xs:string">
            </xs:element>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="scndOperationType">
        <xs:sequence>
            <xs:element name="OPERAND" type="scndAssignOperand"
            maxOccurs="1" minOccurs="1"/>
            <xs:element name="OPERAND" type="scndAssignOperand"
            maxOccurs="1" minOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="eventType">
        <xs:sequence>
            <xs:element name="PARAMETERS" type="eventParameterList"
            maxOccurs="1" minOccurs="0"></xs:element>
        </xs:sequence>
        <xs:attribute name="AGENT_ID" type="xs:string" use="optional">
        </xs:attribute>
        <xs:attribute name="NAME" type="xs:string" use="required">
        </xs:attribute>
    </xs:complexType>
```

```
    <xs:complexType name="eventParameterList">
      <xs:choice maxOccurs="1" minOccurs="1">
            <xs:element name="ELEMENT_ATT" type="elementAttType">
            </xs:element>
            <xs:element name="ELEMENT_PROP" type="elementPhyPropType">
            </xs:element>
            <xs:element name="PARAMETER" type="argumentType">
            </xs:element>
            <xs:element name="FEEDBACK" type="argumentType">
            </xs:element>
            <xs:element name="BOOLEAN_VAL" type="xs:boolean">
            </xs:element>
            <xs:element name="INT_VAL" type="xs:integer">
            </xs:element>
            <xs:element name="FLOAT_VAL" type="xs:float">
            </xs:element>
            <xs:element name="DOUBLE_VAL" type="xs:double">
            </xs:element>
            <xs:element name="STRING_VAL" type="xs:string">
            </xs:element>
      </xs:choice>
   </xs:complexType>
</xs:schema>
```

Definición de la clase IVEArtifact.java en CArtAgO

```
import cartago.*;

class IVEArtifact extends Artifact {

        private ObsProperty position;
        private ObsProperty velocity;
        private ObsProperty orientation;
        private ObsProperty shape;
        private ObsProperty sound;
        private ObsProperty distance;
        private ObsProperty angle;
        private Vector3D acceleration;
        private double mass;
        private double size;

        void init(double px, double py, double pz,
                double vx, double vy, double vz,
                double ox, double oy, double oz,
                String shape, String sound,
                double distance,
                double angle,
                double ax, double ay, double az,
                double mass, double size) {

                defineObsProperty("position", px, py, pz);
                this.position = getObsProperty("position");

                defineObsProperty("velocity", vx, vy, vz);
                this.velocity = getObsProperty("velocity");

                defineObsProperty("orientation", ox, oy, oz);
                this.orientation = getObsProperty("orientation");

                defineObsProperty("shape", shape);
                this.shape = getObsProperty("shape");

                defineObsProperty("sound", sound);
                this.sound = getObsProperty("sound");
```

```
            defineObsProperty("distance", distance);
            this.distance = getObsProperty("distance");

            defineObsProperty("angle", angle);
            this.angle = getObsProperty("angle");

            this.acceleration = new Vector3D(ax, ay, az);

            this.mass = mass;

            this.size = size;
        }

        public void setPositionValue(int id, double value) {
            this.position.updateValue(id, value);
        }
        public Vector3D getPositionValue() {

            Vector3D position = new Vector3D();

            position.x = this.position.doubleValue(0);
            position.y = this.position.doubleValue(1);
            position.z = this.position.doubleValue(2);

            return position;
        }

        public void setVelocity(int id, double value) {
            this.velocity.updateValue(id, value);
        }
        public Vector3D getVelocity() {

            Vector3D velocity = new Vector3D();

            velocity.x = this.velocity.doubleValue(0);
            velocity.y = this.velocity.doubleValue(1);
            velocity.z = this.velocity.doubleValue(2);

            return velocity;
        }
```

```
	public void setOrientation(int id, double value) {
		this.orientation.updateValue(id, value);
	}
	public Vector3D getOrientation() {

		Vector3D orientation = new Vector3D();

		orientation.x = this.orientation.doubleValue(0);
		orientation.y = this.orientation.doubleValue(1);
		orientation.z = this.orientation.doubleValue(2);

		return orientation;
	}

	public void setShape(String shape) {
		this.shape.updateValue(shape);
	}
	public String getShape() {

		return this.shape.stringValue();
	}

	public void setSound(String value) {
		this.sound.updateValue(value);
	}
	public String getSound() {
		return this.sound.stringValue();
	}

	public void setDistance(int id, double value) {
		this.distance.updateValue(id, value);
	}
	public double getDistance() {
		return this.distance.doubleValues();
	}
```

```
        public void setAngle(int id, double value) {
                this.distance.updateValue(id, value);
        }
        public double getAngle() {
                return this.angle.doubleValues();
        }

        public void setAcceleration(int id, double value) {
                if(id == 0)
                        this.acceleration.x = value;
                else if(id == 1)
                        this.acceleration.y = value;
                else
                        this.acceleration.z = value;
        }
        public Vector3D getAcceleration() {
                return this.acceleration;
        }

        public void setMass(double mass) {
                this.mass = mass;
        }
        public double getMass() {
                return this.mass;
        }

        public void setSize(double size) {
                this.size = size;
        }
        public double getSize() {
                return this.size;
        }
}
```

Clase auxiliar Vector3D desarrollada para soportar el caso de estudio CTF [26]

```
public class Vector3D {

	public double x;
	public double y;
	public double z;

	public Vector3D() {
	}

	public Vector3D(double x, double y, double z) {
		this.x = x;
		this.y = y;
		this.z = z;
	}
}
```

Bibliografía

[1] Michael Wooldridge. An introduction to Multiagent Systems. John Wiley & Sons, Ltd.2002.

[2] Victor R. Lesser. Cooperative Multiagent systems: A Personal View of the State of Art. In *IEEE Trans. Knowledge and Data Engineering*, Jan.-Feb. Special Issue.1999.

[3] Gerhard Weiss. Multiagent Systems: A Modern Approach to Distributed Modern Approach to Artificial Intelligence. The MIT Press. Cambridge, Massachusetts. 1999.

[4] G. Anastassakis, T. Ritchings and T. Panayiotopoulos. Multi-agent systems as intelligent virtual environments. In *Proceedings of Advances in Artificial Intelligent, Joint German/Austrian Conference on* AI - KI. 2001.

[5] A. Barella, A. Ricci, O. Boissier and C. Carrascosa. MAM5: Multi-Agent Model For Intelligent Virtual Environments. In *EUMAS* 2012.

[6] Zhong Zhang, James D. Mccalley. Distributed decision-making in electric power system transmission maintenance scheduling using multi-agent systems. Doctoral Dissertation. Iowa State University Ames, IA, USA. 2004.

[7] Juan Pavon, Candelaria Sansores, Jorge J. Gomez-Sanz, Fei-Yue Wang. Modelling and simulation of social systems with INGENIAS. In *International Journal of Agent-Oriented Software Engineering* 2 (2). pp. 196–221. 2002.

[8] Paul Davidsson. Agent Based Social Simulation: A Computer Science View. In *Journal of Artificial Societies and Social Simulation*, vol. 5, no. 1. http://jasss.soc.surrey.ac.uk/5/1/7.html. 2002.

[9] A. Paiva, J. Dias, D. Sobral, R. Aylett, S. Woods, L. Hall and C. Zoll. Learning by feeling: Evoking empathy with synthetic characters. *Applied Artificial Intelligence*, 19(3):235–266. 2005.

[10] Marco Raberto, Silvano Cincotti and Sergio M Focardi, Michele Marchesi. Agent-based simulation of a financial market. In *Physica A: Statistical Mechanics and its Applications* journal, vol. 299, no. 1, p. 319-327. 2001.

[11] Li, M., Chong, K. W., Chan, S., Hallan, J., Agent-Oriented Urban traffic Simulation, The 1st International Conference on Industrial Engineering Application and Practic. 1996.

[12] Ana Bazzan, Franziska Klügl, Sascha Ossowski, Giuseppe Vizzari. Agents in Traffic and Transportation. An International Workshop Series. http://www.ia.urjc.es/ATT/. 2000-2012.

[13] C. Sierra, F. Dignum: Agent-Mediated Electronic Commerce: Scientific and Technological Roadmap, In F. Dignum y C. Sierra (Eds.) Agent-mediated Electronic commerce (The European AgentLink Perspective), LNAI 1991, pp. 1-18. SpringerVerlag: 2001.

[14] R.H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agent-mediated Electronic Commerce: A Survey. In *Knowledge Engineering Review*, vol. 13:3. 1998.

[15] S. Bussmann and K. Schild. An agent-based approach to the control of flexible production systems. In *Proceedings of 8th IEEE International Conference on Emerging Technologies and Factory Automation*, vol. 2. Antibes, France, pp. 481–488. 2001.

[16] Stefan Hahndel, Florian Fuchs and Paul Levi. Distributed Negotiation-based Task Planning for a Flexible Manufacturing Environment. In *Distributed Software Agents and Applications (MAAMAW-94)*, vol. 1069 of LNAI. 1996.

[17] Surangika Ranathunga, Stephen Cranefield and Martin Purvis. Extracting data from second life. In *Technical Report. Otago University*, 2011.

[18] Ivan Medeiros Monteiro y Luis Otávio Alvares. A teamwork infraestructure for computer games with real-time requirements. In *AGS*, pages 48-62, 2009.

[19] Fabio Y. Okuyama, Rafael H. Bordini and Antônio Carlos da Rocha Costa. ELMS: An Environment Description Language for Multi-Agent Simulation.

[20] Special issue on environments for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):1-116, 2007.

[21] Jeehang Lee, Vincent Baines and Julian Padget. Decoupling cognitive agents and virtual environments. In *CAVE*, pages 17-32, 2012.

[22] FIPA (2001), Fipa interaction protocol library specification, Technical Report XC00025E. http://www.fipa.org/specs/fipa00085/SC00085J.html [Consultado por última vez 22/09/2013]

[23] Joao Dias, Samuel Mascarenhas, Ana Paiva. FAtiMA Modular: Towards an Agent Architecture with a Generic Appraisal Framework. INESC-ID, Instituto Superior Técnico, Portugal.

[24] Michael Luck and Ruth Aylett. Applying artificial intelligence to virtual reality: Intelligent virtual environments. *Applied Artificial intelligence*, 14(1):3-32, 2000.

[25] Alessandro Ricci, Michele Piunti y Mirko Viroli. Environment Programming in Multi-Agent Systems – An Artifact-Based Perspective. DEIS, Alma Mater Studiorum Università di Bologna.

[26] Antonio Barella, Soledad Valero y Carlos Carrascosa. JGOMAS: New Approach to AI Teaching. IEEE Transactions on Education, 2009. IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS INC, volume 52, number 2, pages 228-235, isbn 0018-9358.