Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Herramienta de modelado de entornos virtuales inteligentes

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Azucena Sastre Garrido

Tutor: Carlos Carrascosa Casamayor

2019

Resumen

A lo largo de esta memoria se va a seguir el proceso de diseño de una aplicación de escritorio que permita realizar un modelo para entornos virtuales inteligentes. Está destinada, específicamente, para complementar al *framework* JaCalIVE de desarrollo de sistemas multiagente. Se empieza explicando y estudiando esta herramienta, comprendiendo los aspectos más básicos para determinar lo necesario, y así crear una herramienta que pueda servir de base. Se detallará cada una de las fases de análisis mediante la propuesta de casos de uso, estructuras del modelo de datos y el diseño de la interfaz.

Palabras clave: sistemas multi-agente, modelado, JaCalIVE, interfaz, entornos virtuales inteligentes.

Abstract

This document will go through the design of a desktop application for creating models for intelligent virtual environments. It is specifically intended to complement the JaCalIVE multiagent systems development framework. The process begins by explaining and studying this tool, understanding the most basic aspects to determine what is necessary, and thus create a tool that may serve as a starting point. Each of the analysis phases will be detailed through use cases, data model structures and interface design.

Keywords: multiagent system, modelling, JaCalIVE, interface, intelligent virtual environments.

Tabla de contenidos

Índice de figuras

Índice de tablas

1. Introducción

Los entornos virtuales son simulaciones por ordenador de mundos reales o imaginarios creados con el fin de representar situaciones.

Esta definición es similar a la que podemos encontrar para los sistemas multiagente donde un conjunto de programas informáticos que actúan como una entidad autónoma, los llamados agentes, interactúan entre ellos y el entorno. Consiguen resolver un problema o lograr unos objetivos comunes al ser dotados de un comportamiento racional mediante técnicas de inteligencia artificial.

Unir estos dos conceptos en un solo ámbito permite crear representaciones de espacios visualmente convincentes en los que habitan entidades con aptitudes inteligentes capaces de perseguir sus propios objetivos, interactuar con el espacio y otras entidades (el usuario entre ellas) sin necesidad de indicarles cada acción que tienen que hacer. Al observar a otros personajes actuando por su cuenta se aumenta aún más la sensación de inmersión en un entorno virtual.

El desarrollo de entornos virtuales inteligentes está dividido entre la creación de los mundos interactivos y de los cuerpos y objetos gráficos; y el diseño del modelo de agentes y comportamiento de las entidades.

En este trabajo, se estudiarán diferentes herramientas de modelado de sistemas multiagente y, concretamente, se realizará un análisis de JaCaIVE, proponiendo un diseño de herramienta que se adecue a sus características, poniendo en práctica lo destacado de los ejemplos.

a. Objetivos

En este apartado se definirán los propósitos de este trabajo, los objetivos generales que se desean cumplir. Para conseguirlos, se identificarán también unos objetivos específicos que sirvan de pasos intermedios para lograr las metas finales.

El objetivo general es conseguir diseñar e implementar una interfaz de usuario compatible con el *framework* de entornos virtuales inteligentes JaCaIVE para su fase de modelo. Para ello se han establecido los siguientes objetivos específicos:

- Comparar diferentes tipos de desarrollo de *software* y determinar, de entre ellos, el proceso adecuado a este trabajo.
- Estudiar el modelo propuesto en JaCaIVE y realizar una síntesis de las características más importantes con el fin de especificar las funcionalidades básicas de la aplicación.
- Proponer un diseño de la estructura de la aplicación que se adecue a los requisitos reunidos en el objetivo anterior.
- Implementar una solución que se adapte al diseño propuesto

b. Estructura

A lo largo de este documento se va a describir el proceso que se ha seguido para alcanzar el cumplimiento de los objetivos. Pero antes de entrar en esos detalles, en el segundo apartado se procede a mostrar un estudio del marco teórico que rodea el área de estudio de este trabajo, comentando la situación de los entornos virtuales inteligentes hasta llegar al análisis de otras herramientas de modelado de sistemas multiagente.

A continuación, la memoria sigue con el desarrollo del proyecto comenzando en el apartado tres con la selección de un proceso de *software* a seguir durante la construcción de la aplicación y es en el apartado cuatro donde el método decidido se lleva a cabo. Dicho apartado se subdivide en tres correspondiendo a la descripción del diseño, la arquitectura y estudio detallado de la aplicación.

Será en el apartado cinco donde se explique la implementación de los diseños especificados en el apartado anterior. Se encontrarán en el apartado seis las indicaciones para el uso del *software* y en el siete se comentarán los casos de prueba de la solución.

Finalmente, la memoria sigue en el apartado siete con la evaluación de la ejecución de los objetivos y de los conocimientos que se han reforzado o adquirido en este trabajo y concluye en el apartado ocho con unas reflexiones sobre el trabajo futuro.

2. State-of-the-art

Previo al desarrollo del proyecto, se van a describir los elementos que están relacionados con él para dar a conocer una visión del entorno que da una motivación a esta herramienta.

Comenzando con el mundo de los entornos virtuales inteligentes se explican las disciplinas que los componen, centrando la atención en la parte de inteligencia artificial, y se cuentan algunos de sus usos y aplicaciones. A continuación, se explora el campo de los agentes, su definición, características y algunos tipos que pueden encontrarse.

Para la última parte, se ha realizado una búsqueda de herramientas de modelado de sistemas multiagente donde se han encontrado características de creación de modelos que se han considerado que servirán de ayuda para el diseño de la aplicación. Primero se listarán estos programas y se ofrecerá una descripción base de la organización de modelado y de su forma de creación de modelos para, finalmente, analizar y estudiar esos métodos con el fin de determinar el más adecuado a este trabajo.

a. Entornos virtuales inteligentes

Uno de los resultados del avanzado desarrollo de la computación es el hecho de que está consiguiendo salir del ámbito especializado de la investigación puramente informática. Así pues, se ha conseguido que su aplicación llegue hasta un gran número de campos de investigación, donde incluso varias disciplinas pueden llegar a unirse combinando conocimientos y metodologías, ampliando el área de estudio de forma que el beneficio mutuo ayude al desarrollo de las investigaciones individuales y alcance nuevos objetivos. Así sucede con la fusión, principalmente, de la inteligencia artificial (IA) y la realidad virtual (RV) de la surgen los conocidos como entornos virtuales inteligentes (IVE, del inglés, *intelligent virtual environments*).

Los IVE hacen uso de las metodologías y herramientas utilizadas en inteligencia artificial para desarrollar sistemas multiagente combinadas con las prácticas y los medios de la computación gráfica para crear representaciones gráficas interactivas. En [1] y [2] se recopilan ciertos factores que han llevado a ambos campos a la búsqueda de otros elementos que mejoren las funcionalidades de sus investigaciones o incluso pueden agregar otras nuevas.

Por una parte, el aumento de la capacidad computacional que ha mejorado las técnicas de informática gráfica y la calidad de las imágenes en 3D hace posible llegar hasta el realismo virtual e ir más allá al añadir características inteligentes. Dichas cualidades complementan a los entornos virtuales, que mantienen un mundo estático y predecible, incorporando, por ejemplo, un comportamiento autónomo a las entidades que allí habitan o tecnologías de reconocimiento de lenguaje natural.

La inteligencia artificial aporta a estos modelos gráficos un comportamiento racional en temas de resolución de problemas, adaptabilidad del sistema a la interacción del usuario, interacción a más alto nivel (por ejemplo, mediante lenguaje natural), definición de emociones y personalidad o abstracción del modelo visual (representación conceptual de la escena).

Por otro lado, la evolución de las técnicas de uso y tratamiento de la información dedicada a la computación gráfica puede servir para crear representaciones visuales adecuadas a los complejos modelos de estudio o entornos de investigación actuales.

De esta forma, se obtiene una simulación en un mundo virtual realista que ofrece interacción a tiempo real y de una forma más natural, facilitando la experimentación con modelos basados en agentes al evitar los inconvenientes mecánicos (montaje, movilidad, sensores, etc.) que conlleva el uso de un robot en un espacio físico real.

Algunas de las aplicaciones que podemos encontrar de los entornos virtuales inteligentes son:

- En el ámbito de la medicina, la realidad virtual ha podido encontrar su lugar con diferentes tipos de aplicaciones [22], estando dirigidos al aprendizaje, a la práctica de procedimientos, al diagnóstico o al tratamiento. Cada uno de estos necesita más algún aspecto de los RV que otros.

Ilustración 1. Acadicus.

  Por ejemplo, situaciones destinadas al diagnóstico requerirán una fidelidad a un cuerpo real basados en imágenes médicas y datos extraídos de pruebas como de tomografías computarizadas (TC) o de resonancias magnéticas (IRM) y la capacidad de navegación por esas recreaciones.

  Para otros casos es útil el sentimiento de inmersión característico de la RV mediante herramientas de interacción natural como gafas o guantes y la precisión que se pueda conseguir con esos instrumentos. Este último tiene que ver con el entrenamiento de procedimientos, de cirugías o realización de pruebas que ofrezcan una simulación monitorizable y programable.

  La inteligencia artificial puede aportar aquí el poder de añadir comportamiento a los pacientes o evolución de las enfermedades y así responder de forma correcta a las acciones del usuario. Existen varios programas que han empezado a poner en práctica estos métodos, como Acadicus[1] [Ilustración 1], SimX[2] o RCSI Medical Training Sim[3].

  También en el ámbito de la psicología [23] pueden aplicarse estas características, desde el lado de los profesionales, que podrían realizar prácticas de diagnóstico utilizando simuladores de conversación con agentes

[1] https://acadicus.com/
[2] https://www.simxar.com/#home
[3] https://msurgery.ie/vr

humanos virtuales. Y desde el lado de los pacientes, como asistentes virtuales que puedan dar información de forma constante y siempre disponible. Por ejemplo, SimCoach[4] [Ilustración 2].

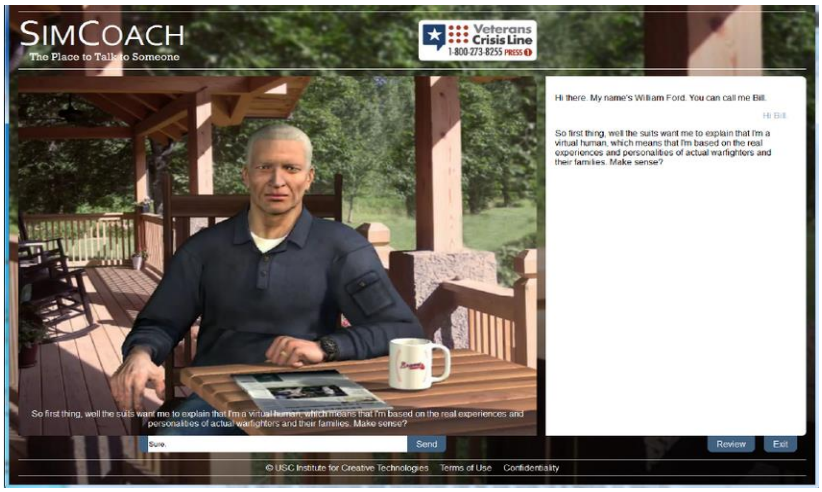

Ilustración 2. SimCoach

- Crear grandes grupos de agentes que circulan por una ciudad o espacio determinado y controlar el flujo de movimiento en condiciones normales y extraordinarias. Esto se conoce con el nombre de simulación de multitudes (*crowd simulation*).

  Comúnmente, los agentes del modelo representan a humanos, aunque también pueden encontrarse como vehículos que circulan por carreteras urbanas o entre ciudades. Las simulaciones destinadas a la observación del conjunto de agentes en movimiento [30] no suelen ser de carácter inmersivo ya que su objetivo es determinar cómo se mueven estos agentes por el entorno.

  Es una práctica útil en arquitectura para testear edificios que soportarán un uso diario de muchas personas (por ejemplo, centros comerciales o museos) y situaciones de evacuación en entornos tanto cerrados como abiertos. Las situaciones de evacuación tratan de medir tiempos de respuesta de reacción y evacuación, eficacia de las salidas disponibles y daños y pérdidas provocadas.

  El comportamiento de los agentes en situaciones de emergencia debe pretender simular emociones humanas como pánico o miedo, ya que estas pueden afectar a su capacidad reactiva, de igual forma que ocurre con las personas. Algunos experimentos se han llevado a cabo utilizando agentes BDI [26], que se explicarán con detalle más adelante, para lograr un acercamiento inteligente a la actuación humana. MassMotion[5] [Ilustración 6], Evacuate[6], CrowdMaster[7].

4 https://www.simcoach.org/
5 https://www.oasys-software.com/products/pedestrian-simulation/massmotion/
6 http://www.evacuate.eu/
7 http://crowdmaster.org/

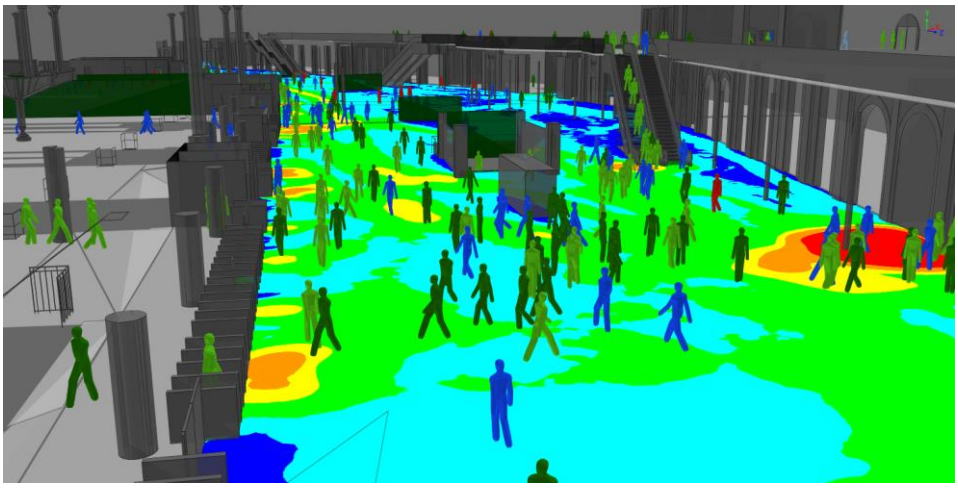

Ilustración 3. MassMotion.

Otros trabajos con masas de agentes [9] se han realizado con el fin de estudiar aquellos aspectos que hacen que un agente virtual parezca más real a ojos del usuario. Se busca en qué medida la autonomía de los agentes de moverse por el entorno, interactuar con él, con otros agentes o con el usuario resulta creíble.

Por ejemplo, agentes que son capaces de evitar colisiones, de mantener contacto visual o atender al espacio personal son más convincentes al ojo humano. Estos estudios sí pueden ser probados con técnicas de inmersión, y este aspecto da más riqueza a la investigación ya que pueden evaluarse las emociones y/o reacciones fisiológicas que se dan durante la simulación.

- La industria de los videojuegos ha estado haciendo uso cada vez más y más en el uso tanto de la inteligencia artificial [14] como de la realidad virtual. El uso de técnicas como máquinas de estado finitas para determinar el comportamiento del entorno y de personaje no jugables resulta sencillo de implementar y eficaz para la experiencia de juego, pero para jugadores más expertos suelen resultando en acciones predecibles.

  Aunque hoy en día sigan utilizándose estos métodos, el avance de la IA puede aplicarse a juegos cada vez más complejos que buscan ofrecer más dificultad o más sensación de realismo en el juego. A medida que se busca mejorar en estos aspectos, se necesitan otras técnicas de inteligencia artificial, como los agentes inteligentes, que puedan llevar un control de la situación y de las acciones aceptadas en base a ello.

Ilustración 4. Los Sims 4

Por ejemplo, juegos de acción y/o aventura que implican batallas o peleas con enemigos y junto a aliados virtuales como la saga de rol de mundo abierto The Elder Scrolls[8] o el juego de disparos en primera persona Overwatch[9]. También se encuentran en juegos de simulación, como Los Sims[10] [Ilustración 4] o Animal Crossing[11], donde los personajes no jugables atienden a unos rasgos de personalidad definidos y hablan y actúan en consecuencia junto al jugador que participa controlando un avatar.

Aunque la producción de juegos de realidad virtual es pequeña en comparación a la total [5], poco a poco va aumentando su número [27] y tanto ellos como los dispositivos y consolas tienen buena aceptación en el mercado. Entre los juegos disponibles se encuentran de diversos géneros como aventuras (Chronos[12]), simulación (Vacation Simulator[13]) o musicales (Beat Saber[14] [Ilustración 5]).

Ilustración 5. Beat Saber.

[8] https://elderscrolls.bethesda.net/es/
[9] https://playoverwatch.com/es-es/
[10] https://www.ea.com/es-es/games/the-sims/the-sims-4/pc/store/mac-pc-download-base-game-standard-edition
[11] http://www.animal-crossing.com/es/
[12] https://gunfiregames.com/chronos
[13] https://vacationsimulatorgame.com/
[14] https://beatsaber.com/

Como se ha mencionado anteriormente, la parte de inteligencia artificial se centra en el uso de los agentes autónomos como entidades que habitan en un mundo virtual y son capaces de interactuar en él. Desde este punto de vista de los IVE, Ruth Aylett y Michael Luck [2] definen un espectro de clasificación de los agentes según si se centra en temas físicos o cognitivos.

El término espectro se señala como idea de que no se establece una división excluyente entre tipos de agentes, sino que cada uno puede incluir características del otro. Un agente ideal tendría perfeccionadas tanto las aptitudes físicas como las cognitivas, aquellas relacionadas con el conocimiento. Sin embargo, en algunas ocasiones es más importante un aspecto que otro, y la dificultad de conseguir la coordinación de ambos lleva a centrarse más en uno que en otro e implementar las funciones básicas de la otra parte.

Por un lado, el desarrollo de agentes físicos tiene como finalidad dar credibilidad al comportamiento gráfico dentro del entorno virtual. Participan en este lado del espectro los estudios relacionados con la realidad virtual cuya finalidad es construir un espacio 3D para la inmersión del usuario y los elementos que lo componen como si se tratara de un mundo real, mejorando la inmersión. Entre los principales puntos a tratar encontramos:

- Simulación de las características físicas del mundo y que afectan a los objetos, como la gravedad, rozamientos, texturas, etc.

- Movimiento realista de los agentes como entes vivos, detallando lenguaje corporal, gestos, expresión facial, etc. Hay que tener en cuenta que también son afectados por los aspectos del punto anterior.

- Interacción entre el entorno y sus objetos con los agentes virtuales. Los agentes interaccionan con el entorno de dos formas: mediante sensores obtienen información del espacio, sus elementos y características (interacción pasiva) y mediante actuadores son capaces de alterar el estado del entorno (interacción activa).

  A través de detección de colisiones y control de contacto entre superficies se consigue que los agentes puedan explorar el entorno virtual y recibir retroalimentación de sus acciones.

En el otro extremo, se encuentra la parte que busca potenciar las capacidades cognitivas, inteligentes, de las entidades virtuales. También se intenta dar una sensación de credibilidad gracias a la actuación del agente. En este aspecto se define una interacción con el entorno donde la información obtenida está destinada a determinar el comportamiento de los agentes.

Al ser un sistema creado por un diseñador, los agentes podrían saber todo lo que necesitan de forma previa a la simulación, lo que provocaría una conducta predecible y estática. Pero si tanto del entorno como de los componentes físicos se espera una simulación de la realidad, los agentes que habitan en él deberían mostrar un comportamiento dinámico adecuado.

A lo largo de este apartado, se ha mencionando continuamente a los agentes como las entidades que encontraremos en un entorno virtual. A continuación, se va a proporcionar una definición de agente que justifique su uso en los IVE y se extenderá su estudio hasta las aplicaciones de creación de sistemas multiagente.

b. Sistemas multiagente

Hoy en día, el concepto de agente aún no tiene una descripción global (incluso dentro de la IA), pues según el fin que se quiera obtener, un agente debe mostrar una serie de características u otras, con más o menos importancia. Por ejemplo, Russell y Norvig [24] identifican a un agente como “cualquier cosa capaz de percibir su medioambiente con la ayuda de sensores y actuar en ese medio utilizando actuadores”, dejando poca distinción con cualquier programa informático que recibe entradas de teclado y muestra resultados por pantalla.

Sí que establece una separación al entrar en el ámbito de los agentes racionales indicando que responden a “aquel que actúa con la intención de alcanzar el mejor resultado o, cuando hay incertidumbre, el mejor resultado esperado [...] debe ser autónomo, debe saber aprender a determinar cómo tiene que compensar el conocimiento incompleto o parcial inicial” [24].

Este último aporte se asemeja más a otras definiciones, como la de Wooldridge [31]: “sistema informático que se sitúa en un entorno, y es capaz de actuar de forma autónoma en él para cumplir sus objetivos de diseño", en las que la autonomía es un requisito fundamental en la esencia de un agente. De ahí que cuando se habla de agentes racionales, se puede terminar en extensiones que muestran otras habilidades tales como reactividad, proactividad o habilidades sociales.

Por lo que se refiere a la autonomía, la habilidad para decidir la forma de lograr un objetivo sin haber recibido órdenes directas de cómo hacerlo [24]. Dentro de los IVE, la autonomía aporta al agente ese comportamiento natural que se buscaba para representar las limitaciones que un agente en el mundo real tendría a la hora de recoger información y formar un modelo de la información obtenida desde el que llevar a cabo todo el proceso de razonamiento, aprendizaje, planificación y toma de decisiones.

El estudio del uso del conocimiento en los agentes a nivel individual explora la arquitectura con el objetivo de integrar percepción, planificación y actuación. Determina los mecanismos que utiliza un agente para reaccionar a los estímulos y eventos del entorno. A continuación, vemos algunas de estas arquitecturas:

- Reactivas. Este tipo basa su funcionamiento en el principio acción-reacción. Básicamente, estos agentes se centran únicamente en sus percepciones sobre el entorno actual, y actúan en respuesta a ellas, sin tener en cuenta ningún tipo de historial. Las relaciones entre el estímulo y la acción a realizar se denominan reglas.

  En [Ilustración 6] se muestra un esquema simple donde se observa que la información recibida de los sensores se evalúa según las reglas definidas y se eligen las acciones en consecuencia. En los agentes reactivos es más importante la capacidad de identificar de manera rápida cualquier elemento inesperado e identificar la forma de actuar.

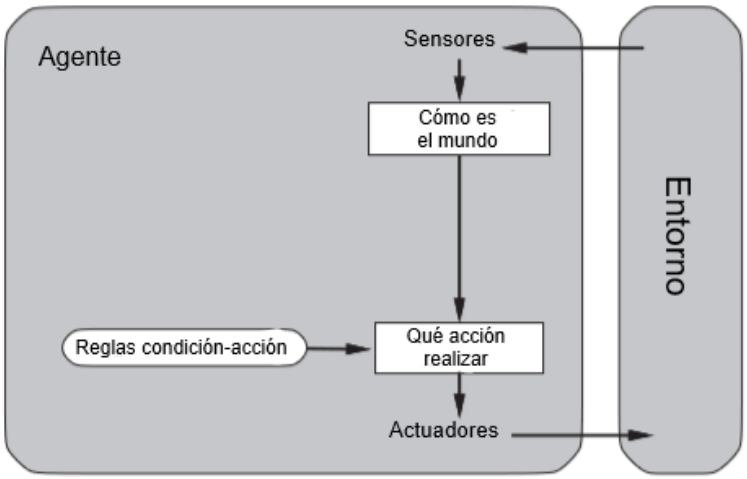

Ilustración 6. Arquitectura de agente reactiva

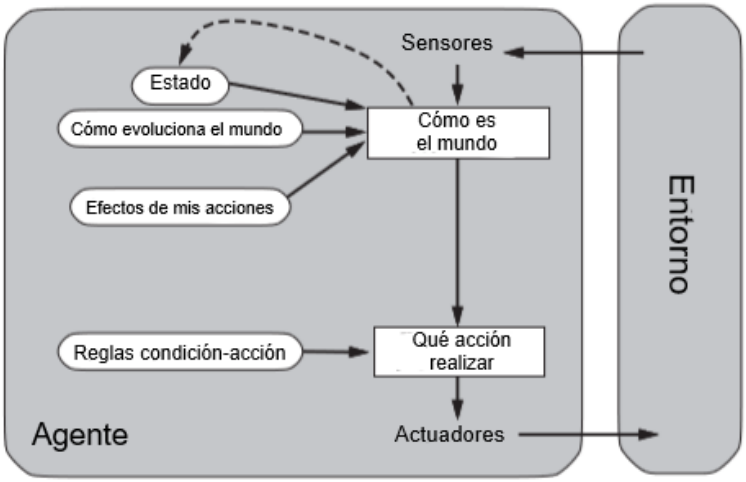

Ilustración 7. Arquitectura de agente deliberativa

- Deliberativas. Estas arquitecturas utilizan modelos de representación simbólica del conocimiento, es decir, se mantiene un estado interno del entorno y su historia. El agente interpreta esa información para intentar predecir los efectos de sus acciones y así deducir el siguiente paso. En [Ilustración 7] se observa el esquema visto en la arquitectura reactiva, con la adición de los elementos que determinan una arquitectura deliberativa: la actualización de un estado y un estudio de posibles efectos derivados de una posible acción.

  Una de las arquitecturas de carácter deliberativo más extendidas es el modelo BDI (*Belief-desire-intention*) de razonamiento práctico dirigido al futuro. Define tres componentes principales: las creencias (*beliefs*) representan el estado que se tiene el entorno o sistema; los deseos (*desires*) representan los objetivos, estados finales deseados, que el agente debería intentar conseguir; y las intenciones (*intentions*) lo que el agente ha elegido hacer.

- Híbridas. Así como hemos clasificado los agentes en los IVE como un espectro donde es posible (y lo común) mezclar características de los extremos, las

arquitecturas de agentes también pueden combinar sus estructuras. En consecuencia, un agente mixto juntará los beneficios de un agente reactivo (simplicidad, rapidez de respuesta o sencillez computacional) con los de un agente deliberativo (visión global, capacidad de aprendizaje o planificación de objetivos).

En este punto, ya se ha conseguido describir una perspectiva de lo que representa un agente y cómo funciona, no obstante, la implementación de un agente individual no es suficiente para abarcar la totalidad de lo que se espera. “No hay tal cosa como un sistema de un único agente”, cita Wooldridge [31] como un lema común en el desarrollo de sistemas multiagente (MAS, del inglés, *Multi Agent System*) para destacar que el mundo de los agentes funciona con ellos en comunidad. Desde este nuevo punto de vista, se contemplan otras habilidades para los agentes como comunicación, coordinación o colaboración.

Esta definición coincide con la proporcionada por en [12], “[...] grupos de agentes que interaccionan entre sí para conseguir objetivos comunes.”. En su tesis, además, Gómez Sanz indica una serie de elementos definidos por Ferber que están presentes en un MAS. En resumen, estos componentes son: un entorno, un conjunto de objetos perceptibles pasivos, un conjunto de agentes (objetos especiales) como entidades activas, las relaciones que los unen y un grupo de operaciones de percepción y actuación, así como de la reacción que su aplicación provoca en el mundo.

c. Herramientas de modelado

A continuación, con el fin de estudiar las diferentes formas que existen de definir estos elementos en una sociedad de agentes, se han buscado ejemplos de herramientas de modelado de MAS para analizar sus características más importantes.

De todas las herramientas que se han observado para este proyecto, la siguiente selección está compuesta por aquellas en las que se han encontrado características sobre la creación de modelos que se deseaban analizar teniendo en cuenta la idea de la aplicación que se tiene en mente.

Esta idea se desarrollará más adelante, pero, en resumen, es la de una aplicación que genere un archivo XML con los datos introducidos, evitando así tener que escribir, editar y visualizar ese código a mano. Por lo tanto, se busca una interfaz sencilla que haga cómodo el hecho de crear un modelo y ese es uno de los aspectos que se ha tenido en cuenta para las siguientes herramientas.

Por ejemplo, se han descartado lenguajes de programación, como SARL [15] o AgentSpeak (actualmente interpretado por Jason[16]). Otras que se vieron con un poco de profundidad, como AgentSheets[17] o AnyLogic[18], se descartaron finalmente por considerar que la creación de modelos estaba más orientada a diseñar simulaciones en 2D o 3D, aunque también integran métodos de inteligencia artificial y basados en

[15] http://www.sarl.io/
[16] http://jason.sourceforge.net/wp/
[17] https://www.agentsheets.com/store
[18] https://www.anylogic.com/

agentes. También se eliminó JADE[19] de la selección por estar más dirigido a soportar la comunicación entre agentes de forma distribuida.

Ilustración 8. ZEUS

- ZEUS[20] [4] [6] [7] [Ilustración 8] es una herramienta de propósito general para el desarrollo de sistemas de agentes colaborativos. Uno de sus principales objetivos consiste en poder ser utilizado incluso con conocimientos básicos sobre teoría de agentes, por eso se basa el estilo de la aplicación en la programación visual.

  Para conocer a grandes rasgos la organización de modelo básica que se puede crear en ZEUS, se presentan los cinco pasos propuestos en [3] para la construcción de un MAS, cada uno de ellos se centra en un aspecto clave de la especificación en ZEUS.

  1. Creación de la ontología, conjunto de declaraciones que definen los fundamentos.

  2. Creación de los agentes que habitan en el sistema. A estos agentes son los que se les asignan unas acciones y relaciones determinadas.

  3. Configuración de los agentes que gestionan utilidades del sistema como la visualización, base de datos o agenda de nombres.

  4. Configuración de las tareas de los agentes.

  5. Generación de código.

  A través de una interacción mediante menús y ventanas, ZEUS ofrece al usuario una amplia visión general del modelo y una fácil accesibilidad a la información de detalle. Al ejecutar la aplicación, ZEUS presenta una primera ventana con los elementos principales del modelo: proyecto, ontología, agentes y tareas.

  Al acceder a la edición de cada elemento se da el paso a una única ventana para su configuración individual, estas secciones organizan la información en

[19] https://jade.tilab.com/
[20] https://sourceforge.net/projects/zeusagent/

paneles agrupados por categorías en pestañas. Existe una ventana dedicada a la representación gráfica del modelo de sociedad de agentes que se ha ido creando en la interfaz.

Ilustración 9. NetLogo

- *NetLogo*[21] [16] [Ilustración 9] es un entorno de modelado destinado a la implementación y simulación de sistemas basados en agentes para estudio e investigación de fenómenos sociales o naturales. Aunque sus capacidades han llegado hasta la creación de modelos para estudios importantes y complejos, NetLogo siempre ha destacado su orientación a la enseñanza. Características y componentes de modelado:

    - Los agentes móviles que habitan en un entorno se llaman tortugas (*turtles*). Se definen por una posición una dirección. Tienen una forma y un tamaño determinado. Pueden ser visibles u ocultos.

    - Los agentes estáticos llamados parcelas (*patches*). Forman una cuadrícula en el mundo.

    - Los agentes *links* sirven para crear relaciones y enlaces entre *turtles*.

    - El agente observador es el de mayor jerarquía y es invisible. Da órdenes a otros agentes, respondiendo a las instrucciones del usuario desde la ventana del observador.

    - Los *procedures* son series de instrucciones en un único comando definido por el usuario.

    - El entorno de trabajo está definido por un espacio y tiempo discretos.

    - Se pueden almacenar variables globales (solo un valor), de agente (cada agente tiene su propio valor) y locales (usadas solo en el contexto de una función).

  El principal método para desarrollar los sistemas es utilizando el lenguaje de programación NetLogo, basado en el lenguaje Logo[22]. Puede utilizarse en la

[21] https://ccl.northwestern.edu/netlogo/
[22] https://el.media.mit.edu/logo-foundation/what_is_logo/index.html

aplicación a través de una interfaz de comandos realizando órdenes directas sobre el mundo o utilizar un estilo funcional para definir métodos (los procedures) en la pestaña de edición de código. Para conocer la distribución de agentes y su comportamiento antes de la simulación es necesario comprender el lenguaje e identificar en el código las definiciones y usos de los agentes y variables.

Otra de las herramientas que ofrece NetLogo es el modelado de mediante diagrama de dinámicas que también genera código. El uso de este editor tiene una finalidad diferente al uso que se da normalmente de NetLogo, el de especificar el comportamiento individual de cada tipo de agente, más fiel al enfoque de los sistemas basados en agentes. Con el modelado de dinámicas, la idea es definir el comportamiento del grupo que conforma la totalidad de agentes de un tipo.

Ilustración 10. Repast

- Repast Simphony[23] [Ilustración 10] es un *plug-in* de Eclipse[24] para el desarrollo de sistemas basados en agentes. Permite la construcción de los modelos mediante el uso de diversos lenguajes de programación como ReLogo, Java, Groovy o programación visual mediante diagramas de estado. El primero de estos está basado en el lenguaje dinámico Groovy, utilizando los elementos semánticos de Logo.

  Para implementar modelos utilizando Repast es fundamental tener conocimientos de programación de cualquiera de estos lenguajes. Sin embargo, Repast permite un tipo de programación visual basado en diagramas de actividad que puede servir de apoyo para diseñar el comportamiento de un agente.

- CORMAS[25] (del inglés, *COmmon-pool Resources and Multi-Agent Simulations*) [3] [Ilustración 11] es una herramienta de diseño de sistemas basados en agentes dirigida a la gestión de recursos naturales. Intenta reducir al mínimo la cantidad de código programado por el usuario. Por ello, la aplicación está basada en un alto nivel de utilización de ventanas y menús específicos donde el usuario utiliza tablas, listas y botones para seleccionar y modificar las características y propiedades, dejando al modelador únicamente la programación en SmallTalk de los métodos de comportamiento de las entidades.

[23] https://repast.github.io/
[24] https://www.eclipse.org/
[25] http://cormas.cirad.fr/indexeng.htm

Ilustración 11. CORMAS

Los tres pasos que seguir propuestos para la creación de un modelo son:

1. Creación e implementación de clases. Aquí se puede observar una clara división de las posibles entidades a crear: grupo de agentes sociales, grupo espacial y un grupo de entidades pasivas destinado a la definición de mensajes, conexiones, etc. Desde cada uno de los agentes creados se accede a la configuración de atributos y métodos.

2. Inicialización del modelo y evolución de la simulación. Similar a la definición de métodos de los individuos, este paso especializa la implementación de los métodos para definir el estado de inicio del modelo y los pasos del tiempo. En esta fase, se tiene acceso también a una lista de todos los atributos del modelo y al editor de diagramas de actividad que estudiaremos en el próximo apartado.

3. Configuración de la visualización.

A continuación, se describen las características, algunas de ellas pueden encontrarse en [Ilustración 12]:

- *CormasModel* es la clase abstracta organizadora de todo el modelo. Al crear uno nuevo, se especializa un objeto de esta clase para personalizar el funcionamiento predeterminado del sistema.

- Existe una clara división de las posibles entidades a crear: grupo de agentes sociales, grupo espacial y un grupo de entidades pasivas destinado a la definición de mensajes, conexiones, etc.

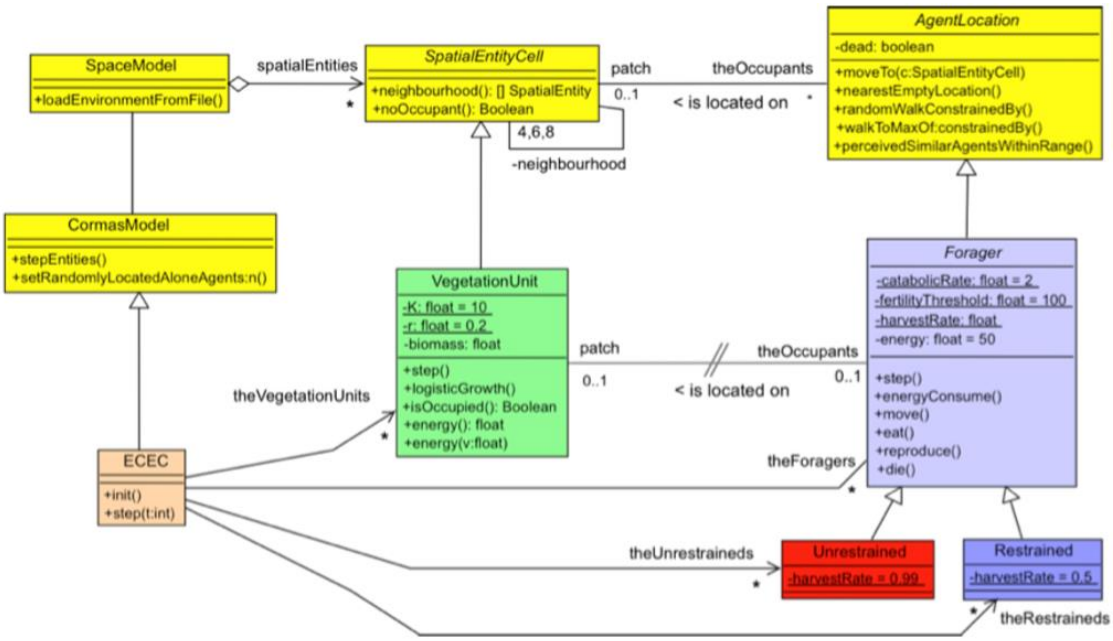

Ilustración 12. Diagrama de clases de un modelo en CORMAS

- Para todo el primer grupo, el de los agentes, existen clases genéricas con acciones obligatorias y comunes en los agentes como la inscripción en el registro global, la capacidad de moverse y percibir el entorno (clase *AgentLocation*) o la comunicación mediante mensajes (clase *AgentComm*).

- En el caso de las entidades que pertenecen al entorno, CORMAS proporciona la clase abstracta *SpatialEntity*, que puede derivar en dos tipos: el tipo básico (las celdas del entorno) y demás elementos que se puedan añadir (CORMAS permite establecer nuevos elementos individuales o compuestos).

- De acuerdo con la propuesta *ComMod*, CORMAS también busca la participación del usuario en el desarrollo de la simulación. Para ello, es posible definir el punto de vista del agente, interacción con el entorno o instrucciones a seguir. Además, gracias a que la simulación es capaz de ser ejecutada de forma distribuida en varios ordenadores, más de un usuario interactivo puede actuar simultáneamente.

Por último, cabe destacar la creación de diagramas dinámicos de estado para definir el comportamiento global de las entidades sin tener que escribir código. Estos diagramas hacen uso de los métodos definidos en el sistema para organizarlos de una forma sencilla: solo dispone de un punto de inicio, un punto final y dos tipos de actividades (de transición y de decisión).

Ilustración 13. GAMA

- GAMA [26] (del inglés, GIS (*Geographic Information System*) *Agent-based Modeling Architecture*) [29] [Ilustración 13] es un entorno de desarrollo y simulación de sistemas basados en agentes dirigido a la implementación de sistemas geográficos. La aplicación de GAMA está basada en la organización de interfaz de los entornos de desarrollo (IDE, del inglés, *Integrated Development Environment*), concretamente de Eclipse, por lo que consta de una única ventana de implementación donde navegar por los proyectos y archivos de código.

  La construcción de los sistemas se realiza con el lenguaje orientado a agentes propio de GAMA llamado GAML. Un archivo para un modelo tiene una estructura definida en tres secciones principales:

  - Definición de los agentes, denominados especie, que habitan en el mundo. Cada agente tiene un conjunto de atributos, acciones y comportamientos (acciones especiales que se ejecutan a cada paso de tiempo). Los agentes en GAML también soportan herencia, uso de habilidades (conjuntos de atributos y métodos predefinidos) y comunicación/interacción entre agentes.

  - Configuración del entorno, que funciona como un agente único en todo el modelo, con sus atributos y métodos globales. Dentro de este apartado se puede, por ejemplo, instanciar el número inicial de agentes.

  - Los experimentos. De cada modelo pueden existir varias simulaciones diferentes, cado uno de ellos con sus propios parámetros de entrada o las salidas (representaciones visuales, gráficas de datos o archivos).

[26] https://gama-platform.github.io/

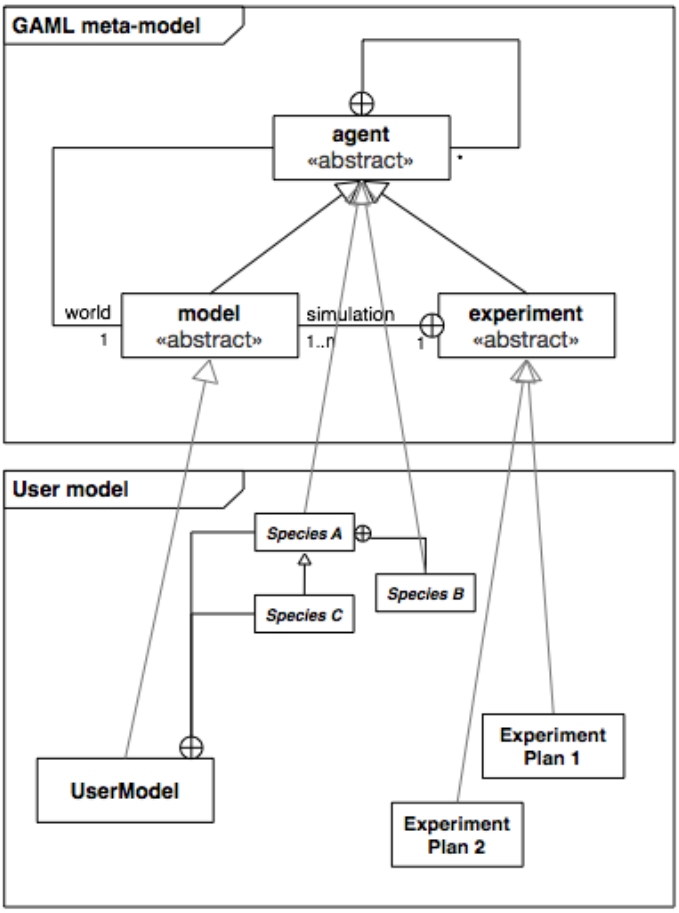

Ilustración 14. Diagrama de clases de un modelo en GAMA

En [Ilustración 14] se encuentran, en el cuadro superior, los tres elementos principales descritos en el párrafo anterior (agentes, modelo y experimentos; y las relaciones entre ellos) y, en el cuadro inferior, un ejemplo de modelo que muestra la especialización de sus componentes con relación al metamodelo.

d. Crítica del *state-of-the-art*

Como se ha indicado en la introducción y en los objetivos al principio de este documento, la aplicación final gira en torno al proceso de diseñar un modelo de sistema multiagente basado en un metamodelo determinado, dejando de lado las partes de programación y simulación.

Por ello, al presentar las herramientas anteriores se quieren buscar características que sirvan de inspiración o crítica en los diferentes tipos de creación de modelos que se han estudiado

Dado que todas las herramientas requieren un mínimo de programación, en [Tabla 1] indicaremos el nivel de utilización de ese lenguaje a la hora de definir el modelo. De igual forma, en los diagramas indicaremos si los diagramas disponibles en la

aplicación si la creación de un modelo como definición de todos (o al menos, sus elementos, relaciones y propiedades) puede conseguirse a través de un diagrama.

	Programación	Diagramas	Ventanas y menús	Generación automática de código
ZEUS	Bajo	Sí	Sí	Sí
NetLogo	Medio	Medio	No	Sí
Repast	Alto	No	No	No
CORMAS	Bajo	No	Sí	No
GAMA	Medio	Sí	No	Sí

Tabla 1. Tabla comparativa de herramientas de modelado

Las únicas herramientas que utilizan un sistema de ventanas y menús para la creación del modelo, es decir, las ventanas pueden significar un área de especificación de una parte del modelo. Por ejemplo, en [Ilustración 15] se observa cómo ZEUS ofrece una ventana que enmarca toda la configuración de la ontología donde a través de botones, tablas y listas permite la creación y configuración de los atributos, agrupada por pestañas.

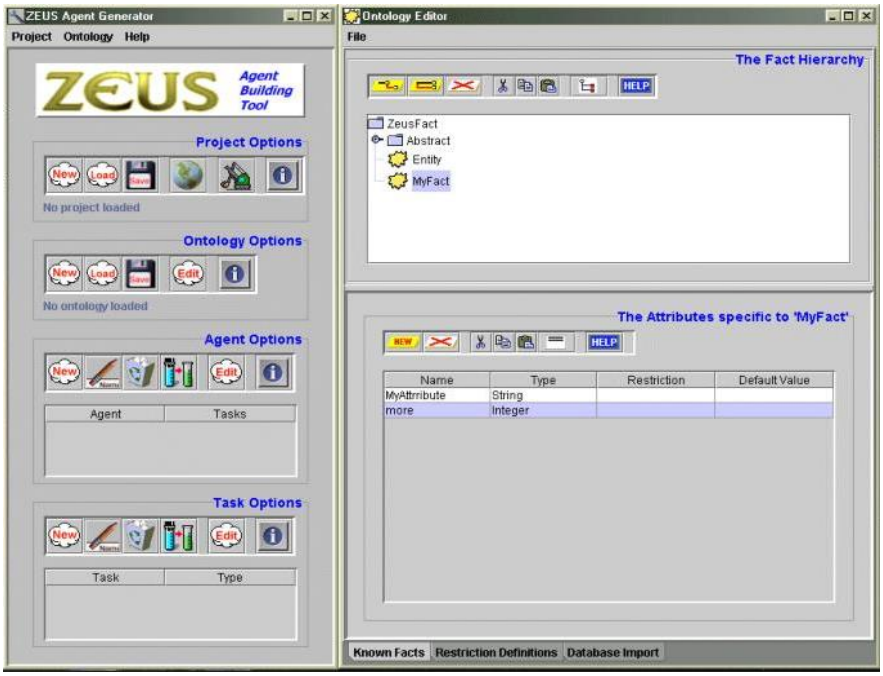

Ilustración 15. A la izquierda, ventana inicial de ZEUS. A la derecha, ventana de edición de ontología.

Con una buena organización y distribución de los menús y las ventanas, la creación del modelo puede terminar resultando intuitiva y sencilla. El caso de ZEUS, también en [Ilustración 15], es un ejemplo de la sencillez, pues desde la primera ventana se pueden observar todos los elementos necesarios y acceder a las configuraciones específicas de estos elementos.

Por el contrario, se ha encontrado CORMAS con una navegación de ventanas más compleja: el número de ventanas y menús es mucho más mayor y distribuidos en varios niveles. Es posible tener varias ventanas abiertas a la vez e ir seleccionando la activa, lo que permite una personalización del entorno de trabajo manteniendo las ventanas deseadas, pero puede desembocar en un entorno confuso y muy cargado.

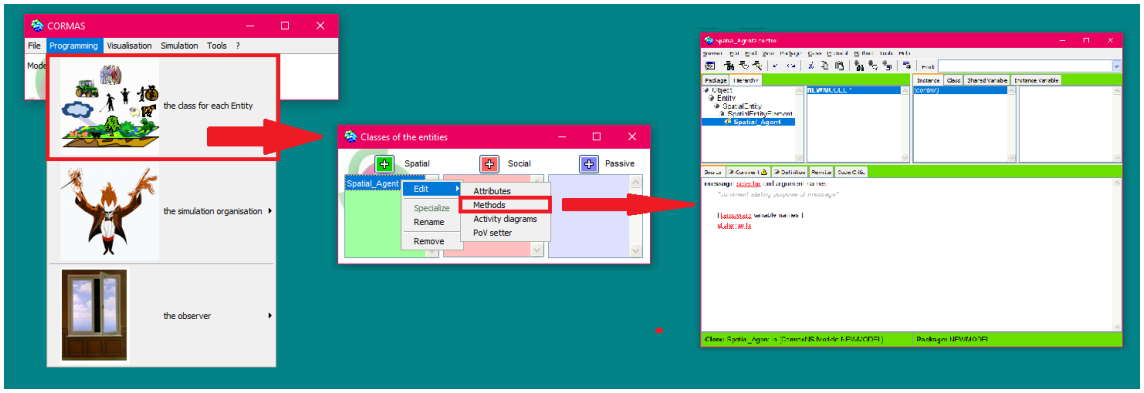

Ilustración 16. De derecha a izquierda: Menú desplegable de programación en la ventana inicial de CORMAS, ventana de edición de agentes y ventana de programación de agentes.

La interfaz de CORMAS también ha jugado con los colores para ayudar a la comprensión de la diferenciación de los elementos. En [Ilustración 16], la ventana situada en el medio es aquella donde se crean las clases de los diferentes tipos de agentes y puede apreciarse una distinción por colores que apoya la separación de listas.

Además, cuando se accede a la configuración de los métodos de un agente, esa ventana mantiene el mismo diseño para los diferentes tipos de agentes, pero el color acorde al establecido en la ventana de clases.

En cuanto a los diagramas, estos son eficaces a la hora de representar la organización de elementos y sus relaciones, pero a la hora de introducir atributos, como estos difieren en características y número según el elemento, es difícil encontrar una forma común y compacta de representación visual.

En GAMA se observa una solución, [Ilustración 17] y [Ilustración 18], a este aspecto combinando con ventanas de configuración para la edición de atributos e incluso permite la implementación de inicio o de los métodos *reflex*.

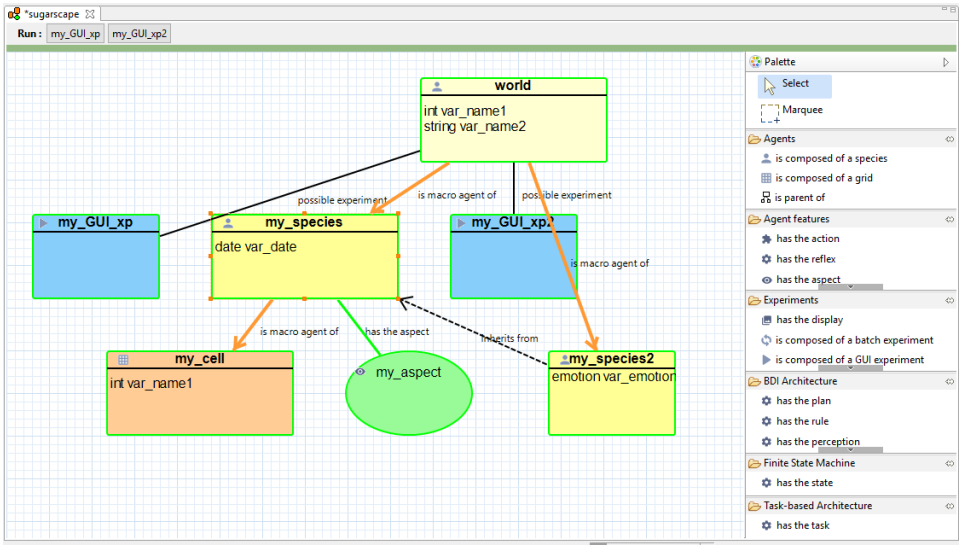

Ilustración 17. A la izquierda, editor de modelos mediante diagramas de GAMA

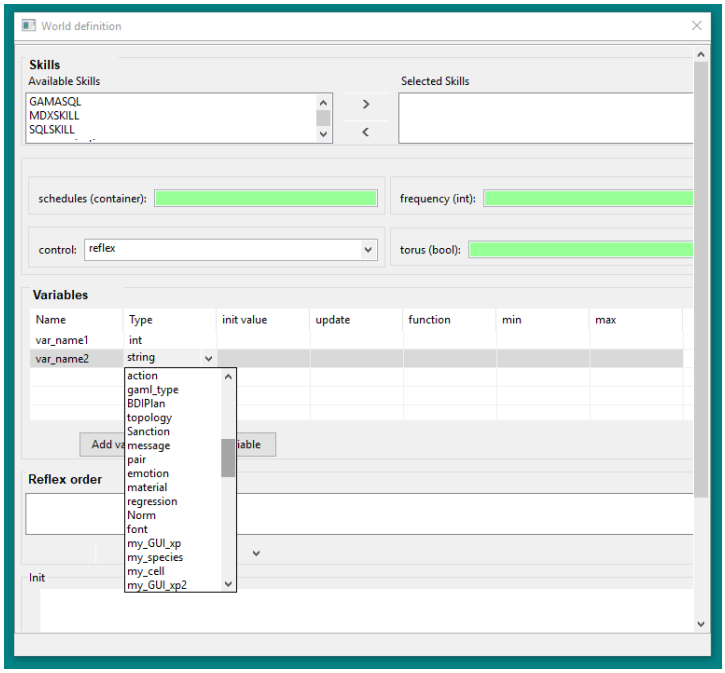

Ilustración 18. Ventana de edición del agente mundo.

EN GAMA un diagrama inicial por defecto cuenta con el nodo agente que representa el mundo y un nodo de experimento (que puede eliminarse para añadir otro de los tipos). Del nodo mundo se enlazan los demás agentes que luego podrán también relacionarse entre ellos.

En el diagrama pueden encontrarse, como mínimo, los tres elementos base que se habían definido para un modelo en GAML, vemos una clara correspondencia con el metamodelo. Aquí también encontramos diferenciación de colores, aunque no se aprecia la distinción en la ventana de configuración de un elemento.

Repast, de forma similar a GAMA, utiliza una combinación de diagrama y ventanas para ampliar la capacidad de modelado [Ilustración 19]. En este caso, las ventanas son divisiones dinámicas del espacio de la ventana principal, extendiendo las

propiedades de los elementos del diagrama en las ventanas del entorno de desarrollo. En estas propiedades se puede introducir código de comportamiento de entrada y salida del nodo y configurar el lenguaje de programación usado.

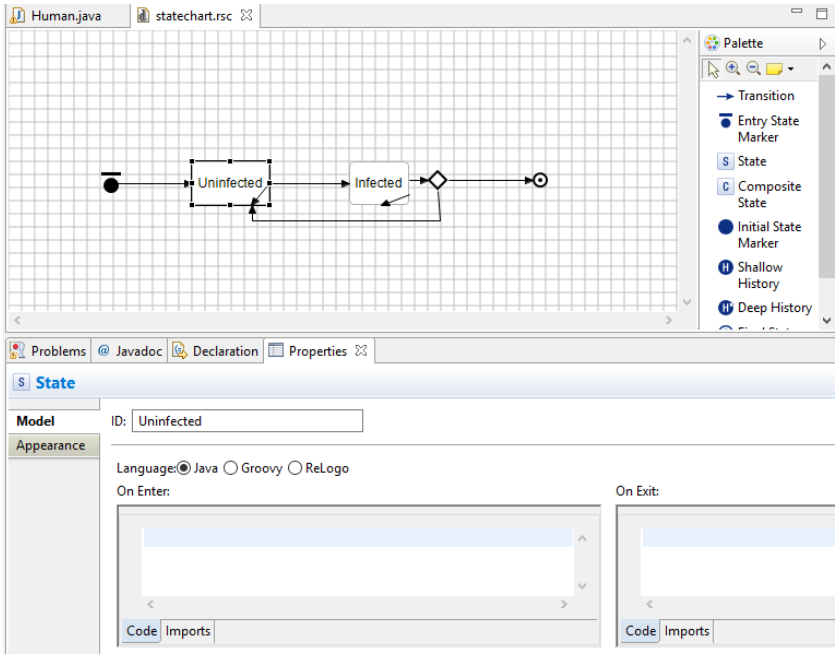

Ilustración 19. Arriba, editor de programación mediante diagramas de estado en Repast. Debajo, ventana de edición de un estado.

NetLogo ofrece una herramienta de creación de diagramas muy simple [Ilustración 20], con pocos elementos de construcción de esquema y poquísima configuración de los elementos (básicamente atributos y enlaces). No permite una amplia configuración de los elementos más allá de indicar el nombre y un valor de inicialización, esta parte se realiza completando el código que se genera automáticamente a partir del esquema.

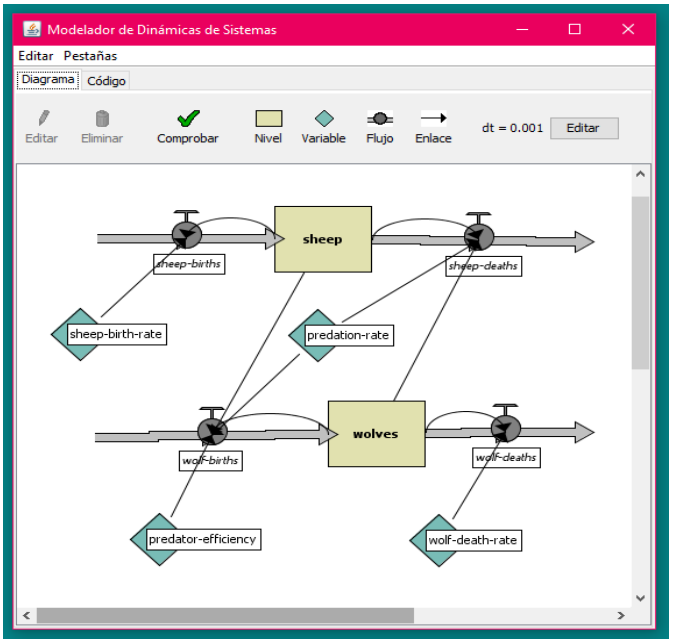

Ilustración 20. Ventana de edición de un modelo de dinámicas de sistemas en NetLogo.

Anteriormente, se ha comentado CORMAS desde el punto de vista de menús y ventanas, pero también encontramos un método de diagrama que, aunque no permite la creación total del modelo, hace uso de este para definir el comportamiento de las actividades definidas.

En [Ilustración 21] observamos una ventana de edición de diagramas simple (se componen básicamente de nodos de actividad y relaciones) donde la asignación de las actividades recoge las definidas en el modelo para su selección.

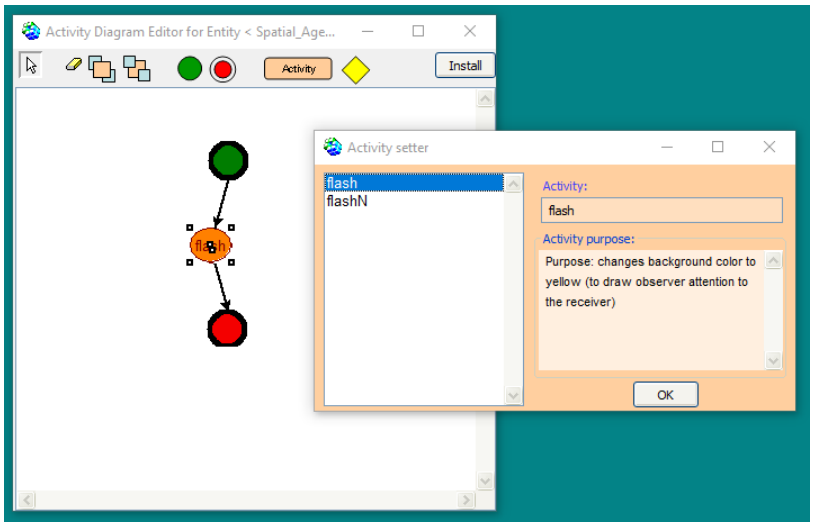

Ilustración 21. A la derecha, ventana de edición del diagrama de actividad de un modelo en CORMAS. A la izquierda, ventana de selección de actividad para el nodo seleccionado.

En términos de implementación, los diagramas son más complicados que las ventanas de menú pues implican definir representaciones visuales que se correspondan con el metamodelo con que se trabaja intentando abarcar lo máximo posible de sus elementos.

Como se ha observado con los ejemplos anteriores, el uso de elementos visuales proporciona un esquema global del modelo, pero suele llevar a una pérdida de representación de características y propiedades concretas que tiene que reemplazarse con otros métodos. Además, los diagramas requieren de programación de manipulación directa que debe intentar ser fluida y eficiente, para lo que es necesario conocimientos de programación más concretos.

Por otro lado, la creación de un modelo mediante el uso de listas, botones, menús y ventanas resulta más simple y mantiene la información de modelo más organizada. Sin embargo, este aspecto puede llegar a ser negativo en metamodelos grandes, con muchos elementos, porque el uso de un gran número de ventanas llega a ser caótico y se pierde aún más la visión global. La creación de un modelo puede resultar intuitiva y sencilla con una estructura ordenada de los elementos de interacción.

Con esta comparación no se pretende llegar a una conclusión sobre qué método es mejor que otro a la hora de implementar una herramienta de modelado. Se han observado puntos a favor y en contra de ambos estilos e incluso se contempla que no

son excluyentes, sino que puede complementarse para lograr una solución más completa. Se han estudiado estos ejemplos con el fin de tener referencias y experiencias de herramientas reales y, una vez identificadas las características de esta aplicación, encontrar el método adecuado en base al análisis realizado.

3. Análisis

Se ha mencionado que el objetivo de este trabajo es diseñar e implementar una aplicación para el modelado. Para ello hemos presentado un grupo de herramientas que ya existen y hemos estudiado los tipos de interacción de cada una de ellas.

Un proceso de *software* según Sommerville [28] es "un conjunto de pasos que conforman la elaboración de un producto *software"* y afirma que existen cuatro actividades que resultan necesarias para todo proceso que se desee desarrollar:

- Especificación. Consiste en determinar el software que se va a hacer, definiendo las características funcionales deseadas, las limitaciones en el proceso y cualquier aspecto para tener en cuenta para su creación.
- Desarrollo. Agrupa el detallar el diseño del producto orientado a su posterior implementación.
- Validación. Realización de diversas pruebas que confirmen la aprobación del *software* y detecten fallos.
- Evolución. Fase que termina el ciclo al modificar los errores y puede comenzar uno nuevo al actualizar los requerimientos, atendiendo a peticiones del cliente o usuario final.

La ingeniería del *software* contempla diferentes tipos de proceso de creación. En este apartado se van a estudiar diferentes modelos encontrados con el fin de comparar las características de cada uno y determinar el proceso de desarrollo más adecuado a seguir.

a. Identificación y análisis de soluciones posibles

Un modelo de proceso es una descripción representativa de un proceso que ofrece una perspectiva, por ejemplo, definir una secuencia de ejecución de las actividades de un proceso.

Tipos de modelo generales encontrados en [28]:

- Cascada. Cada una de las actividades fundamentales corresponde a una fase. Este modelo considera de gran importancia la creación de documentos al final de cada etapa que recojan todo el trabajo realizado para su posterior corrección o aprobación.

  Los pasos se suceden secuencialmente, además de una última fase final que retroalimenta todas las fases anteriores para aportar más información a la próxima iteración. Es ahí donde se testea el *software* y se detectan los errores de todas las fases.

  Es un método rígido dado que los cambios se realizan en nuevas iteraciones completas, así que es muy difícil atender a las modificaciones de requisitos en fases adelantadas. Requiere una fuerte planificación y seguimiento de las especificaciones.

- Desarrollo incremental. Consiste en ir realizando versiones de prueba del producto desde el comienzo y a cada fase de todo el proceso. De cada prototipo se hace un análisis para conseguir retroalimentación. De esta forma, se puede empezar a visualizar y probar una posible solución desde las especificaciones iniciales y el producto es más flexible a cambios durante el proceso de creación.

  Es bastante apto para proyectos personales o donde sea posible mantener contacto continuo con el cliente/usuario de forma que fácil conseguir análisis del producto. Se reduce mucho la documentación generada dado que, con un mayor número de entregas, el coste sería mucho mayor.

- Ingeniería del *software* orientada a la reutilización. Se basa en el hecho de que en el desarrollo de un proyecto se utilizan aspectos comunes de otros productos. En este modelo surgen nuevas fases más adecuadas al uso de código ya creado que a su elaboración: análisis de componentes, modificación de requerimientos según los componentes seleccionados, diseño del sistema con reutilización, desarrollo e integración. Al ser menor la cantidad de *software* que se desarrolla, se disminuyen costos y tiempos.

b. Solución propuesta

En base a las características presentadas en el apartado anterior y a las que presenta nuestro caso se ha descartado una metodología basada en un plan. Tanto la metodología incremental como la orientada a la reutilización tienen aspectos que se consideran adecuados para este tipo de trabajo, como pueden ser la agilidad de trabajo (del incremental), la utilización de elementos ya existentes (reutilización de código) y el hecho de que ambos se presentan como modelos de trabajo más rápidos que el de plan.

Sin embargo, el tamaño del *software* que se contempla desarrollar en esta aplicación no es tan amplio como para integrar un número de componentes considerable que requiera de un estudio intenso de componentes externos.

4. Diseño

En este punto de la memoria, se procede a entrar en el proceso de estudio y análisis del metamodelo que dará paso directo a la implementación de la herramienta.

Para comenzar, se describe de forma general del *framework* JaCalIVE, cuáles son sus fundamentos, sus características y sus fases de desarrollo. De estas últimas, se centra la atención en la fase de modelado, que es la que concierne a este trabajo. Se mostrará con detalle el metamodelo propuesto y de él se extraerán los requerimientos de la aplicación organizados en tablas.

Una vez indicada la funcionalidad, se decide una arquitectura para el sistema que da pie a concretar el diseño de la estructura de datos. Con la ayuda de diagramas de clase se enseñarán los elementos necesarios, sus propiedades y funciones, relacionados con los requerimientos planteados.

Por último, se justifica la decisión del lenguaje de programación y los componentes seleccionados para la implementación de la solución.

a. Especificación del Sistema

En este apartado se procede a explicar con detalle en qué consiste JaCalIVE [19] [20] [21] y de qué se compone este metamodelo. Fundamentalmente, JaCalIVE está conformado por la unión de dos metamodelos, cada uno de ellos aporta un aspecto diferente a la imagen inicial de un MAS presentada anteriormente. A continuación se introduce la definición, presentada por Rincon A., de dichos metamodelos que posteriormente servirán para entender los elementos de JaCalIVE.

El metamodelo llamado *Agents & Artifacts* (A&A) “permite distinguir dos tipos de entidades: los agentes y los artefactos. Los agentes implementan o encapsulan la inteligencia y los artefactos representan cuerpos de estos agentes. Siguiendo la misma metodología de separar la mente del cuerpo, también se encuentra un metamodelo que está basado en el metamodelo A&A y este distingue qué entidades tendrían una representación virtual (visualización en algún motor gráfico)” [19].

Multi Agent Model For Intelligent Virtual Environments (MAM5) es “un metamodelo para el diseño de IVE, y permite al desarrollador separar del IVE los agentes que tendrán una representación física. MAM5 clasifica en dos grupos las entidades que se encuentran en el IVE: el primer grupo hace referencia a las entidades que poseen un cuerpo dentro del IVE y el segundo grupo son aquellas entidades que no poseen dicha representación” [19].

En este trabajo se presenta JaCalIVE, como una herramienta que ayude a los desarrolladores a modelar, programar y mantener los IVE.

JaCalIVE (JASON CArtAgO *implemented Intelligent Virtual Environment*) es un *framework* cuyo desarrollo está dirigido a la creación de entornos virtuales inteligentes, por lo tanto, integra desde un modelo para el diseño del IVE hasta un soporte para la simulación de comportamiento de dichos entornos. Para implementar un entorno virtual inteligente, en JaCalIVE se define un método de desarrollo de IVE compuesto por tres pasos:

1. Modelado. La primera parte consiste en realizar el diseño del entorno virtual, describir las características del espacio, la distribución de las entidades que habitan en él y sus características. El resultado que se busca obtener es un modelo definido en formato XML, cuya estructura se muestra en la [Ilustración 22], para el cual JaCalIVE propone un esquema XSD de su estructura. Este es el punto sobre el que gira este trabajo, así que lo extenderemos más adelante.

2. Traducción. Una vez generado el archivo XML con el diseño, se procede a la generación automática de código esqueleto, que será la base para definir el comportamiento de las entidades. La aplicación traduce la información correspondiente del XML a fragmentos de código Java o JASON.

3. Simulación. Por último, a partir del código generado en la traducción de archivos puede comenzar la fase de ejecución del IVE. Para llevarla a cabo JacalIVE integra una serie de herramientas de soporte:

    a. JASON es una plataforma implementada en Java para gestionar la creación y actuación de agentes deliberativos que siguen la arquitectura BDI en MAS.

    b. CArtAgO (*Common ARTifact infrastructure for AGents Open environments*) proporciona un *framework* de programación y ejecución de agentes situados en un entorno virtual. Al igual que JaCalIVE, como veremos más adelante, da soporte al metamodelo A&A para el diseño de MAS y, aunque no fija ningún modelo específico para la programación de agentes, se aconseja implementar aquellos basados en la arquitectura BDI.

    c. JBullet es la herramienta utilizada para representar las leyes e interacciones físicas del entorno virtual como rozamiento o detección de colisiones.

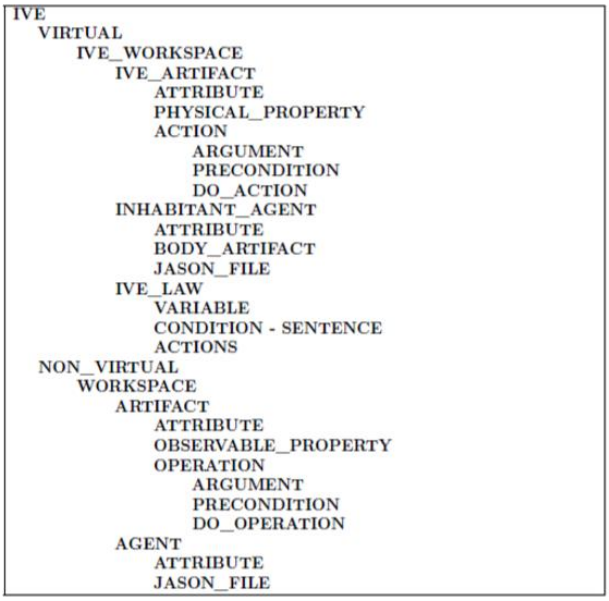

Ilustración 22. Esquema de la estructura para el XML de un modelo en JaCalIVE.

Como se ha mencionado en el punto anterior, JaCalIVE proporciona un XSD con la estructura de organización del IVE para la creación de un XML del modelo. Este XSD está basado en el metamodelo MAM5 para definir modelos de sistemas multiagente para entornos virtuales inteligentes.

Según este modelo, los elementos principales definidos en un IVE están sometidos a una división de los elementos según si tendrán una representación virtual o no, es decir, aquellos que tendrán un renderizado 3D en el entorno. A continuación, describimos los elementos principales, cuya situación dentro de un modelo puede verse en la [Ilustración 22]:

- ***Workspace***. Consiste en el espacio que agrupa un conjunto de entidades. Un ***IVEWorkspace*** es aquel que, al tener una representación virtual, poseerá propiedades físicas como el tamaño o gravedad y las entidades que habiten en él se situarán en una posición concreta. En un IVE puede haber más de un *Workspace* o *IVEWorkspace*.

- ***Artifact***. Son aquellos elementos no autónomos que se encuentran en el sistema y tiene una serie de propiedades llamadas ***Observable***. Los artifacts que tienen representación física en el entorno son los ***IVEArtifact*** y estarán definidos por un conjunto de características, estas son denominadas ***Physical***, que pueden ser de dos tipos: ***Perceivable*** son las propiedades que pueden percibirse de forma externa (por ejemplo, posición) e ***Internal***, las que solo el artefacto conoce y puede utilizar (por ejemplo, masa).

  Además, los *Artifact* son elementos que pueden realizar una serie de acciones en el entorno. Estas funciones se llaman ***Action*** y ***Operation*** en los A*rtifact* que tienen representación virtual y que no, respectivamente. Las funciones están formadas por un conjunto de argumentos, una precondición y una serie de acciones secundarias.

- ***Agent***. Representan las entidades cognitivas y autónomas que habitan en el entorno. Los agentes que no tienen representación virtual pueden realizar tareas de gestión interna del sistema e interactuar con los artefactos sin representación como, por ejemplo, bases de datos. Los agentes que sí poseen un cuerpo virtual, un conjunto de *IVEArtifact*, y habitan en un *IVEWorkspace* se denominan ***Inhabitant Agents***.

- ***IVELaws***. Definen las leyes físicas por las que se rige un *IVEWorkspace* y que, como la gravedad, afectarán también a todas las entidades presentes en él.

Se procede a empezar a comentar los requerimientos de la aplicación. Los requerimientos suelen clasificarse en funcionales y no funcionales. Los primeros corresponden a aquellos requisitos que identifican lo que el usuario espera del *software*. Los no funcionales están sujetos a consideraciones externas al sistema, por lo que no se van a tener en cuenta en este proyecto.

Las tablas que siguen a continuación son una forma estructurada de presentar los requerimientos para concretar las acciones principales que debe poder realizar un usuario para crear un modelo y trabajar con la aplicación. La plantilla de una tabla está formada por unas características que detallan los aspectos de las funcionalidades:

- Nombre: identificador de la acción.

- Descripción: explicación breve en lenguaje natural de la acción.

- Entradas: cualquier dato que necesite obtener del usuario para completar la acción correctamente.
- Salidas: cualquier archivo o ventana que la aplicación cree o muestre por pantalla.
- Precondiciones: acciones obligatoriamente previas a la actual o datos que deban ser correctos.
- Efectos: acciones consecuentes a la acción actual que el sistema realizará automáticamente.

La presentación de las tablas se ha organizado de forma que se agrupen las acciones comunes sobre diferentes elementos:

- CREAR.

Nombre	Crear proyecto.
Descripción	Establecer en la aplicación un nuevo modelo vacío.
Entradas	-Nombre o identificador del proyecto.
Salidas	-Ventana de entrada de datos
Precondiciones	-
Efectos	-

Tabla 2. Tabla de requerimiento: crear Proyecto.

Nombre	Crear espacio.
Descripción	Añadir al modelo un nuevo espacio.
Entradas	-Clase de espacio: *Workspace*, *IVEWorkspace*. -Nombre o identificador del espacio.
Salidas	-Ventana de entrada de datos
Precondiciones	-Debe crearse en un proyecto existente. -El nombre debe ser único en el modelo.
Efectos	-

Tabla 3. Tabla de requerimiento: crear espacio.

Nombre	Crear agente.
Descripción	Añadir un nuevo agente a un espacio del modelo.
Entradas	-Clase de agente: *Agent*, Inhabitant *Agent*... -Nombre o identificador del agente.
Salidas	-Ventana de entrada de datos
Precondiciones	-Debe crearse en un espacio existente. -El nombre debe ser único en el modelo. -Un *Agent* solo puede añadirse a un *Workspace*. -Todas las clases de Inhabitant *Agent* solo pueden crearse en un *IVEWorkspace*.
Efectos	-

Tabla 4. Tabla de requerimiento: crear agente.

Nombre	Crear artefacto.
Descripción	Añadir un nuevo artefacto a un espacio del modelo.
Entradas	-Clase de artefacto: *Artifact*, *IVEArtifact*... -Nombre o identificador del artefacto.
Salidas	-Ventana de entrada de datos
Precondiciones	-Debe crearse en un espacio existente. -El nombre debe ser único en el modelo. -Un *Artifact* solo puede crearse en un *Workspace*. -Un *IVEArtifact* solo puede crearse en un *IVEWorkspace*.
Efectos	-

Tabla 5. Tabla de requerimiento: crear artefacto.

Nombre	Crear ley.
Descripción	Definir una ley física en un espacio
Entradas	-Nombre o identificador de la ley.
Salidas	-Ventana de entrada de datos
Precondiciones	-Debe crearse en un espacio de clase *IVEWorkspace* existente. -El nombre debe ser único en el modelo.
Efectos	-

Tabla 6. Tabla de requerimiento: crear ley.

Nombre	Crear propiedad.
Descripción	Definir una cualidad de un artefacto.
Entradas	-Clase de propiedad: *Observable*, *Physical*, *Perceivable* o *Internal*. -Tipo de propiedad: numérico, texto, *booleano*… -Nombre o identificador de la ley.
Salidas	-Ventana de entrada de datos
Precondiciones	-Debe crearse en un artefacto existente. -El nombre debe ser único en el modelo. -Los *Artifact* solo aceptan propiedades *Observable*. -Los *IVEArtifact* solo tienen propiedades *Physical*, que pueden ser de dos subtipos: *Perceivable* e *Internal*.
Efectos	-

Tabla 7. Tabla de requerimiento: crear propiedad.

Nombre	Crear acción/operación.
Descripción	Definir una función de un artefacto.
Entradas	-Nombre o identificador de la función.
Salidas	-Ventana de entrada de datos
Precondiciones	-Debe crearse en un artefacto existente. -El nombre debe ser único en el modelo. -Las funciones en *Artifact* serán de tipo *Operation*. -Las funciones en *Artifact* serán de tipo *Action*.
Efectos	-

Tabla 8. Tabla de requerimiento: crear acción u operación.

- ELIMINAR

Nombre	Eliminar espacio.
Descripción	Borrar un espacio del sistema.
Entradas	-Identificar el espacio que se desea eliminar.
Salidas	-Ventana de confirmación.
Precondiciones	-
Efectos	-Se eliminarán los agentes y artefactos creados en ese espacio. -Desaparecerán las leyes asociadas a ese espacio.

Tabla 9. Tabla de requerimiento: eliminar espacio.

Nombre	Eliminar agente.
Descripción	Borrar un agente del sistema.
Entradas	-Identificar el agente que se desea eliminar.
Salidas	-Ventana de confirmación.
Precondiciones	-
Efectos	-

Tabla 10. Tabla de requerimiento: eliminar agente.

Nombre	Eliminar artefacto.
Descripción	Borrar un artefacto del sistema.
Entradas	-Identificar el artefacto que se desea eliminar.
Salidas	-Ventana de confirmación.
Precondiciones	-
Efectos	-Desaparecerán los valores de las propiedades añadidas -Si el artefacto es un *IVEArtifact*, desaparecerá la posible relación con un Inhabitant *Agent* asignado.

Tabla 11. Tabla de requerimiento: eliminar artefacto.

Nombre	Eliminar ley.
Descripción	Borrar una ley del sistema.
Entradas	-Identificar la ley que se desea eliminar.
Salidas	-Ventana de confirmación.
Precondiciones	-
Efectos	-

Tabla 12. Tabla de requerimiento: eliminar ley.

Nombre	Eliminar propiedad.
Descripción	Borrar una propiedad de un artefacto.
Entradas	-Identificar la propiedad que se desea eliminar.
Salidas	-Ventana de confirmación.
Precondiciones	-
Efectos	-

Tabla 13. Tabla de requerimiento: eliminar propiedad.

Nombre	Eliminar acción/operación.
Descripción	Borrar una función de un artefacto.
Entradas	-Identificar la función que se desea eliminar.
Salidas	-Ventana de confirmación.
Precondiciones	-
Efectos	-

Tabla 14. Tabla de requerimiento: eliminar acción u operación.

- GUARDAR PROYECTO

Nombre	Guardar proyecto.
Descripción	Guardar un proyecto [en disco o en aplicación].
Entradas	-
Salidas	-
Precondiciones	-Tener el proyecto que se desea guardar abierto en la aplicación
Efectos	-

Tabla 15. Tabla de requerimiento: guardar proyecto.

- ABRIR PROYECTO

Nombre	Abrir proyecto.
Descripción	Cargar en la aplicación un proyecto.
Entradas	-Nombre o fichero del proyecto / ruta de acceso al archivo en disco.
Salidas	-
Precondiciones	-Tienen que existir proyectos guardados previamente en el disco o aplicación
Efectos	-

Tabla 16. Tabla de requerimiento: abrir proyecto.

- TRADUCIR PROYECTO

Nombre	Traducir proyecto
Descripción	Crea el código XML correspondiente al modelo.
Entradas	-Ruta de salida deseada. -Nombre o identificador del archivo (se asigna nombre del proyecto por defecto).
Salidas	-Archivo XML en la zona del disco seleccionado.
Precondiciones	-Tener el proyecto que se desea traducir abierto en la aplicación.
Efectos	-

Tabla 17. Tabla de requerimiento: traducir proyecto.

b. Arquitectura del Sistema

La implementación de esta interfaz de usuario se ha realizado siguiendo el patrón de arquitectura conocido como Modelo-Vista-Controlador (MVC) que divide la estructura de en tres partes que se apoyan unas a otras. Estos componentes son:

- Modelo. Corresponde a la estructura de la información que se maneja en el sistema. Gestiona los accesos y actualizaciones de los datos.
- Vista. Representación de los datos. Muestra al usuario la información tal y como aparece en el modelo y ofrece formas de interacción para poder alterarla.
- Controlador. Responde a la interacción con la interfaz, gestionando los eventos y accediendo al modelo para modificar la información o enviarla a la vista.

c. Diseño detallado

Una vez descritos los elementos principales y la funcionalidad básica del sistema, esta fase del diseño corresponde a la transformación a objetos y métodos para la programación. En un primer boceto de estructura se ha creado una clase por cada entidad y se han establecido las relaciones. Para mejor comprensión del diagrama de clases, se han asignado colores a los elementos de forma que se aprecie la distinción entre los elementos que tienen representación virtual y los que no.

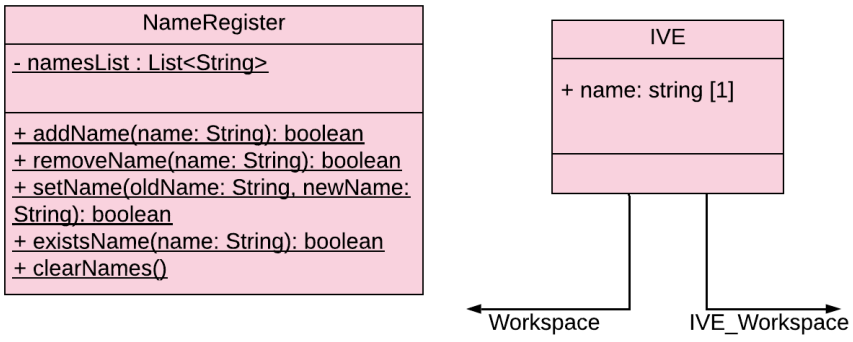

Ilustración 23. Diagrama de clases 1: clase IVE y clase registradora de nombres.

Las clases se han distribuido en cuatro colores, a continuación, se explica cada uno de estos grupos:

- En color morado [Ilustración 22] se agrupan:

- La clase IVE inicial que representa todo el entorno virtual inteligente y que está formado por dos conjuntos de entornos de trabajo, *Workspace* y *IVE_Workspace*.

- La clase *EntityRegister* que se encargan de gestionar los nombres de los elementos que tienen que ser únicos en el modelo: IVE, los espacios de trabajo, los agentes y los artefactos; los demás objetos, como las propiedades, pueden ser utilizados en diferentes elementos así que debe poderse repetir su nombre.

  Por ello mantiene una lista estática privada y los métodos estáticos para su control:

  - *AddName* añade un nombre a la lista (y devuelve *true*) si no existe ya en ella (devuelve *false*).
  - *RemoveName* elimina un nombre de la lista (devuelve *true*) si existe y si no, la lista se mantiene (devuelve *false*).
  - *SetName* actualiza un nombre (devuelve *true*) a no ser que el nuevo nombre ya exista (devuelve *false*).
  - *ExistsName* comprueba si existe un nombre (devuelve *true*) o no (devuelve *false*).
  - *ClearNames* limpia la lista entera y la deja sin elementos. Este método sirve ya si no se vacía manualmente, la lista estática se mantiene hasta que se cierre por completo la aplicación.

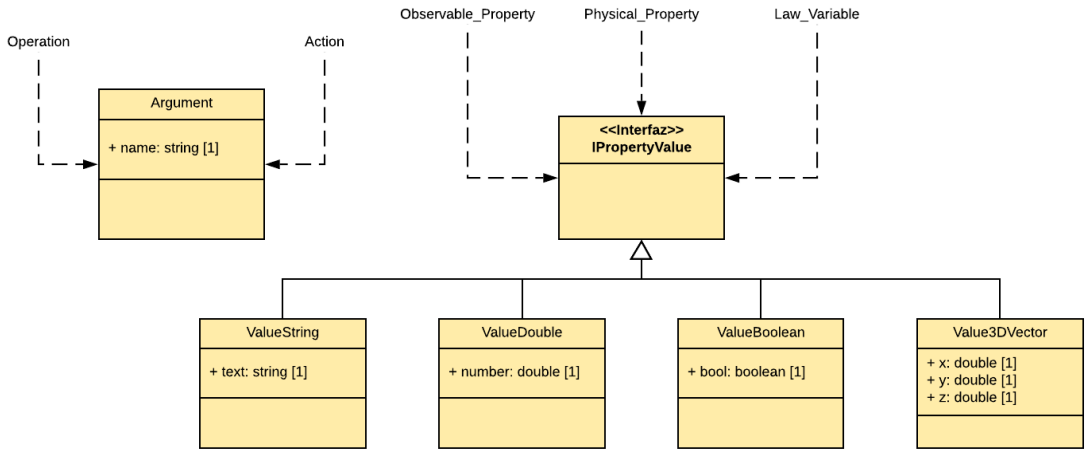

Ilustración 24. Diagrama de clases 2: clase argumento y clases tipo de valor.

- Se ha utilizado el color amarillo [Ilustración 23] para las clases que definen atributos personalizados de los elementos del sistema y pueden pertenecer a cualquiera de los dos grupos anteriores:

  - Clase *Argument*. Representa a los parámetros de las funciones *Action* y *Operation*. Al igual que cuando se define un argumento un método en programación, *Argument* requiere un nombre y un tipo.

- Interfaz *PropertyValue*. Las propiedades que se definen en una clase *Property* no tienen un tipo determinado, así que se hecho uso de las interfaces de programación para esconder a la clase propiedad todos los tipos posibles.

  Para este trabajo se han creado los tipos básicos *String*, *Double* y *Boolean*, junto a un valor complejo que se representa un vector en tres dimensiones, muy útil al trabajar en entornos virtuales. Este último ejemplo muestra cómo pueden crearse valores de propiedad más especiales según las necesidades del modelo.

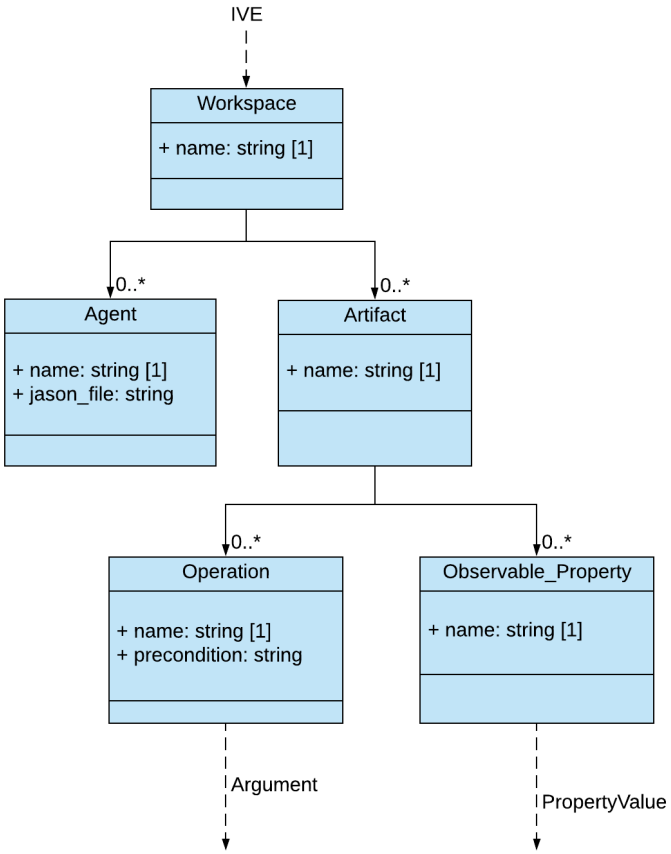

Ilustración 25. Diagrama de clases 3: clases para la parte sin representación virtual.

- El conjunto de elementos no representables virtualmente, en color azul [Ilustración 24], está formado por las clases *Workspace*, *Agent*, *Artifact*, *Observable_Property* y *Operation*:

  - *Workspace* está compuesto por conjuntos de las clases *Agent* y *Artifact*.

  - Un *Agent* guarda únicamente un atributo de texto para almacenar el nombre del archivo JASON que definirá su comportamiento en las fases siguientes.

- Un *Artifact* tiene un conjunto de *Observable_Property* y se asocia a un conjunto de *Operation*.
- Una clase *Operation* consiste en al menos un *Argument* y una posible precondición.
- La clase *Observable_Property* requiere de una implementación de la interfaz *PropertyValue* para representar el valor de la propiedad.

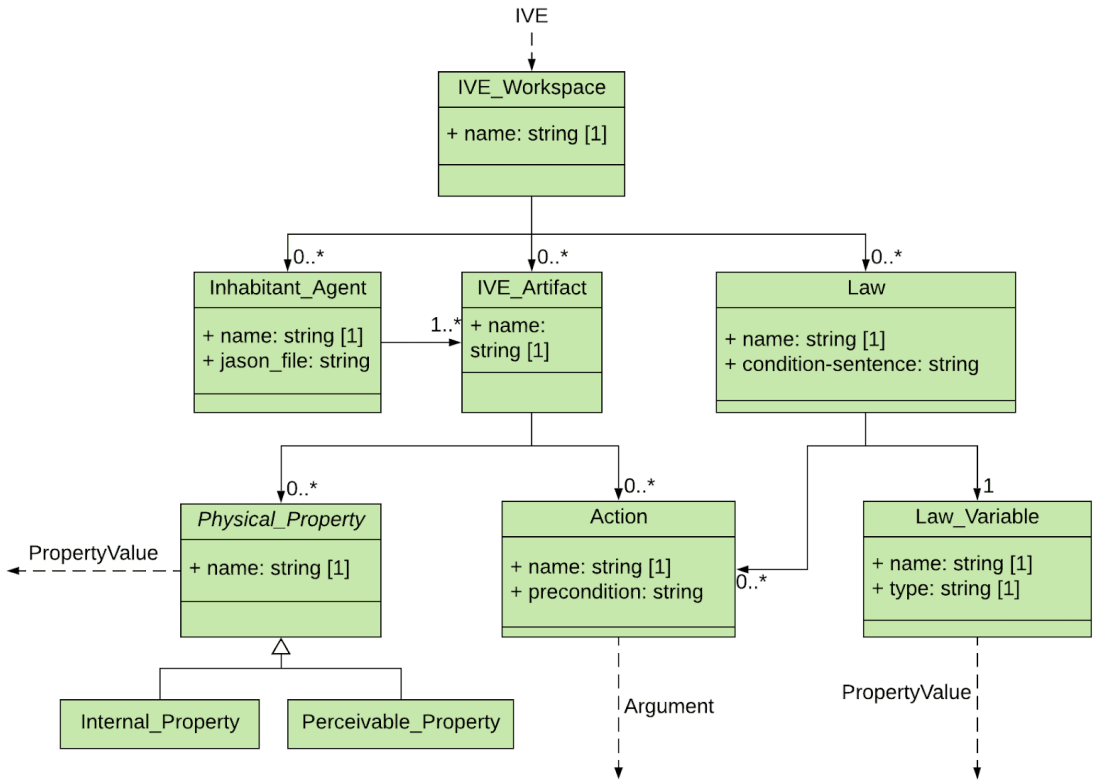

Ilustración 26. Diagrama de clases 4: clases para la parte con representación virtual.

- De forma similar, el color verde [Ilustración 25] agrupa todos los elementos que sí tienen representación virtual:
  - *IVE_Workspace* está compuesto por conjuntos de las clases *Inhabitant_Agent*, *IVE_Artifact* y *Law*.
  - Un *Inhabitant_Agent* guarda relación con un grupo de *IVE_Artifact*s que suponen su representación física y tiene también un atributo de texto para almacenar el nombre del archivo JASON.
  - Un *IVE_Artifact* tiene un conjunto de *Physical_Property* y se asocia a un conjunto de *Action*.
  - La clase *Physical_Property* es en realidad una clase abstracta pues un *IVE_Artifact* se servirá de sus subclases *Internal_Property* y

Perceivable_Property. Requiere de una implementación de la interfaz *PropertyValue* para representar el valor de la propiedad.

- Una clase *Action* consiste en al menos un *Argument* y una posible precondición.

- Para definir una clase *Law* es necesario que tenga una variable, la clase *Law_Variable* hace esa función junto con *PropertyValue*, y una serie de acciones a las que afecta, estas son clases *Action* definidas en el IVE. La clase tiene un atributo para que, de forma opcional, se pueda escribir una condición.

d. Tecnología utilizada

En este apartado final del diseño se exponen aquellas opciones que existen para la implementación del sistema propuesto y se concluye con la tecnología elegida.

Primero, en relación con el lenguaje de programación las dos grandes opciones orientadas a objetos que se han contemplado son C# y Java. Por una parte, C# es el lenguaje ampliamente utilizado en todo el *framework* de desarrollo de aplicaciones Microsoft .NET. Por otra parte, uno de los lenguajes más populares en todo el mundo y del que una de sus principales características es la independencia de plataforma, por ejemplo, a través de la tecnología JavaFX para el desarrollo multiplataforma.

Este último punto es clave para la toma de decisión, pues puede resultar bastante útil en el futuro la posibilidad del uso de la aplicación en diferentes sistemas operativos o plataformas. Teniendo en cuenta ese aspecto, se descarta la opción de usar C# en .NET pues la opción ofrecida (.NET Core) todavía no está planteada para la creación de aplicaciones de escritorio en otros sistemas operativos.

Por el contrario, como se ha mencionado, Java es notablemente utilizada en el desarrollo de *software*. De hecho, de las cinco herramientas estudiadas en el segundo apartado, cuatro de ellas (siendo la excepción CORMAS) están desarrolladas en Java y/o hacen uso de ese lenguaje para la creación de los modelos.

Habiendo decidido Java como el medio para programar la herramienta encontramos diversos entornos de desarrollo integrado (IDE) como IntelliJ IDEA, Eclipse y NetBeans. La elección de un IDE para este trabajo se considera trivial después de haber probado los tres y comprobar que el flujo de trabajo sería similar en términos de creación y programación de proyectos o compatibilidad con JavaFX y Scene Builder para implementación de interfaces.

La decisión, entonces, resulta en preferencias sobre el estilo, la usabilidad o *software* libre o comercial. En conclusión, el IDE con el que se implementará la aplicación será Eclipse Java Developer por la experiencia que ya se tiene de su uso.

5. Desarrollo de la solución propuesta

El proceso de implementación ha supuesto realizar algunos cambios en el modelo originados por la intención de simplificar la ejecución de determinadas acciones. Además de comentar los arreglos del paso de modelo a código, en esta parte se comentan los aspectos relacionados con la interfaz de la aplicación. Esto son las dificultades encontradas a la hora de enlazar la vista con el modelo y las soluciones encontradas.

El segundo aparatado explica cómo se ha llevado a cabo la transformación de los datos al archivo XML que se requería en la primera fase de JaCalIVE y que funciona como archivo de guardado.

A continuación, se presentan los cambios realizados en la aplicación con relación a su estilo y diseño y a qué son debidos.

Por último, un breve apartado finaliza el proceso de desarrollo con la forma de ejecución de la herramienta.

a. Modelo e interfaz

En este apartado se van a describir los aspectos más importantes que se han llevado a cabo en la implementación de la aplicación y aquellos inconvenientes o decisiones que han llevado a cambios en el diseño propuesto en el apartado 4.3.

La implementación ha consistido básicamente en la creación de una clase por cada cuadro del diagrama y la declaración de sus atributos, relaciones y métodos. Se han mantenido los atributos de nombre, *jason_file*, *precondition* y *condition-sentence* para sus respectivas clases. Las relaciones entre los nodos se traducen a código como propiedades del tipo al que se apunta.

En el modelo puede observarse que casi no se han escrito métodos, esto se debe a que, al ser la aplicación de modelado, solo se requiere crear los objetos y darles valores a sus atributos, como se especifica en los requerimientos. Sin embargo, sí se han creado métodos a la hora de programar, estos son los *get* y *set* de cada propiedad, más la adición y eliminación de elementos a las listas, que se han obviado en el diagrama para que quede más limpio.

Pero se han añadido nuevos tipos *Enumeration*, *PhysicalClass* y *ValueType*, ambos se utilizan para mantener una lista de las subclases de *Physical_Property* y *PropertyValue*, respectivamente. Esto ayuda a la presentación de los diferentes tipos en la interfaz y a encapsular la creación de las diferentes instancias desde la clase controlador. Mediante un patrón de diseño similar a *Factory* [11], existe una clase que contiene un método estático que recibe el *enum* seleccionado en la vista y devuelve la instancia del tipo determinado.

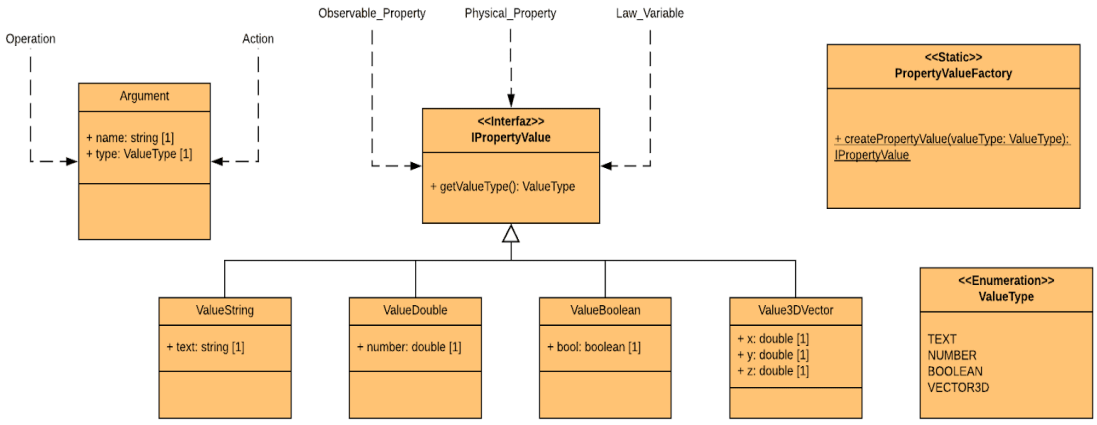

Ilustración 27. Diagrama de clases 5: cambios en tipos de valor.

En [Ilustración 26] se muestra el diagrama de los *PropertyValue* donde se han añadido dos clases. En la esquina inferior derecha, el enumerador *ValueType* con los tres diferentes valores que se han definido: *string*, *double* y *boolean*; cada una de las clases *PropertyValue* implementará el método que devuelve el *ValueType* adecuado. A la derecha de la imagen se encuentra la clase estática *ValueFactory* con el método estático que recibe el *ValueType* de la clase que se quiere instanciar.

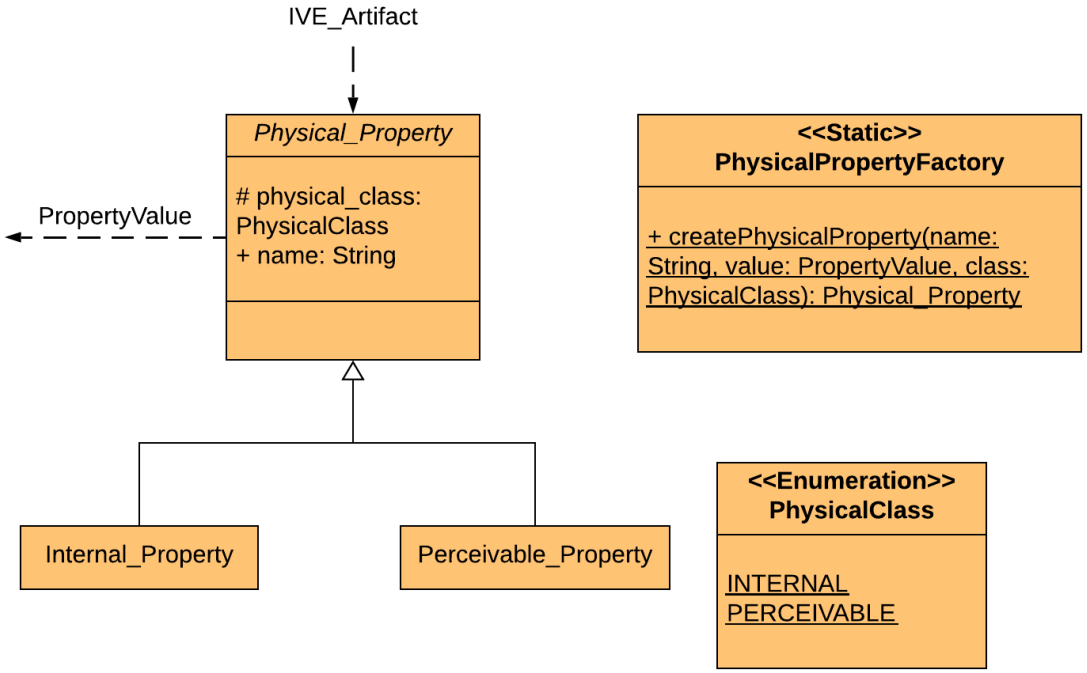

Ilustración 28. Diagrama de clases 6: cambios en las clases para las propiedades físicas.

De similar forma, en [Ilustración 27] se muestra el diagrama de los *Physical_Property* donde se han añadido dos clases, las que se encuentran a la derecha de la imagen. El

enumerador *PhysicalClass* con los dos de propiedades físicas que existen: *internal* y *perceivable*; cada una de esas clases implementará el método que devuelve el *PhysicalClass* adecuado. Encima de la clase *Enumeration* está la clase estática *PhysicalPropertyFactory* con el método estático que recibe los argumentos necesarios para instanciar un *Physical_Property* más el *PhysicalClass* de la clase correspondiente.

Estando completa la descripción del modelo, se continúa con la implementación de la interfaz, que va a incluir la parte de vista y controladores, ya que por cada vista tendremos una clase controlador mediante la cual se explicará el comportamiento. En [Ilustración 28] se introduce un esquema de navegación de ventanas de la interfaz.

Esto quiere decir qué ventanas son accesibles desde otra, indicado a través de flechas, y qué ventanas están compuestas por otras, marcado con la notación de UML de composición.

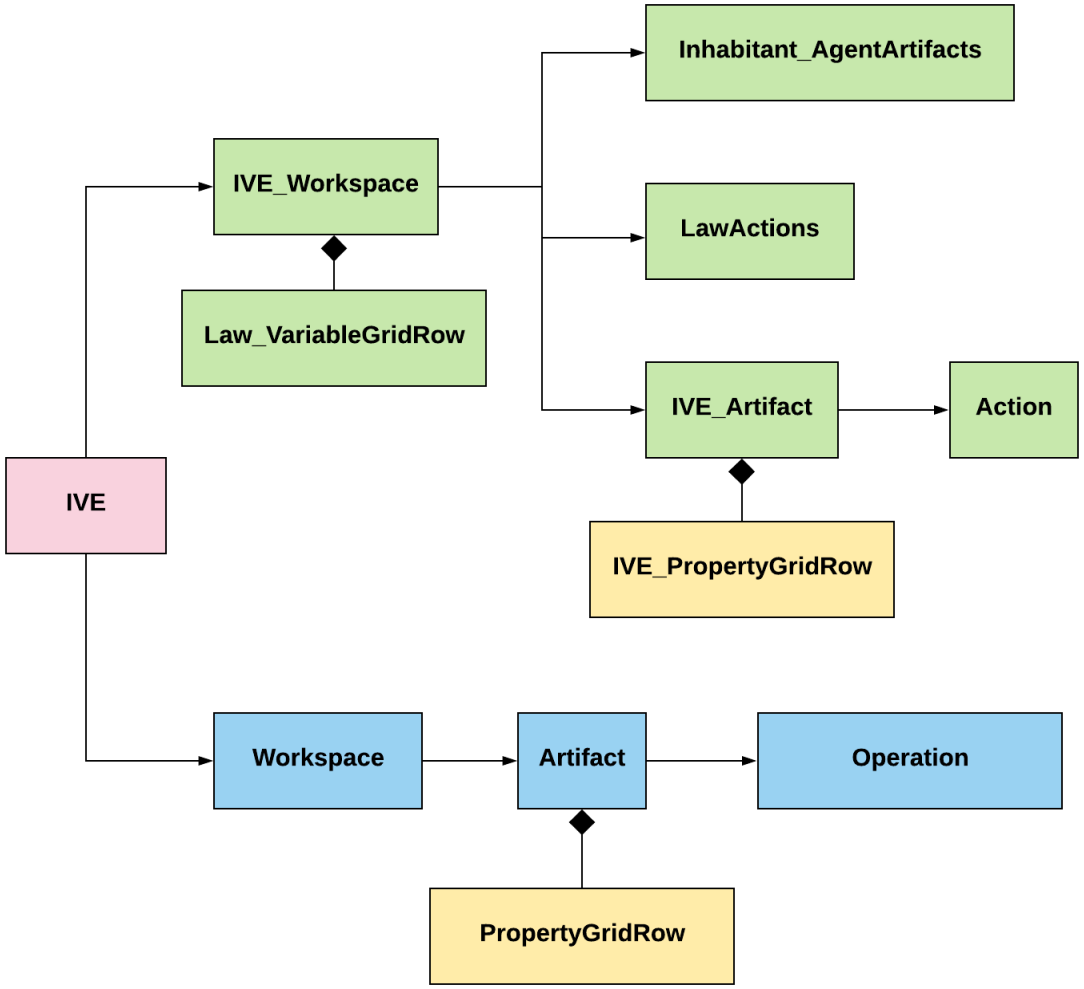

Ilustración 29. Diagrama de clases vista de la interfaz.

El orden de ventanas termina siendo similar a la jerarquía del modelo. Se accede a la edición de cada uno de los elementos a través de la ventana del elemento que los contiene, después de haber sido creados.

La [Ilustración 29] corresponde a la ventana inicial que representa un IVE, este también es el diseño base para las ventanas *IVE_Workspace*, *Workspace*, *Artifact* e *IVE_Artifact*. Consiste en dividir por apartados los objetos que se pueden crear desde elemento, proporcionando la forma de crearlos, eliminarlos y acceder a ellos. Para llevar un control general, los elementos creados se listan en una tabla que indica, además del nombre identificador, algunas de sus características, por ejemplo, el número de agentes y artefactos de un *Workspace* determinado.

Cuando se crea un objeto aparece un cuadro de diálogo simple que exigirá, como mínimo, el nombre de la instancia. Si se escribe un nombre que ya existe, sale un mensaje de error y la acción no se completa. En determinados objetos puede exigir otras propiedades, por ejemplo, seleccionar el tipo de *Physical_Property* o el tipo de *PropertyValue*.

Para poder mostrar el número de los elementos de las listas, se hace uso de un atributo privado solo modificable desde la propia clase que se enlaza a la lista correspondiente y se actualiza cuando ocurren cambios en esta.

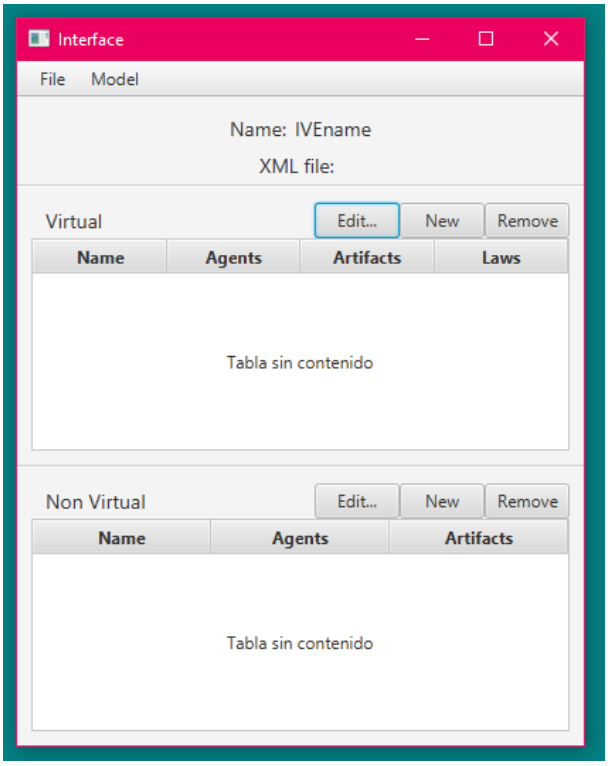

Ilustración 30. Interfaz 1: organización de ventana inicial.

Uno de los mayores problemas que se han encontrado a la hora de representar la información se ha dado en el caso de los valores de las propiedades y las variables de las leyes. Como se ha explicado en los diagramas un *PropertyValue* puede ser de diferentes tipos, y existen diferentes formas correctas de representar un valor.

Una tabla como las de [Ilustración 29] tiene el inconveniente de que cada columna solo puede tener asignado un tipo. Una solución de representación podría ser indicar un tipo texto para la columna y personalizar el formato de escritura de cada uno de los valores. Sin embargo, esto no mejora la modificación de un valor *boolean* o de otros valores complejos que puedan requerirse en un futuro ya que solo podría introducirse texto que debería especificar la forma concreta de formato.

En JavaFX se encuentran fácilmente los controles para introducir cadenas de texto (*TextField*) adecuado para *ValueString* y para marcado de sí o no (*CheckBox*) que se corresponde con *ValueBoolean*. El caso de un *ValueDouble* es más complicado pues solo existen *TextField* y *TextArea* (para introducción de texto más amplio) como forma de escribir desde teclado, por lo que será necesario un tratamiento del texto que exige comprobación de caracteres y transformación a tipos numéricos.

En consecuencia, se ha decidido realizar un tipo de tabla personalizada que se utilizará en las ventanas *IVE_Workspace*, *IVE_Artifact* y *Artifact*, que son aquellas que hacen uso de los objetos *PropertyValue*. En el diagrama se observa que estas ventanas son las que están compuestas por otras que corresponden a las instancias de las clases *Observable_Property*, *Physical_Property* y *Law_Variable*. La terminación *GridRow* al final de dichas ventanas indica que cada una de estas será utilizada como fila de la tabla.

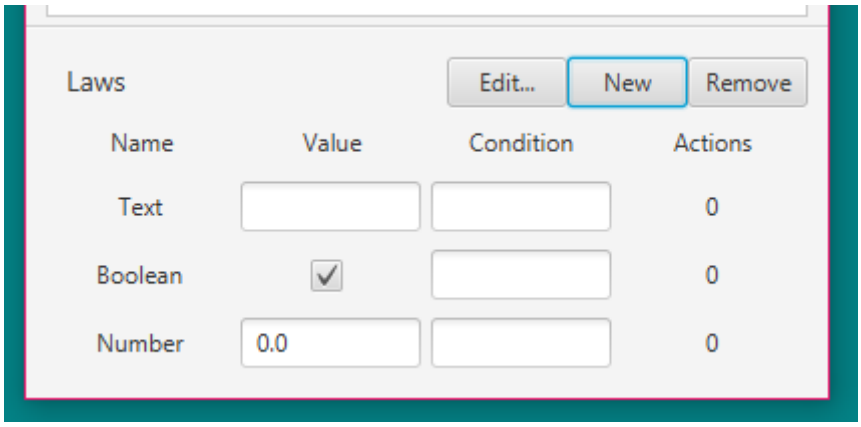

Ilustración 31. Interfaz 2: organización de tablas personalizadas.

Una tabla personalizada es básicamente un contenedor de JavaFX que dispone los elementos que se van añadiendo de forma vertical (*VBox*). Habrá un contenedor de tipo cuadrícula (*Grid*) fijo que servirá de encabezado de la tabla, con las columnas requeridas para representar los atributos y las etiquetas con el nombre de dichos atributos.

Por lo tanto, una ventana *GridRow* consistirá también en un *Grid* similar al encabezado, formado por una fila y con el mismo número de columnas que el encabezado. En las celdas correspondientes se introducirán los controles que requeridos para representar los valores.

Por ejemplo, en [Ilustración 30] se muestra la tabla personalizada de un objeto *Law*. Recordando el apartado de diseño, una instancia de este tipo tiene los siguientes atributos: *name* (*string*), *value* (*PropertyValue*), *condition-sentence* (*string*), *actions* (lista de *Action*). Tanto el nombre como la condición son de tipo texto, así que se les adjudica un *TextBox* para mostrar y editar el valor. Las acciones, al igual que en las tablas anteriores, se mostrarán como el número de elementos que son para esa instancia, y se resuelve con una etiqueta que muestre el valor.

La columna del *PropertyValue*, por la que se debe este método, podrá tener diferentes controles según el tipo que se haya creado. Esto se consigue gracias al controlador de la ventana, que en el momento de crear una instancia del *GridRow*, hace visible el control indicado según el tipo de *PropertyValue*.

Otro tema que se trata de una forma especial es el de *Inhabitant_Agent* y *Law* en el caso de su relación con *IVE_Artifact* y *Action*, respectivamente. Esta relación debe suceder hacia instancias de estas clases ya existentes, es decir, desde un *Inhabitant_Agent* no se crea un *IVE_Artifact,* sino que se seleccionan de los ya creados en el modelo y ocurre los mismo con *Law* y *Action*.

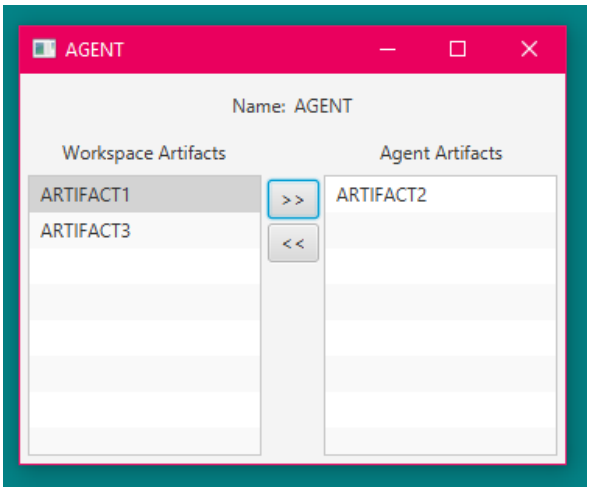

Ilustración 32. Interfaz 3: ventana de adición de artefactos a un agente.

La solución a este caso no es muy compleja, en [Ilustración 31] se muestra la ventana que aparece cuando se acciona el botón de añadir un cuerpo al *Inhabitant_Agent*. Su diseño cuenta principalmente con dos listas: la de la derecha contiene los *IVE_Artifact* que ya han sido añadidos al cuerpo del agente y la lista de la izquierda muestra todos los *IVE_Artifact* que se han creado en el *IVE_Workspace*, a excepción de los añadidos a la lista del agente. El paso de artefactos a través de las listas se realiza mediante los botones que se encuentran en el centro de la ventana.

El diseño de la ventana de adición de acciones a una ley es idéntico a esta. La única diferencia es de carácter interno, que sucede en la clase *IVE_Workspace* del modelo. Esta clase tiene por definición una lista de *IVE_Artifact*, que además es visible en la ventana de edición de un *IVE_Workspace*.

Sin embargo, no se había especificado en el diseño una lista de acciones. Para poder ofrecer una lista de acciones correspondiente a un *Law*, se ha creado un método en *IVE_Workspace* que recorre todos los *IVE_Artifact* y recoge todos sus *Action*. Este método se llama nada más se abre la ventana.

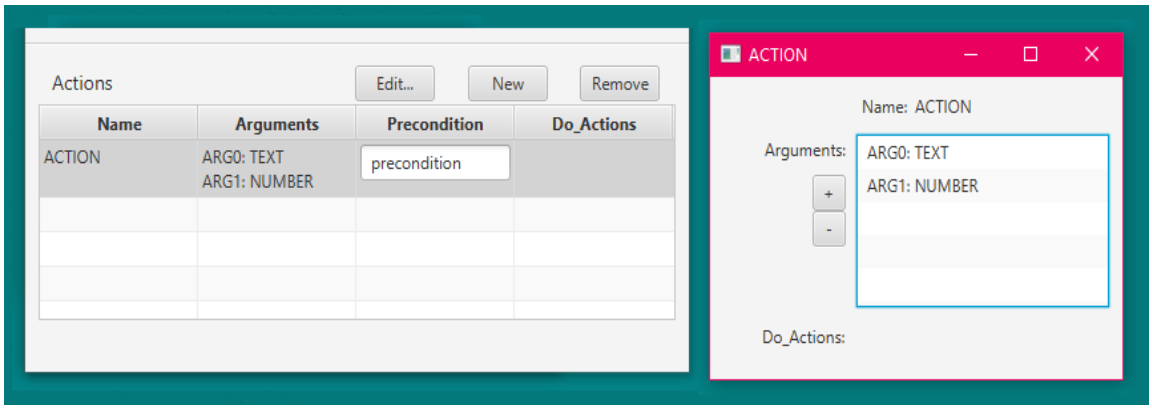

Ilustración 33. Interaz 5 (de izquierda a derecha): Panel de acciones en un IVE_Workspace y ventana de adición de argumentos a una acción.

Para acabar, se explica el caso de los argumentos en *Action* y *Operation*. [Ilustración 32] es la ventana de edición de los argumentos de un *Action* donde se encuentra la lista editable de argumentos. Para crear un argumento solo se necesita especificar su nombre y su tipo y se representan como una cadena con el formato UML para los argumentos.

A la derecha, [Ilustración 32] muestra la tabla de acciones de un *IVE_Workspace* donde se ha decidido cambiar el formato de representación de listas numérica que se veía en [Ilustración 29] por una cadena que junta todos los atributos separados por saltos de línea.

Se ha decidido usar este formato para los argumentos ya que se espera que estos no alcancen un número alto. Si este método no resulta en una forma de complicar la tabla y, además, se considera que mejora la visualización general de los objetos, podría trasladarse el cambio a los demás casos y mostrar una cadena con los nombres.

b. Traducción

La traducción del modelo a un archivo XML se ha llevado a cabo usando el *framework Java Architecture for XML Binding* (JAXB). Su uso consiste en añadir anotaciones en el propio código de las clases del modelo que indiquen qué objetos y propiedades se añaden y cómo. A continuación, se citan las anotaciones usadas en el proyecto y cuál es su función:

- *XmlRootElement* tiene que colocarse con la definición de la clase para que esta pueda usarse en la traducción. Permite personalizar el nombre que aparece en la etiqueta.

- *XmlAccessorType* se utiliza para definir la adición de los campos del objeto. Si no se indica, todos los campos que tengan ambos métodos *get* y *set*, a menos que se indique lo contrario. En el proyecto se ha especificado lo contrario, no añade ningún campo que no se haya indicado con anotaciones.

- *XmlAttribute* se coloca junto a los campos que se quieren añadir en la misma definición XML del objeto.

- *XmlElement* se coloca junto a los campos que quieren traducirse como elementos etiquetados dentro del objeto al que pertenecen.

- *XmlElements* es un tipo de anotación que contiene un conjunto de *XmlElement* diferentes. Se utiliza con campos que definen colecciones para indicar los diferentes tipos que puede contener la colección. Por ejemplo, en la aplicación se ha utilizado con los diferentes tipos de valores que puede tener una propiedad.

- *XmlID* es una anotación que define un campo cuyo valor de instancia tiene que ser único en todo el documento. Esta anotación es compatible con las anotaciones *XmlAttribute* y *XmlElement*.

- *XmlIDRef* define un elemento etiquetado que señala a un objeto que contiene un *XmlID* y solo muestra esa propiedad del objeto. Resulta muy útil para no repetir objetos enteros que se encuentran como propiedad de varios objetos. Por ejemplo, en la aplicación se ha utilizado con los *IVE_Artifact* a los que puede estar asociado un *Inhabitant_Agent*.

- *XmlTransient* se coloca en la definición de una clase que se desea que sea ignorada en la traducción. Por ejemplo, clases abstractas o interfaces.

c. Estilo de la interfaz

En el apartado 4.c se han ido mostrando capturas de pantalla de las ventanas de la interfaz cuya implementación responde a la propuesta de diseño del apartado y a las exigencias del modelo. En esta parte se van a contar los cambios que han ocurrido en la interfaz en cuestiones de estilo con el fin de hacer la aplicación más atractiva visualmente.

Uno de los inconvenientes mencionados en el apartado 5.a, donde se enfrentaban la manipulación directa y el uso de ventanas, es que puede resultar confuso un gran número de ventanas. Por ello, aunque cada ventana indique el nombre del elemento que se están editando, se ha decidido al usuario a situarse mejor en el IVE modificando los colores de las ventanas y controles.

Partiendo de esta idea, se ha asociado a la parte virtual de un IVE el color verde y a los elementos no representables virtualmente, el color azul; compartiendo la elección de colores mostrada en los diagramas del modelo. A diferencia del diagrama, las propiedades no tienen ningún color asociado, sino que, al encontrarse en las ventanas de las entidades, adoptarán el color correspondiente.

Finalmente, se decidió personalizar también los cuadros de diálogo de introducción de datos, confirmación y error. Cada uno tiene un color diferente asignado: amarillo, rojo y negro, respectivamente [Ilustración 33].

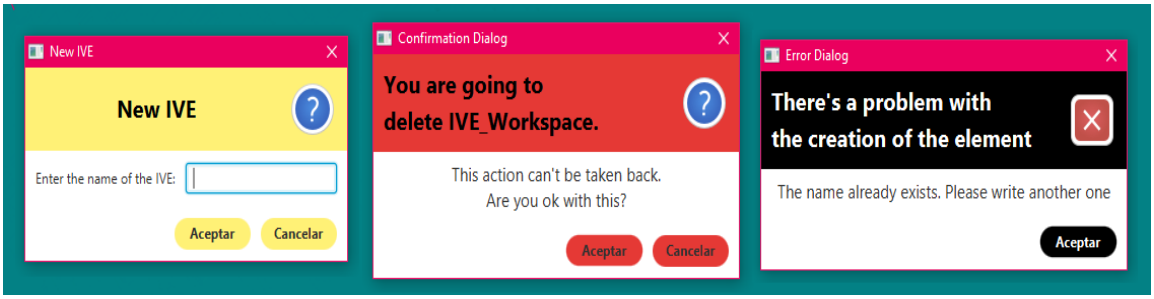

Ilustración 34. Interfaz 6 (de izquierda a derecha): diálogo de introducción de nombre, diálogo de confirmación de eliminación y diálogo de error de coincidencia de nombre.

Debido a este repaso de estilo, la interfaz también ha sufrido cambios con respecto a la organización de las ventanas:

- La ventana inicial separa la parte virtual y no virtual en dos pestañas. En principio, se realizó por cuestiones estéticas, pero finalmente se considera que la reducción del tamaño de la ventana puede ser una mejora. Comentar también en esta ventana un aspecto de usabilidad de deshabilitar los elementos cuando no se ha creado o abierto un modelo.

- Añadir un *ScrollPane* a las ventanas de *Workspaces* y *Artifacts* para poder reducir su tamaño.

d. Implantación

Tal y como ha quedado indicado en el apartado 5.c, la aplicación se ha desarrollado en el lenguaje de programación Java con la intención que pudiera ejecutarse en múltiples sistemas operativos. La solución más adecuada y fácil es crear un archivo JAR ejecutable.

El único requisito necesario para poder abrir un programa Java en un ordenador es tener instalada la máquina virtual de Java. Si todo está configurado correctamente y se ha definido que el archivo utilice la plataforma de ejecución de Java, un simple doble clic sobre el ejecutable abrirá la aplicación.

6. Pruebas

Para mostrar el resultado de la solución, van a proponerse dos casos de prueba que se explicarán, introducirán en la aplicación y traducirán a XML. De cada uno se listarán los elementos que componen el modelo junto con sus características y capturas de pantalla que muestre la introducción de los datos en el modelo.

a. Juego de pelota

La primera prueba consiste en un entorno de juego tipo fútbol, donde dos agentes jugadores se moverán por el campo con la intención de dirigir un balón hacia la portería del equipo contrario. En otros términos, el modelo consiste en un *IVE_Workspace* Campo, dos *Inhabitant_Agent*, con sus respectivos *IVE_Artifact*, otro que corresponde a la pelota y dos últimos *IVE_Artifact* portería. A continuación, se muestran mediante imágenes de la herramienta las propiedades concretas de cada uno de ellos. En el anexo A.I se encuentra el archivo XML resultado de esta prueba.

- El modelo está formado únicamente por un *IVE_Workspace*, que representa el campo de juego [Ilustración 34]. En la tabla se indica el número de agentes, artefactos y leyes presentes en el *IVE_Workspace.*

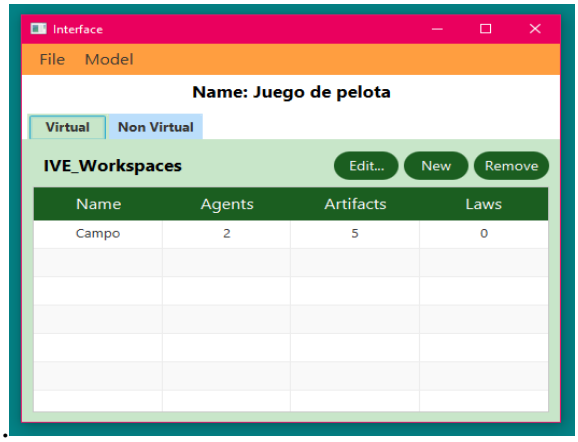

Ilustración 35. Juego de pelota

- El campo de juego [Ilustración 35] contiene dos agentes jugadores y cinco artefactos: dos porterías, una pelota y los que representarán virtualmente a los agentes. La tabla de agentes indica los archivos donde se implementará su comportamiento: *jugador1.jason* y *jugador2.jason*; también, el número de artefactos que componen su representación virtual, con detalle se muestra en la [Ilustración 36] cuáles son esos artefactos.

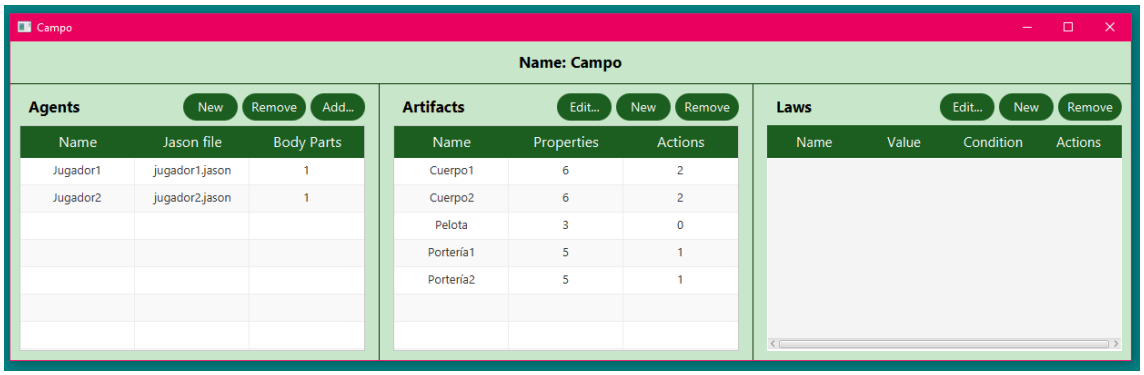

Ilustración 36. Juego de pelota: campo.

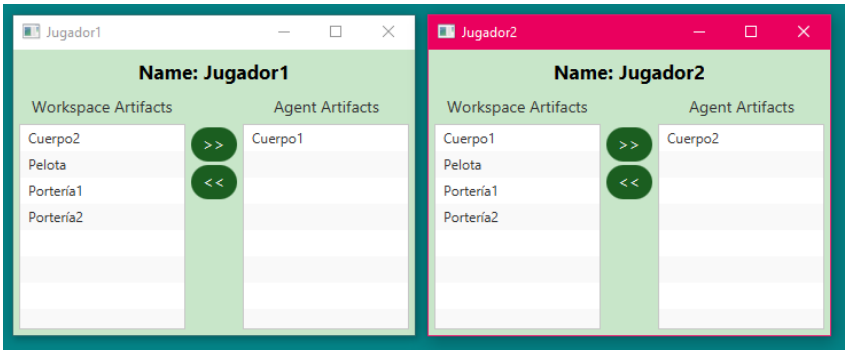

Ilustración 37. Juego de pelota: agentes jugadores.

- El artefacto Pelota [Ilustración 38] tiene las propiedades físicas color (blanco, no pertenece a ningún equipo), un diámetro, de tipo numérico, y una posición de tipo vector 3D. La posición inicial de la pelota es (0, 0, 0), marcando el centro del campo[27].

  El objeto se mueve por efecto de las acciones de los agentes y no por acciones propias, por lo que no se le aplican los métodos como a los artefactos anteriores. Tampoco necesitan propiedades de control para el movimiento como la velocidad.

[27] Para clarificar la situación del campo: los ejes x e y corresponderían a los márgenes laterales y el eje z (sin valores en este caso) representa la altura.

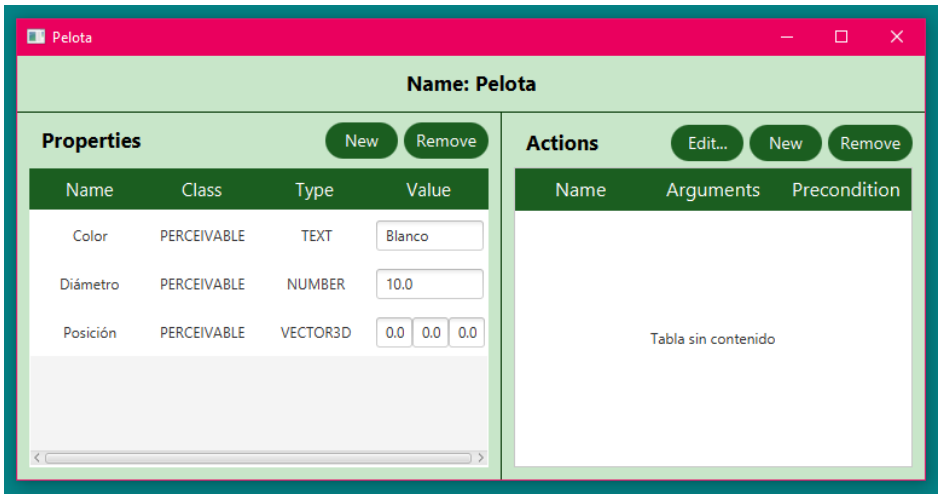

Ilustración 38. Juego de pelota: artefacto Pelota.

- Los artefactos para los agentes [Ilustración 37] tienen seis propiedades: el color que identifica el equipo indicado mediante texto; la posición y la velocidad de movimiento en el campo se representan mediante un vector para indicar los ejes x, y, z; y valores de tipo numérico para las propiedades alto, ancho y largo.

  Los valores iniciales de velocidad cero y de tamaño es igual para ambos artefactos. El color y la posición cambia según el equipo: el agente de la [Ilustración 38], del equipo azul, comienza en la posición -45 en el eje y; mientras que el agente del equipo rojo comenzará en la posición 45 del eje y.

  En cuanto a las acciones, los agentes pueden moverse en una dirección determinada, indicada con un vector 3D, con una aceleración. También puede rotarse un agente, indicando el eje de giro.

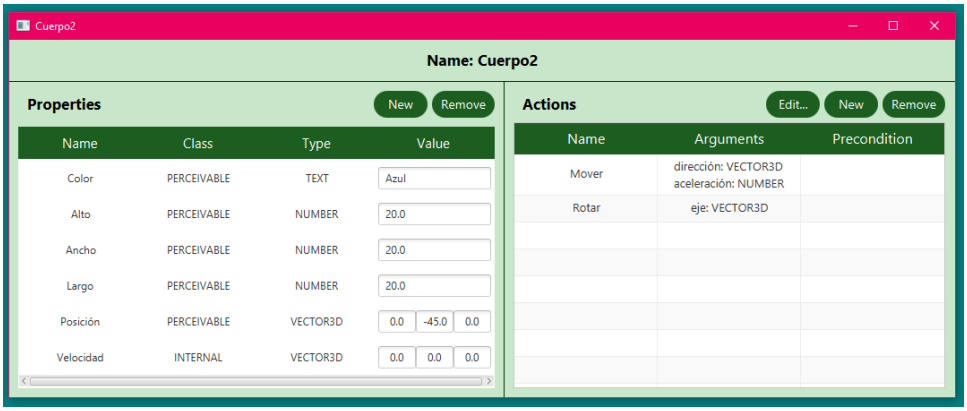

Ilustración 39. Juego de pelota: artefacto cuerpo.

- Los artefactos para las porterías [Ilustración 39] permanecen fijos en el campo así que solo necesitan color, altura, anchura, largo y posición. De igual forma que con los artefactos jugadores, las propiedades correspondientes a la forma del artefacto son idénticas para ambos y tanto el color como la posición cambian según el equipo.

  El artefacto de la [Ilustración 40], del equipo azul, comienza en la posición -65 en el eje y; mientras que el agente del equipo rojo comenzará en la posición 65 del eje y.

  Tienen una acción en la que detectan si la pelota ha entrado en la portería y anuncian gol.

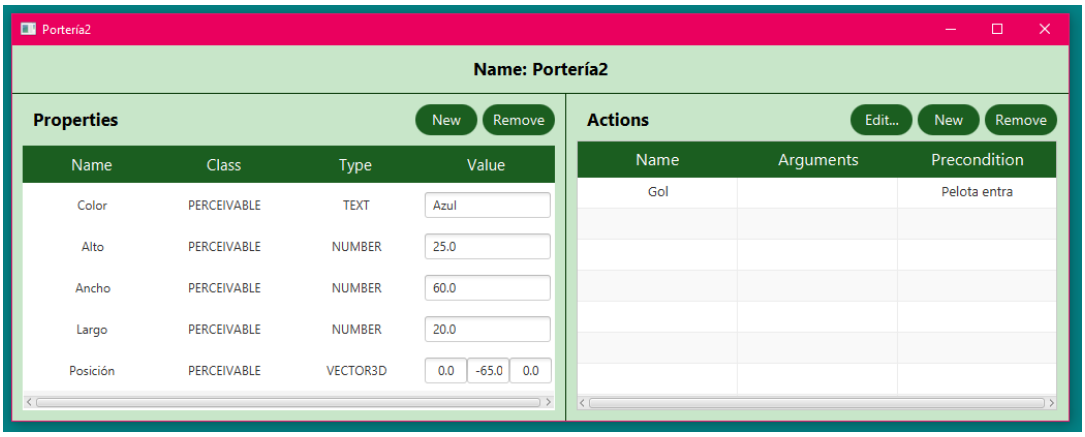

Ilustración 40. Juego de pelota: artefacto Portería.

b. Avión

Para la segunda prueba se ha querido mostrar un ejemplo formado por dos *Workspace*. El entorno de este caso simula un avión volando por un entorno controlado por un agente.

En el W*orkspace* Mundo representable virtualmente se tendrá un único *Inhabitant_Agent* Avión, esta vez su cuerpo estará formado por tres A*rtifact* (el cuerpo del avión y dos propulsores) que corresponden a diferentes partes del avión. Además, el *workspace* incluirá dos *IVE_Law* que representan la fuerza de la gravedad y de vientos del entorno.

El *Workspace* que no es representado virtualmente ControlMundo, relacionado con Mundo, se encarga de la inteligencia artificial detrás del comportamiento cambiante de los vientos mediante un *Agent* ControlVientos. En el anexo A.II se encuentra el archivo XML resultado de esta prueba.

- En este caso el IVE está compuesto por dos *Workspaces*, uno con representación virtual [Ilustración 40] y otro sin ella [Ilustración 41].

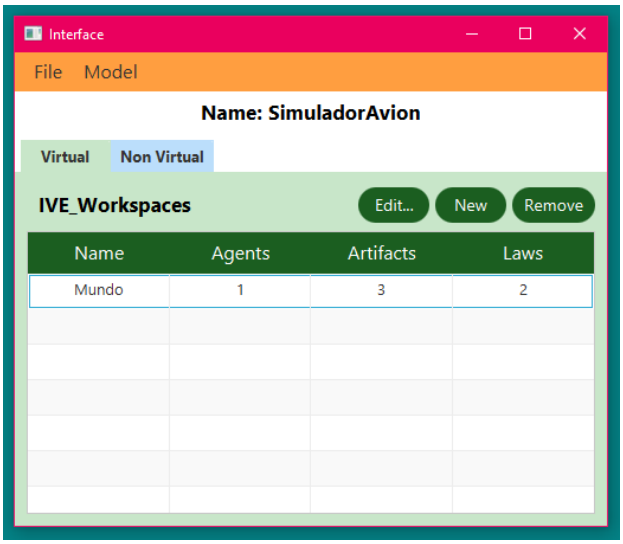

Ilustración 41. Simulador de avión: mundo virtual.

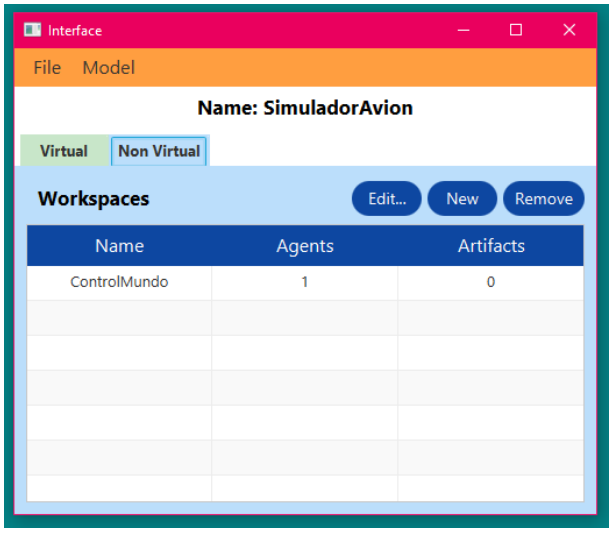

Ilustración 42. Simulador: mundo no virtual.

- Empezando por el mundo virtual [Ilustración 42], este contiene los artefactos que controlan el avión (el cuerpo del avión y los propulsores [Ilustración 43]) y el agente que controla el avión en todo su conjunto, cuyo comportamiento se implementará en el archivo *avion.jason*. Sobre el avión actuarán la fuerza de la gravedad y el viento del entorno, que se representan mediante vectores para indicar los ejes x, y, z.

Ilustración 43. Simulador de avión: mundo.

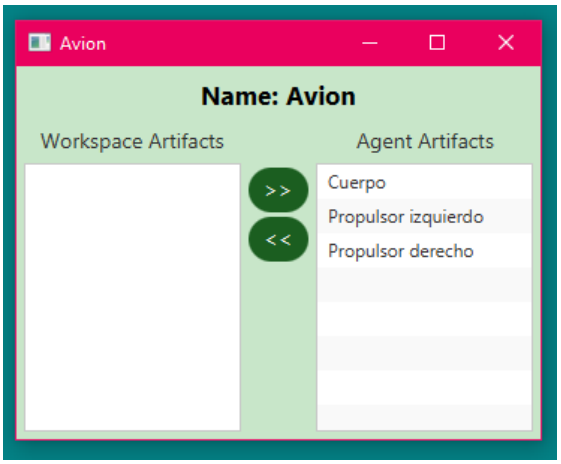

Ilustración 44. Simulador de avión: agente avión.

- El cuerpo del avión [Ilustración 44] únicamente nos indica la masa del objeto, que es un tipo numérico, y la orientación del objeto como un vector 3D. Sobre el artefacto puede realizarse la acción Girar, que requiere un vector 3D que indique el eje de giro y su velocidad.

  Cada propulsor (izquierdo y derecho) indica su potencia actual, cuyo valor inicial es cero, y permite modificar esta propiedad a través de la acción acelerar, que acepta un parámetro numérico.

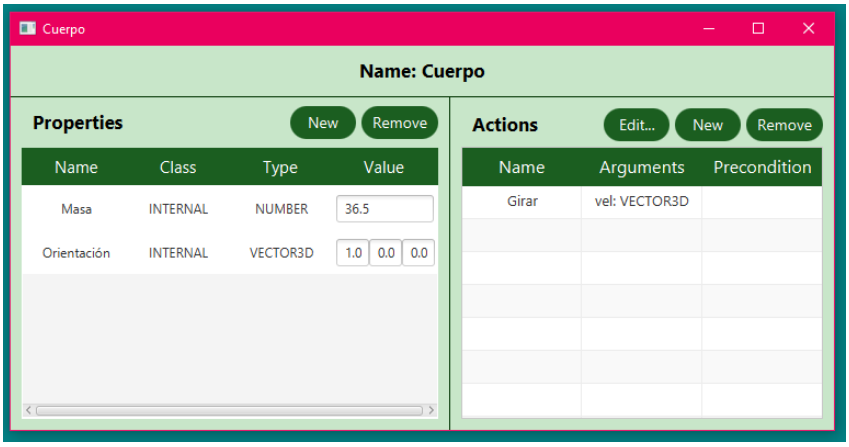

Ilustración 45. Simulador de avión: artefacto cuerpo.

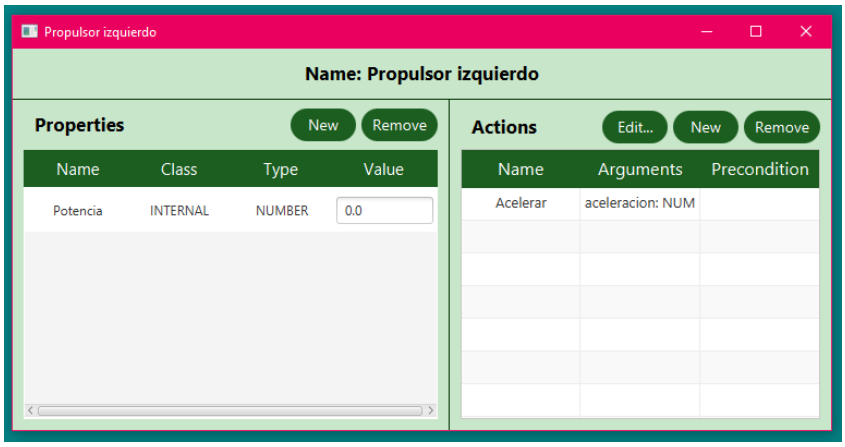

Ilustración 46. Simulador de avión: artefacto propulsor.

- Por otro lado, el mundo virtual [Ilustración 46] únicamente dispone de un agente (que no tiene representación virtual) que se encargará de controlar el viento que afecta al avión modificando los valores de los IVE_Law, ese comportamiento se implementará en el archivo *vientos.jason*.

Ilustración 47. Simulador de avión: control del mundo.

7. Conclusiones

Se ha comprendido la importancia de identificar un marco teórico previo a la realización de un trabajo, pues este ayuda a la situación en un contexto y comprender, por ejemplo, qué motivaciones existen detrás del área donde se desarrolla la idea. También, en qué estado se encuentran otros trabajos similares y por qué.

En un principio, resultó complicado entrar en contacto con elementos nuevos que no se habían tratado durante la carrera, como el mundo específicamente de los IVE. Aunque algunas asignaturas de la rama de computación introducen y estudian los sistemas multiagente y computación gráfica, resultó difícil encontrar explicaciones sobre los IVE.

La mayoría de los documentos encontrados sobre esta disciplina generalmente se centraban en uno solo de los ámbitos que lo forman (realidad virtual e inteligencia artificial). Los trabajos de Aylett junto con Cavazza [1] y Luck [2] consiguieron ser la base para comenzar con el mundo de los IVE.

A partir de este marco teórico, se dio el paso a centrarse en los MAS con el fin reunir un conjunto de herramientas de modelado actuales y encontrar varios estilos de interfaces, buscar sus características similares y diferentes y analizarlas de formas que ayude en la toma de decisiones del diseño de este trabajo. Se llegó a la conclusión de que un método de modelado mediante manipulación directa suele incluir el uso de ventanas y menús para poder abarcar por completo los elementos del metamodelo, por lo que este proyecto se implementaría directamente pensando en un sistema de ventanas y listas, determinado también por la menor complejidad de esta solución.

A continuación, se ha mostrado la explicación del *framework* JaCalIVE de forma que se comprenda en qué consiste y cuál es su estructura a nivel general. Se ha puesto especial atención en estudiar la parte del modelo que será el tema que se desarrollará a lo largo del trabajo.

El hecho de tener que realizar una aplicación específica JaCalIVE ha requerido de un esfuerzo por comprender lo máximo posible de sus características (clasificación de agentes, artefactos y sus propiedades) encontradas en los trabajos de Rincon [19] (et al. [20] [21]). De todas ellas, tener que determinar cuáles son los requisitos necesarios para diseñar una interfaz funcional.

Se han identificado los elementos del metamodelo y determinado las acciones que hacen falta para construir un modelo y se han mostrado en forma de requerimientos básicos. Consiste básicamente en relacionar los elementos para permitir crearlos en la jerarquía adecuada.

A través de diagramas de clases UML se muestra la jerarquía de clases derivada de la identificación de los elementos. El paso de las especificaciones al código del modelo y, posteriormente, las exigencias de la interfaz llevan a recurrir a clases de apoyo como interfaces para agrupar ciertos elementos o a patrones de diseño. Todas estas complicaciones y las decisiones consecuentes se han explicado junto a los diagramas.

Finalmente, se han mostrado unas pruebas que presentan posibles modelos para un IVE introducidos en la aplicación, mediante capturas de la interfaz se observa la herramienta final y cómo se visualizan los elementos del modelo. Los resultados que se requerían, código XML, se han adjuntado en los anexos al final de esta memoria.

8. Trabajos futuros

En este apartado final, se van a dejar listados algunos de los aspectos que se considera que quedan pendientes pensando en el futuro de la aplicación.

- Estudio de organización de ventanas para determinar la distribución óptima de los elementos.

- Completar el modelo añadiendo los elementos atributos, *do_operation* y *do_action*. El primero de estos serían muy similar a las propiedades de los artefactos. El segundo probablemente requerirá definir una clase que se relacione con los argumentos definidos para la acción u operación y que pueda relacionar estos también con las propiedades del artefacto.

- Añadir método de creación de modelos mediante manipulación directa. Este método servirá para crear de forma visual, similar a esquemas o a UML.

- Crear una base de datos integrada en la aplicación que guarde los proyectos y se permita cargarlos sin necesidad de buscar en el ordenador.

- Visualización del texto XML para poder observar los cambios conforme estos se realizan.

9. Bibliografía

[1] AYLETT R., CAVAZZA M. *Intelligent Virtual Environments - A State-of-the-art Report.* University de Salford, CVE, Salford; University of Teesside, School of Computing and Mathematics, Middlesbrough; Reino Unido. 2001.

[2] AYLETT R., LUCK M. *Intelligent Virtual Environment.* Universidad de Salford, CVE, Salford; University of Warwick, Department of Computer Science, Coventry; Reino Unido. 2000.

[3] ALARCÓN, J.M. *Plataforma .NET, Plataforma .NET Core y Xamarin: el panorama de las tecnologías de desarrollo Microsoft en 2018*. Disponible online: https://www.campusmvp.es/recursos/post/plataforma-net-plataforma-net-core-y-xamarin-el-panorama-de-las-tecnologias-de-desarrollo-microsoft-en-2018.aspx. Publicación: 26 de septiembre de 2017. Última consulta: 19/06/2019.

[4] BOMMEL, P., BECU, N., LE PAGE, C., BOUSQUET, F. Cormas, *an Agent-Based Simulation Platform for Coupling Human Decisions with Computerized Dynamics*. En: Kaneda T., Kanegae H., Toyoda Y., Rizzi P. (eds) *Simulation and Gaming in the Network Society* (págs. 387-410). Translational Systems Sciences, vol 9. Springer, Singapore. ISBN: 9789811005749.

[5] BOUVIER, P., DE SORBIER, F., CHAUDEYRAC, P., BIRI, V. *Cross Benefits Between Virtual Reality And Games*. 2008. En: Conference: *International Conference on Computer Games, Multimedia and Allied Technology* (CGAT'08).

[6] COLLIS, J., NDUMU, D. *The Zeus Agent Building Toolkit - ZEUS Technical Manual.* Intelligent Systems Research Group, BT Labs. 1999.

[7] COLLIS, J., NDUMU, D. *The Zeus Agent Building Toolkit - The Runtime Guide.* Intelligent Systems Research Group, BT Labs. 1999.

[8] COLLIS, J., NDUMU, D. *The Zeus Agent Building Toolkit - The Application Realisation Guide.* Intelligent Systems Research Group, BT Labs. 1999.

[9] DICKINSON, P., GERLING, K., HICKS, K. et al. *Virtual reality crowd simulation: effects of agent density on user experience and behaviour*. 2018. En: *Virtual Reality* (2019), Volume 23, Issue 1 (págs. 19–32).

[10] FOWLER, M. *UML distilled: a brief guide to the standard object modeling language*, 3ª edición. Massachusetts: Addison-Wesley, 2003. ISBN 0321193687.

[11] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995. ISBN:0-201-63361-2.

[12] GÓMEZ SANZ, J. J. *Modelado de sistemas multiagente.* Memoria que presenta para optar al grado de Doctor. Departamento de Sistemas Informáticos y Programación, Facultad de Informática, Universidad Complutense de Madrid. Junio, 2002.

[13] KISHORI, S. *Learn JavaFX 8: Building User Experience and Interfaces with Java 8.* Berkeley, CA: Apress L. P., 2015. ISBN: 9781484211427. [En línea]. Disponible: https://www.safaribooksonline.com/library/view/learn-javafx-8/9781484211427/

[14] KUTZKE, K. *Exploring emergent AI in video games*. [En línea] Disponible: https://maherou.github.io/files/CS373/SamplePapers/KutzkeKatieAssignment5.pdf Última consulta: 21/08/2019.

[15] LAUKKANEN, S., KARANTA, I., KOTOVIRTA, V., MARKKANEN, J., RÖNKKÖ, J. *Adding Intelligence to Virtual Reality*. Conferencia: Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain (2004). Adding Intelligence to Virtual Reality. [En línea] Disponible: https://www.researchgate.net/publication/220837605_Adding_Intelligence_to_Virtual_Reality

[16] LEFFINGWELL, D., WIDRIG, D. *Managing software requirements: a use case approach*, 2ª edición. Boston: Addison-Wesley, 2003. Capítulo 14. ISBN: 9780321122476. [En línea]. Disponible: https://www.safaribooksonline.com/library/view/managing-*software*-requirements/032112247X/ch16.html

[17] NORTH, M.J., COLLIER, N.T., OZIK, J. et al. *Complex adaptive systems modeling with Repast Simphony*. 2013. [En línea] Disponible: https://doi.org/10.1186/2194-3206-1-3

[18] QUESADA, F. *Introducción al lenguaje NetLogo y la programación basada en agentes.* 2018. [En línea] Disponible: http://franciscoquesada.com/index.php/netlogo/

[19] RINCON, J.A. *Desarrollo de Entornos Virtuales Inteligentes Basados en el Metamodelo MAM5.* Universitat Politècnica de València, Departamento de Sistemas Informáticos y Computación (DSIC). 2014

[20] RINCON, J.A., COSTA, A., NOVAIS, P., JULIAN, V., CARRASCOSA, C. *Developing Emotional Intelligent Virtual Environments using EJaCalIVE.* Universitat Politècnica de València, D. Sistemas Informáticos y Computación & Centro ALGORITMI, Escola de Engenharia, Universidade do Minho, Braga. 2017.

[21] RINCON, J.A., GARCIA, E., JULIAN, V., CARRASCOSA, C. *Developing Adaptive Agents Situated in Intelligent Virtual Environments.* Universitat Politècnica de València, Departamento de Sistemas Informáticos y Computación (DSIC). 2014.

[22] RIVA, G. *Applications of Virtual Environments in Medicine. Methods of information in medicine*. Applied Technology for Neuro-Psychology Lab., Istituto Auxologico Italiano, Milan, Italy. 2003. DOI: 10.1267/METH03050524.

[23] RIZOO, A., PARSONS, T.D., PATRICK, P., BUCKWALTER, J. (2010). *A New Generation of Intelligent Virtual Patients for Clinical Training.* Conferencia: IEEE Virtual Reality Conference, Waltham, MA. 2010.

[24] RUSSELL, S., NORVIG, P. *Inteligencia artificial: un enfoque moderno*, 2ª edición. Madrid etc.: Pearson Educación, 2004, 2005. Capítulo 2. ISBN 842054003X.

[25] SERENKO, A., DETLOR, B. *Agent toolkits: A general overview of the market and an assessment of instructor satisfaction with utilizing toolkits in the classroom.* Michael G. DeGroote School of Business; McMaster University; Hamilton, Ontario. 2002. [En línea] Disponible: http://hdl.handle.net/11375/5601

[26] SHENDARKAR, A., VASUDEVAN, K., SEUNGHO, L., SON, Y. *Crowd simulation for emergency response using BDI agents 71ágs.71on immersive virtual reality*. 2008. En: *Simulation Modelling Practice and Theory* (2008), Volume 16, Issue 9 (págs. 1415-1429).

[27] SINCLAIR, B. *AR/VR spending to jump 69% in 2019 – IDC*. Diciembre, 2018. [En línea] Disponible: https://www.gamesindustry.biz/articles/2018-12-06-ar-vr-spending-to-jump-69-percent-in-2019-idc Última consulta: 21/08/2019.

[28] SOMMERVILLE, I. *Ingeniería del software*, 7ª edición. Madrid etc.: Pearson Addison Wesley, 2005. ISBN 8478290745.

[29] TAILLANDIER, P., GAUDOU, B., GRIGNARD, A. et al. *Building, composing and experimenting complex spatial models with the GAMA platform*. In Geoinformatica, Springer, 2018 (págs 299–322).

[30] TRANOUEZ, P., ERIC, D., LANGLOIS, P. *A multiagent urban traffic simulation*. 2012. En: *ESCAPE: Exploring by Simulation Cities Awareness on Population Evacuation*.

[31] WOOLDRIDGE, M. *An introduction to multiagent systems*. Chichester: John Wiley & Sons, cop. 2002. Capítulo 2. ISBN 047149691X.

A. Anexos

I. Caso de prueba – Juego de pelota

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<IVE name="Juego de pelota">
  <ive_workspaces name="Campo">
    <agents name="Jugador1">
      <artifacts>Cuerpo1</artifacts>
      <XMLFile>jugador1.jason</XMLFile>
    </agents>
    <agents name="Jugador2">
      <artifacts>Cuerpo2</artifacts>
      <XMLFile>jugador2.jason</XMLFile>
    </agents>
    <artifacts name="Cuerpo1">
      <actions name="Mover">
        <arguments name="dirección">
          <type>VECTOR3D</type>
        </arguments>
        <arguments name="aceleración">
          <type>NUMBER</type>
        </arguments>
        <precondition></precondition>
      </actions>
      <actions name="Rotar">
        <arguments name="Eje">
          <type>VECTOR3D</type>
        </arguments>
        <precondition></precondition>
      </actions>
      <perceivabe_property name="Color">
        <string>Rojo</string>
      </perceivabe_property>
      <perceivabe_property name="Alto">
        <double>20.0</double>
      </perceivabe_property>
      <perceivabe_property name="Ancho">
        <double>20.0</double>
      </perceivabe_property>
      <perceivabe_property name="Largo">
        <double>20.0</double>
      </perceivabe_property>
      <perceivabe_property name="Psoción">
        <vector3D>
          <x>0.0</x>
          <y>45.0</y>
          <z>0.0</z>
        </vector3D>
      </perceivabe_property>
      <internal_property name="Velocidad">
        <vector3D>
          <x>0.0</x>
```

```
            <y>0.0</y>
            <z>0.0</z>
          </vector3D>
        </internal_property>
      </artifacts>
      <artifacts name="Cuerpo2">
        <actions name="Mover">
          <arguments name="dirección">
            <type>VECTOR3D</type>
          </arguments>
          <arguments name="aceleración">
            <type>NUMBER</type>
          </arguments>
          <precondition></precondition>
        </actions>
        <actions name="Rotar">
          <arguments name="eje">
            <type>VECTOR3D</type>
          </arguments>
          <precondition></precondition>
        </actions>
        <perceivabe_property name="Color">
          <string>Azul</string>
        </perceivabe_property>
        <perceivabe_property name="Alto">
          <double>20.0</double>
        </perceivabe_property>
        <perceivabe_property name="Ancho">
          <double>20.0</double>
        </perceivabe_property>
        <perceivabe_property name="Largo">
          <double>20.0</double>
        </perceivabe_property>
        <perceivabe_property name="Posición">
          <vector3D>
            <x>0.0</x>
            <y>-45.0</y>
            <z>0.0</z>
          </vector3D>
        </perceivabe_property>
        <internal_property name="Velocidad">
          <vector3D>
            <x>0.0</x>
            <y>0.0</y>
            <z>0.0</z>
          </vector3D>
        </internal_property>
      </artifacts>
      <artifacts name="Pelota">
        <perceivabe_property name="Color">
            <string>Blanco</string>
          </perceivabe_property>
          <perceivabe_property name="Diámetro">
            <double>10.0</double>
          </perceivabe_property>
```

```
105.              <perceivabe_property name="Posición">
106.                <vector3D>
107.                  <x>0.0</x>
108.                  <y>0.0</y>
109.                  <z>0.0</z>
110.                </vector3D>
111.        </perceivabe_property>
112.            </artifacts>
113.            <artifacts name="Portería1">
114.              <actions name="Gol">
115.                <precondition>Pelota entra</precondition>
116.              </actions>
117.        <perceivabe_property name="Color">
118.                <string>Rojo</string>
119.              </perceivabe_property>
120.              <perceivabe_property name="Alto">
121.                <double>25.0</double>
122.              </perceivabe_property>
123.              <perceivabe_property name="Ancho">
124.                <double>60.0</double>
125.              </perceivabe_property>
126.              <perceivabe_property name="Largo">
127.                <double>20.0</double>
128.              </perceivabe_property>
129.              <perceivabe_property name="Posición">
130.                <vector3D>
131.                  <x>0.0</x>
132.                  <y>65.0</y>
133.                  <z>0.0</z>
134.                </vector3D>
135.              </perceivabe_property>
136.            </artifacts>
137.            <artifacts name="Portería2">
138.              <actions name="Gol">
139.                <precondition>Pelota entra</precondition>
140.              </actions>
141.              <perceivabe_property name="Color">
142.                <string>Azul</string>
143.              </perceivabe_property>
144.              <perceivabe_property name="Alto">
145.                <double>25.0</double>
146.              </perceivabe_property>
147.              <perceivabe_property name="Ancho">
148.                <double>60.0</double>
149.              </perceivabe_property>
150.              <perceivabe_property name="Largo">
151.                <double>20.0</double>
152.              </perceivabe_property>
153.              <perceivabe_property name="Posición">
154.                <vector3D>
155.                  <x>0.0</x>
156.                  <y>-65.0</y>
157.                  <z>0.0</z>
158.                </vector3D>
159.              </perceivabe_property>
```

```
160.            </artifacts>
161.          </ive_workspaces>
162.        </IVE>
```

II. Caso de prueba – Simulador de avión

```
1.  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2.  <IVE name="SimuladorAvion">
3.    <workspaces name="ControlMundo">
4.      <agents name="ControlVientos">
5.        <XMLFile>vientos.jason</XMLFile>
6.      </agents>
7.    </workspaces>
8.    <ive_workspaces name="Mundo">
9.      <agents name="Avion">
10.       <artifacts>Cuerpo</artifacts>
11.       <artifacts>Propulsor izquierdo</artifacts>
12.       <artifacts>Propulsor derecho</artifacts>
13.       <XMLFile>avion.jason</XMLFile>
14.     </agents>
15.     <artifacts name="Cuerpo">
16.       <actions name="Girar">
17.         <arguments name="vel">
18.           <type>VECTOR3D</type>
19.         </arguments>
20.         <precondition></precondition>
21.       </actions>
22.       <internal_property name="Masa">
23.         <double>36.5</double>
24.       </internal_property>
25.       <internal_property name="Orientación">
26.         <vector3D>
27.           <x>1.0</x>
28.           <y>0.0</y>
29.           <z>0.0</z>
30.         </vector3D>
31.       </internal_property>
32.     </artifacts>
33.     <artifacts name="Propulsor izquierdo">
34.       <actions name="Acelerar">
35.         <arguments name="aceleracion">
36.           <type>NUMBER</type>
37.         </arguments>
38.         <precondition></precondition>
39.       </actions>
40.       <internal_property name="Potencia">
41.         <double>0.0</double>
42.       </internal_property>
43.     </artifacts>
44.     <artifacts name="Propulsor derecho">
```

```
45.         <actions name="Acelerar">
46.           <arguments name="acelerar">
47.             <type>NUMBER</type>
48.           </arguments>
49.           <precondition></precondition>
50.         </actions>
51.         <internal_property name="Potencia">
52.           <double>0.0</double>
53.         </internal_property>
54.       </artifacts>
55.       <laws name="Gravedad">
56.         <condition_sentence></condition_sentence>
57.         <variable name="Gravedad">
58.           <vector3D>
59.             <x>0.0</x>
60.             <y>-9.8</y>
61.             <z>0.0</z>
62.           </vector3D>
63.         </variable>
64.       </laws>
65.       <laws name="Viento">
66.         <condition_sentence></condition_sentence>
67.         <variable name="Viento">
68.           <vector3D>
69.             <x>0.0</x>
70.             <y>0.0</y>
71.             <z>0.0</z>
72.           </vector3D>
73.         </variable>
74.       </laws>
75.     </ive_workspaces>
76. </IVE>
```