UML-AT VIZ

Sistemas Informáticos. 2008-2009

Facultad de Informática
Universidad Complutense de Madrid

Autores:
Jonás Fernández Reviejo
Fco. Javier Nieto Espinal
Lara Quijano Sánchez

Director:
Rubén Fuentes Fernández

Resumen:

En el modelado basado en agentes existen algunas limitaciones relevantes en el proceso de generar simulaciones a partir de un modelo no formal. Esta simulación no se puede hacer de una forma global si consideramos los diferentes aspectos de la creación de un modelo, que incluyen, abstracción, diseño, aproximación y codificación. Es necesario separar estos aspectos para facilitar el entendimiento, apreciar las relaciones y señalar el desarrollo de los conceptos que presenta dicho modelo. Además, en el caso de que en el proceso interactuaran diferentes roles, pueden surgir problemas de comunicación entre personas de diferentes especialidades y perspectivas.

Nuestro proyecto consiste en el modelado de una capa intermedia para resolver o reducir esta clase de problemas. Se plantea el uso de metamodelos como un lenguaje formal metodológico de alto nivel, que permite definir lenguajes en forma de diagramas que estén conceptualmente cerca del dominio del experto. A través de esta propuesta pretendemos facilitar la comunicación, especificación, implementación y validación de los modelos de simulación social, así como una visualización de éstos, que permita su análisis y comprensión por parte de los expertos en el área.

Palabras clave: modelado basada en agentes, metamodelos, simulación social, UML-AT.

Summary:

There are several relevant limitations in Agent-based modeling, specially the ones regarding the process of generating running simulations from a non-formal model. This simulation cannot be done in a global form if we consider the different aspects of the creation of a model, which includes abstracting, designing, approximating and coding. It is necessary to separate those aspects in order to facilitate the understanding, grasp the relationships and address the development of the concepts that this model shows. Besides, in the case of several roles participating in such process, communication problems can arise between people with different specialities and perspectives.

Our project consists on modelling a middle-layer which will be able to reduce or solve this kind of problems. It uses metamodels as a high-level formal methodological language, that allows to define languages in the form of diagrams conceptually close to the domain-expert. Through this approach we want to facilitate the communication, specification, implementation and validation of social simulation models, as well as a visualization of them that allows their analysis and comprehension by the domain experts

Keywords: agent-based modelling, metamodeling, social simulation, UML-AT.

Índice

1. Introducción

1.1. Planteamiento del Problema

Un modelo basado en agentes (MBA) es un tipo de modelo computacional que permite la simulación de acciones e interacciones de individuos autónomos dentro de un entorno, y permite determinar qué efectos producen en el conjunto del sistema. Los modelos simulan las operaciones simultáneas de entidades múltiples (agentes), en un intento de recrear y predecir las acciones de fenómenos complejos.

Una de las aplicaciones de la tecnología de agentes es la simulación de fenómenos sociales. Con este propósito se han desarrollado varias herramientas de simulación basada en agentes. Sin embargo, aunque el paradigma de agente debería facilitar, por sus características intencionales y sociales, la realización de modelos para simulación social, la utilización de las herramientas actualmente disponibles requiere un gran conocimiento práctico de programación, con lo que son poco apropiadas para los sociólogos. Ello hace necesario proporcionar nuevas herramientas de modelado, más fácilmente adaptables a las necesidades de los sociólogos, con conocimientos más básicos de programación. Asimismo, otro requisito que surge habitualmente en este tipo de sistemas es la escalabilidad, tanto en lo que se refiere a la edición de los modelos de simulación como a su ejecución. La simulación basada en ordenador está resultando en los últimos años muy útil especialmente en aquellos contextos de investigación donde los sistemas son demasiado complejos para la experimentación tradicional.

Por otro lado, hoy en día, es el programador (normalmente un ingeniero de software especializado) y no el investigador social el que debe realizar el trabajo de implementación del modelo informático y la simulación. Por tanto, a menos que éste tenga un adecuado conocimiento del modelo sociológico, resulta difícil que el científico social (que normalmente tampoco sabrá nada de programación) le comunique cómo quiere que sea el modelo exactamente. Como ambos mundos son bastante diferenciados se necesitan lenguajes intermedios que les permitan comunicarse. El caso ideal sería que el sociólogo, conocedor del modelo sociológico, pudiera servirse de esta herramienta de forma autónoma, aunque siempre habrá puntos donde necesite la ayuda del programador. De hecho, entre el programador y el sociólogo hay un ingeniero software especializado que ayude en el modelado para el modelo computacional desde el modelo abstracto del sociólogo.

1.2. Objetivos del proyecto

Para abordar estos problemas se propone desarrollar un entorno gráfico de simulación, orientado a dominios de aplicación específicos. Para su realización se plantea aplicar un entorno de desarrollo de sistemas multi-agente abierto y flexible.

Nuestra propuesta consiste en un software que supondrá un primer paso en los aspectos de alto nivel a la hora de simular un sistema social, ayudando a que en un futuro los sociólogos puedan elaborar simulaciones sin necesidad de depender de un experto en inteligencia artificial ni de tener conocimientos ni habilidades de programación.

Dicho propósito es innovador, pues aunque existen herramientas de simulación, como Repast o Swarm, que tienen entornos que facilitan parcialmente esta especificación, nosotros facilitamos la especificación con lenguajes específicos de dominio, gráficos y extensibles, que pueden ser fácilmente adaptados a nuevas necesidades del dominio. También cabe destacar que nuestra intención al plantear el software a desarrollar ha consistido en combinar la obtención de buenos resultados con la sencillez de su manejo (algo no siempre fácil de obtener), ya que esto último puede resultar decisivo para que la simulación sea viable como opción de investigación.

El proyecto construirá un entorno de simulación social para sistemas especificados según las pautas del paradigma de Ciencias Sociales conocido como Teoría de Actividad. Como lenguaje a usar en relación con la Teoría de Actividad se considerará la descripción propuesta en UML-AT. Además el sistema ha de soportar un número elevado de artefactos en la simulación. Para su visualización, se parte de entornos de visualización existentes y se proporcionan API para su conexión con nuestro entorno. De este modo, usar un nuevo entono de visualización queda reducido a crear los módulos de conexión intermedios necesarios.

1.3. Planteamiento del trabajo

Intentar definir las características de un sistema tan general es muy difícil y no están muy claras las barreras que se deben tener en cuenta para no caer en la especificidad de un lenguaje en concreto. Para conseguir abordar los objetivos propuestos, el trabajo se ha organizado de la siguiente manera:

1. La primera tarea que se llevará acabo será la Definición del lenguaje UML-AT, la generación del archivo ECore y la utilización del EMF para generar el Editor de Árbol y las Clases JAVA. Estas clases se utilizarán para crear los modelos a visualizar en la aplicación. A continuación se realizarán estos mismos pasos pero con el metamodelo de visualización.

2. La segunda tarea será realizar el Motor y Gestor de Mundo, el diseño de la arquitectura y funcionalidad básica que tendrán estos paquetes. Esto incluye diseñar los siguientes módulos:

- El motor, que recibirá como entrada el modelo a simular generado en el apartado anterior. Éste al cargarse en la aplicación pasará por un preprocesamiento donde se leerá dicho archivo y se separará por relaciones, entidades, etc. para que a continuación el Motor las identifique. Se organizará una máquina de estados, a través de ella se le irá administrando al gestor de mundo la situación en la que está el mundo actual separados por cuantos de tiempo.

- El gestor de mundo, que recibirá una situación para visualizar y el metamodelo de la visualización que al igual que para el metamodelo anterior se le realizará un preprocesamiento.

3. La tercera tarea que se realizará será la construcción de un Wrapper, que permitirá a través de las indicaciones del gestor de mundo crear instrucciones de creación de objetos utilizando una librería de visualización.

4. Por último realizaremos la GUI que utilizará el usuario. Esta le permitirá cargar el ejemplo a simular ya creado, y sus pautas de visualización. Una vez hecho esto podrá poner en marcha la ejecución, ejecutarla paso a paso si lo desea y pararla.

2. Revisión del estado del arte

2.1. Introducción

En las últimas décadas la simulación con ordenadores en general y la basada en modelación de agentes (ABM) en particular se ha convertido en una de las técnicas de modelación establecidas en muchos campos científicos y especialmente en el campo de las ciencias sociales, como la economía o la sociología [5]. ABM permite complementar los modelos no formales de los diseñadores, que normalmente están expresados en lenguaje natural, con modelos hechos por ordenador, que son más formales. Esta nueva propuesta, evita, al menos parcialmente, tener que hacer estudios laboriosos para hacerlo tratable analíticamente. Una de las principales ventajas de ABM, y en nuestra opinión lo que la distingue de otros paradigmas de modelación es que facilita una conexión directa entre entidades y objetivos del sistema, y las partes del modelo computacional que los representan [6].

El proceso necesario para transformar el sistema, que expresa un objetivo real de investigación, en un modelo de simulación es complejo. Este proceso requiere de la participación de diversos roles y de la realización de diferentes subtareas, para realizar este trabajo se necesitan diversas experiencias y competencias en el diseño, implementación y uso del arquetipo de simulación basado en agentes. Básicamente se pueden diferenciar tres roles principales en el proceso de remodelación: el experto en el dominio, el responsable del modelo y el ingeniero [7].

El rol del experto en el dominio idealmente sería llevado a cabo por una persona que se encargase de producir la primera conceptualización del sistema objetivo, el caso de estudio. Esta tarea incluye: definir los objetivos y el propósito de la tarea de modelado; identificando componentes relevantes del sistema y relaciones entre ellos; también describiendo las dependencias más importantes que existen entre ellos.

El trabajo del modelador es producir requerimientos formales para los modelos comenzando por las ideas que introdujo el responsable de la temática. Estos requisitos permitirán al responsable del siguiente rol (el ingeniero) formular un modelo factible, uno que pueda ser ejecutado en un ordenador. Sin embargo no todas las especificaciones formales pueden ser implementadas en un ordenador. El ingeniero es el responsable de encontrar una aproximación adecuada para el modelo formal que propone el modelador y que ésta, a su vez, pueda ser ejecutada en un sistema computacional con la tecnología disponible. Para concluir, existiría un último rol, el del programador, que es implementar el modelo ejecutable que ha diseñado el ingeniero.

En términos prácticos, los modeladores en ciencias sociales tienen dos problemas. Cuando se trata de desarrollos individuales, es difícil que una persona tenga toda la experiencia requerida, y por otro lado, cuando hablamos de grupos, la comunicación siempre es un problema.

El primer problema surge cuando la misma persona realiza todos los roles en un proceso. Una de las consecuencias negativas del modelado por ordenador, es que frecuentemente convierte a buenos científicos en malos programadores. La mayor parte de los científicos no poseen conocimiento de Ingeniería del Software. Como resultado producen herramientas computacionales pobremente diseñadas desde el punto de vista de la Ingeniería del Software. Además los científicos encuentran difícil entender el comportamiento detallado de dicho software ya que para conseguirlo necesitarían una comprensión completa de su implementación.

Por otro lado hay casos donde los científicos no siguen el principio "KISS" (Keep It Simple, Stupid; mantener las cosas sencillas) y la mayor parte de las veces necesitan introducir equipos multidisciplinares para su desarrollo. En dicha organización con cada miembro especializado en un rol diferente, surge el segundo problema: Los dominios de comunicación entre ellos son completamente diferentes (por ejemplo sociología e ingeniería del software). En la mayor parte de los casos, cuando finalmente se obtiene el programa es difícil captar las características sociales que fueron planeadas. Por tanto hay dificultades para asegurar que el programa realmente implemente su modelo conceptual.

Para tratar de apaliar estos problemas nuestro proyecto propone la creación de una herramienta que trabaja en un aspecto concreto del problema, que es la animación de las especificaciones siguiendo unas pautas predefinidas. Esta herramienta haría más fácil la representación de los conceptos relacionados con el trasfondo de la temática y al mismo tiempo facilitaría la labor del ingeniero del software. Debemos tener en cuenta que cualquier desajuste entre las especificaciones y el modelo actual si se pasa al siguiente escalón producirá un error.

Es mas, una herramienta de comunicación a alto nivel podría ayudar incluso en la validación. La validación del modelo es un proceso para determinar que el comportamiento del modelo representa el sistema real, con los niveles satisfactorios de confianza y exactitud que han sido determinados por el modelo de aplicación y su dominio. Cuando trabajamos con sistemas complejos, como es frecuente en modelación basada en agentes, los métodos tradicionales en dicha validación no son ampliamente aceptados. En esos casos una buena opción para la validación de modelos conceptuales es comprobar si los fundamentos y supuestos teóricos son razonables dentro del contexto de los objetivos de simulación. Esta validación estructural se realiza algunas veces en base a métodos participativos con expertos del dominio de modelación. Otras veces estos expertos no suelen tener conocimientos de ingeniería del software y por consiguiente un lenguaje dotado de descripciones a alto nivel facilita la comunicación, modificación y crítica de los modelos en los pasos de validación.

Nuestro proyecto facilita de forma automática dicho proceso de validación a la hora de crear un modelo, facilitando así la tarea del experto en el dominio, pues no necesita de conocimientos de ingeniería del software para comprobar si su ejemplo de investigación está correctamente modelado.

2.2 Otras aplicaciones

Cuando se aplica el modelado basado en agentes (ABM) a los procesos sociales, principalmente se usan conceptos y herramientas que provienen de las ciencias sociales y de la informática. Este proceso representa un enfoque con una metodología que podría en últimas instancias permitir dos innovaciones importantes:

- Un riguroso testeado, refinamiento y extensión de teorías existentes que normalmente resultan muy difíciles de formular y evaluar si se trata de usar las herramientas matemáticas y estadísticas que existen en la actualidad.

- Un entendimiento mayor de los mecanismos principales que se utilizan actualmente en el estudio de sistemas multiagentes. Dos ejemplos de posibles herramientas para ABM son, Repast [8], y Swarm [9], que están específicamente diseñadas para la aplicación de modelado basado en agentes en el campo de las ciencias sociales. Para construir un modelo con ellas, es decir una aplicación, estas plataformas proporcionan una serie de documentación, ejemplos de otras aplicaciones previamente hechas y código con las que permiten al usuario programarse dicho modelo, de una manera bastante sencilla ya que el software de estas dos plataformas es totalmente libre. A continuación describimos brevemente en que consisten y en que casos se pueden aplicar:

- **Repast (Recursive Porous Agent Simulation Toolkit)**, es una plataforma ampliamente usada que proporciona una colección de herramientas para el modelado basado en agentes y su simulación. Está específicamente diseñada para utilizarse en el ámbito de las ciencias sociales. Tiene múltiples implementaciones en varios lenguajes de programación como Java, C, Python y puede ser ejecutada en casi todas las plataformas existentes. Está construida con algunas características adaptativas muy concretas como algoritmos genéticos y regresión.

  Repast permite el estudio sistemático de comportamientos de sistemas complejos a través de experimentos computacionales controlados y replicables. Fue originalmente desarrollada en el Argonne National Laboratory de la Universidad de Chicago por David Sallach. Actualmente Repast es gestionada por la organización ROAD (*Repast Organization for Architecture and Development*).

  Repast proporciona un conjunto de clases, que constituyen un esqueleto a la aplicación a construir, estas son una colección de herramientas para la construcción y ejecución de simulaciones basadas en agentes, además proporciona un set de herramientas visuales como tablas esquemas y gráficos. Una característica especialmente interesante de Repast es la habilidad de integrar datos provenientes del GIS (*geographical information science*) directamente dentro de las simulaciones.

- **Swarm,** es un paquete software para la simulación de complejos sistemas multiagentes. Se desarrolló en el Instituto de Santa Fe. Con la construcción de esta plataforma se pretendía proporcionar una herramienta útil para los investigadores del estudio de modelación basada en agentes.

El software que proporciona Swarm incluye una colección de librerías de código que permite ejecutar simulaciones de ABM escritas en lenguaje Java o C. Estas librerías al igual que ocurría con Repast funcionan en un amplio rango de plataformas.

La arquitectura básica de Swarm es la simulación de colecciones de agentes interactuando concurrentemente entre ellos, con esta arquitectura asegura ofrecer una amplia variedad para la implementación de modelos basados en agentes.

Swarm es aún un software experimental lo que significa que aunque está suficientemente completo para su uso, siempre está en constante desarrollo.

2.3 Conclusiones

Los metamodelos que se realizan sirven como guía para que el investigador pueda construir modelos para su posterior ejecución. Por eso los metamodelos deben comprender toda la información que se debe a tener en cuenta para especificar un SMA, aunque manteniendo abiertas diferentes estrategias de llevar el desarrollo. A la hora de definir el orden en el que se generan los modelos y con qué nivel de detalle, existe total libertad, respecto a esto, no existe ninguna restricción.

Los metamodelos presentados incluyen resultados de la investigación en tecnología de agentes y áreas relacionadas. En cada uno se ha revisado brevemente qué parcelas de investigación habría que tener en cuenta y cómo influyen éstas en cada metamodelo. También se han considerado aspectos de consistencia de cada metamodelo con respecto a otros metamodelos.

El uso de los metamodelos se puede aplicar a temas relacionados con investigaciones actuales, como la planificación de tareas, filtrado colaborativo de información y diseño de agentes de interfaz. Además el proyecto ofrece enormes ventajas para las investigaciones sociológicas, ya que la propuesta facilita su comprensión al poder ser ejecutas y ofrece otro punto de estudio al permitir su visualización.

En cuanto a las dos plataformas vistas anteriormente, la principal diferencia que tienen respecto de nuestro proyecto están basadas en construir los modelos programando, por lo que los sociólogos no las pueden usar correctamente, sin embargo, nosotros pretendemos generar ese código de una forma transparente a partir de los modelos de los sociólogos. Por tanto Swarm y Repast podrían ser vistas como posibles plataformas de animación para vuestro sistema. Aunque nuestro proyecto no garantiza una flexibilidad completa, nuestra principal ventaja respecto de estas es el mayor nivel de abstracción.

3. Preliminares

3.1. UML-AT

UML-AT es un lenguaje para especificar sistemas sociales, lo hace a través de grafos, éstos contienen sistemas de actividad interconectados entre sí y los artefactos contenidos en ellos. Estos artefactos pueden ejercer diferentes roles, por ejemplo el rol *composition* para representar las transformaciones que indican que un artefacto se obtiene por composición de otros; Estas composiciones son realmente actividades AT simplificadas, pero al considerarlas de forma aparte facilita la descripción de sistemas sociales. Los A*ctivitySystems* representan los sistemas de actividad que agrupan los artefactos para una actividad. En la siguiente figura mostramos la estructura del lenguaje que propone UML-AT.

AT_Specification ::= (**AT_Role** +)(**AT_Structure** *)(**AT_Initialization** *)(**AT_Instances** *)

AT_Role ::= systems (*artefactID*) | activities (*artefactID*) |
compositions (*artefactID*) |
subjects (*artefactID*) | objectives (*artefactID*) |
objects (*artefactID*) | outcomes (*artefactID*) | tools (*artefactID*) |
communities (*artefactID*) | rules (*artefactID*) | divisions (*artefactID*)

AT_Structure ::= activitySystem (*activitySystemID*, *activityID*) |
executedBy (*activitySystemID*, *subjectID*) | try (*activitySystemID*, *objetiveID*) |
transform (*activitySystemID*, *objectID*) | use (*activitySystemID*, *toolID*) |
produce (*activitySystemID*, *outcomeID*) | consume (*activitySystemID*, *artefactID*) |
include (*activitySystemID*, *artefactID*) | accomplishedBy (*activitySystemID*, *communityI*
ruledBy (*activitySystemID*, *ruleID*) | organizedBy (*activitySystemID*, *divisionID*) |
ruleContext (*ruleID*, *artefactID*) | rulePositive (*ruleID*, *artefactID*) | ruleNegative (*ruleI*
input (*compositionID*, *objectID*) | output (*compositionID*, *outcomeID*) |
divisionOfLabour (*divisionID*, *communityID*, *subjectID*, **Relation**) |
contribute (*artefactID*, *artefactID*, **Contribution**) |
pursue (*subjectID*, *objetiveID*) | surpass (*subjectID*, *objetiveID*, *objetiveID*)

Relation ::= superior

Contribution ::= guarantee | essential | positively | undefined | negatively | impede

AT_Initialization ::= initial (*artefactID*)

A continuación explicamos las variables y elementos de este lenguaje de especificación de sistemas sociales:

Las variables en forma de *roleID* representan los identificadores de tipos en los sistemas sociales cuando ejecutan un rol dado. De hecho, todas las propuestas tienen como argumentos identificadores de tipo artefacto, por ejemplo *artefactID*; el resto de las variables *roleID* simplemente señalan el rol esperado para los tipos de artefactos en la proposición. Las variables *instanceID* corresponden a los tipos de las instancias. Las proposiciones que describen la estructura estática de los sistemas sociales son:

- *AT_Role.* Esta categoría incluye una colección de predicados que describen la relación de instanciación entre el nivel de metamodelado del lenguaje, usado para describir los sistemas sociales, y el nivel de modelado. Hay una proposición en forma de *role(artefactID)* para cada rol AT en el dominio y por cada tipo de artefacto que lo ejecuta en el sistema social. Para aclararlo se puede decir que el significado de *role(artefactID)* es que el tipo de artefacto *artefactID* ejecuta el rol de *role* en el sistema social. La colección de *artefactID* no está definida explícitamente con una proposición sino que se define como la unión de los otros predicados de *roleID*. Estas proposiciones corresponden a la relación *instance of* de los metamodelos. La proposición *instanceOf* relaciona instancias con estos tipos. Por lo tanto el significado de *instanceOf(artefactID, instanceID)* es que la instancia del artefacto *instanceID* es de tipo *artefactID* en el sistema social. Estas proposiciones corresponden a la relación *instance of* de los modelos. Ambos artefactos junto con sus instancias pueden participar en varias de estas relaciones, aunque hay algunas limitaciones semánticas. Por ejemplo, un tipo de artefacto puede ser al mismo tiempo un *activity* o un *outcome*, que significa que este es el resultado de algún sistema de actividad, pero no puede ser también un sujeto, que sería el participante activo de un sistema de actividad.

- *AT_Structure.* Los predicados en esta categoría son los que hacen explícitamente la estructura de los sistemas de actividad en UML-AT añadiendo *compositions* y *constrains* en la ejecución. Los predicados que representan estas relaciones están en la primera forma normal. Por ejemplo, el predicado *activitySystem(activitySystemID, activityID)* significa que el tipo de sistema de actividad *activitySystemID* tiene un tipo de actividad *activityID*. Cabe destacar que la mayoría de estos predicados pueden aparecer varias veces para el mismo parámetro inicial. Por ejemplo, puede haber varias proposiciones *ruleContext* con el mismo tipo de *rule id* para especificar un contexto que incluye diferentes artefactos. Las únicas excepciones a esto son las proposiciones *activitySystem*, ya que un sistema de actividad tiene exactamente una actividad, y *output*, ya que una composición solo produce un artefacto. Hay que darse cuenta de que en vez de usar los identificadores generales *artefactID*, las proposiciones usan variables de tipo *roleID*. Esto simplemente indica que el tipo *roleID* es un *artefactID* que aparece en una proposición *role(artefactID).*

La categoría *AT_Structure* requiere los siguientes comentarios adicionales:

La proposición *initial* determina que tipos de artefactos e instancias están inicialmente disponibles en los sistemas. Si un tipo está disponible inicialmente, todas sus instancias estarán también disponibles. Cuando consideramos el comportamiento dinámico de los sistemas, los artefactos disponibles deben estar inicialmente disponibles o ser creados (y no destruidos) por la ejecución de una actividad o una composición.

Las proposiciones *input* y *output* definen respectivamente las entradas que se requieren en una composición y sus resultados. Como se dijo anteriormente una composición puede tener nada más que un único resultado. La proposición *pursue* relaciona un sujeto con los objetivos que este intenta llevar a cabo. La proposición *surpass* define la prioridad entre los objetivos perseguidos por un sujeto.

Existen dos tipos de limitaciones para la ejecución de sistemas de actividad, por ejemplo las *rules* y las *divisions of labour*. Todas las reglas que son aplicables a un sistema de actividad (por ejemplo la proposición *ruledBy*) deben ser satisfactorias para que se pueda ejecutar ese sistema. Una regla está compuesta por un contexto (por ejemplo la proposición *ruleContext*), por restricciones positivas (por ejemplo la proposición *rulePositive*) y por restricciones negativas (por ejemplo la proposición *ruleNegative*). El contexto de una regla se satisface si alguno de sus artefactos está disponible. Las restricciones positivas de una regla se satisfacen si todos sus artefactos están disponibles, y las restricciones negativas si ninguno de sus artefactos está disponible. La proposición *divisionOfLabour* describe las relaciones en una comunidad que estén relacionadas con sus actividades. Actualmente solo existe una clase de relación que se llame *superior*. Una proposición de la forma *divisionOfLabour(divisionID, communityID, subjectID, superior)* significa que los sujetos que pertenecen a la comunidad *communityID* persiguen todos los objetivos que el sujeto *subjectID* persigue.

La proposición *contribute* indica como los artefactos influyen en los propósitos de otros artefactos. Por ejemplo, la presencia de un artefacto puede asegurar (*guarantee*) el éxito en la realización de algún objetivo o afectar a contribuir negativamente (*contribute negatively*) en otro artefacto. Con la excepción de los valores *guarantee*, *essential* y *impede*, el resto implica efectos que no son definitivos. Por ejemplo una relación *contribute negatively* puede hacer un objetivo más difícil de alcanzar, pero no lo impide.

Con todos los elementos anteriores la especificación de un sistema social U con UML-AT tiene la forma de U = (I, S), donde I es la colección de los identificadores de los tipos e instancias en el sistema, y S es la colección de proposiciones vistas en el lenguaje que caracterizan sus estructura y comportamiento. A continuación ponemos un ejemplo de UML-AT para especificar sistemas sociales. En el que se incluye las proposiciones S del sistema social para consultarías. La colección I de los identificadores de tipo contiene todos los identificadores en estas proposiciones.

Procedemos a ver un ejemplo. Declaración de roles:

systems (*studySystem*)	systems (*evaluateServiceSystem*)	systems (*evaluateQualitySystem*)
activities (*studyInform*)	activities (*evaluateService*)	activities (*evaluateQualityWithCustomer*)
subjects (*customer*)	subjects (*consultant*)	
objectives (*getInformation*)	objectives (*helpConsultancyFirm*)	objectives (*getAssesment*)
objects (*request*)	objects (*inform*)	objects (*reportAboutService*)
outcomes (*knowledgeForTheCustomer*)	outcomes (*reportAboutService*)	outcomes (*qualityAssesment*)
tool (*inform*)	tool (*request*)	
rules (*supportTheConsultancyFirm*)		

Especificación de la estructura:

activitySystem (*studySystem*, *studyInform*)

activitySystem (*evaluateServiceSystem*, *evaluateService*)

activitySystem (*evaluateQualitySystem*, *evaluateQualityWithCustomer*

executedBy (*studySystem*, *customer*)

executedBy (*evaluateServiceSystem*, *customer*)

executedBy (*evaluateQualitySystem*, *consultant*)

try (*studySystem*, *getInformation*)

try (*evaluateServiceSystem*, *helpConsultancyFirm*)

try (*evaluateQualitySystem*, *getAssesment*)

transform (*studySystem*, *request*)

transform (*studySystem*, *inform*)

transform (*evaluateServiceSystem*, *request*)

transform (*evaluateServiceSystem*, *inform*)

transform (*evaluateQualitySystem*, *reportAboutService*)

use (*evaluateQualitySystem*, *request*)

use (*evaluateQualitySystem*, *inform*)

produce (*studySystem*, *knowledgeForTheCustomer*)

produce (*evaluateServiceSystem*, *reportAboutService*)

produce (*evaluateQualitySystem*, *qualityAssesment*)

ruledBy (*studySystem*, *supportTheConsultancyFirm*)

ruleContext (*supportTheConsultancyFirm*, *supportTheConsultancyFirm*)

rulePositive (*supportTheConsultancyFirm*, ∅)

ruleNegative (*supportTheConsultancyFirm*, ∅)

contribute (*supportTheConsultancyFirm*, *helpConsultancyFirm*, positively)

contribute (*knowledgeForTheCustomer*, *getInformation*, guarantee)

contribute (*reportAboutService*, *helpConsultancyFirm*, guarantee)

contribute (*qualityAssesment*, *getAssesment*, guarantee)

pursue (*customer*, *getInformation*)

pursue (*customer*, *helpConsultancyFirm*)

pursue (*consultant*, *getAssesment*)

surpass (*customer*, *getInformation*, *helpConsultancyFirm*)

3.2. Metamodelado EMF. ECORE [2]

Los metamodelos son un formalismo para definir la sintaxis abstracta de la modelación de lenguajes. Al utilizarlos encontramos un problema común que es el diseño de un metamodelo que encaje de acuerdo con las características elegidas para el lenguaje a definir. A continuación mostramos las pautas que nos sirvieron para ayudarnos en la definición de dichos metamodelos usando "Eclipse Modeling Framework". Estas pautas muestran un enfoque basado en la entidad relación tanto para extraer las características estructurales relevantes del modelado de lenguajes como para la organización de sus metamodelos. Algunos ejemplos de las características estructurales que estamos considerando son las relaciones n-arias o las propiedades relacionadas con las propias relaciones o sus extremos.

Actualmente existe un creciente interés en el uso de metamodelos para la descripción de la sintaxis abstracta de los lenguajes de modelado (ML). Este interés surge de dos líneas principales de investigación, por un lado la ingeniería de modelos (Model-Driven Engineering, MDE) apoya al desarrollo de sistemas a través de transformaciones automáticas de sus modelos. La definición de estas transformaciones recae en metamodelos que definen la fuente y objetivo de estos lenguajes. La otra línea de interés es los lenguajes de modelado de un dominio específico (Domain-Specific Modeling Languages, DSML). Estos lenguajes se construyen para resolver problemas específicos de dominio usando una base conceptual cercana a la que aceptada comúnmente por los expertos en un dominio dado. Los DSML se describen normalmente con metamodelos de los que se pueden generar automáticamente herramientas de tipo "CASE" adaptadas a ellos. Estos usos de los metamodelos requieren un diseño cuidadoso para obtener ciertas características como la representación de relaciones n-arias usando representaciones heterogéneas adaptadas a los elementos de modelado diferentes o para obtener modelos de navegación eficientes.

La complejidad de definir estos metamodelos lleva a los diseñadores a apoyarse en guías para ayudarlos en este proceso. Para este objetivo ejemplos de metamodelos como los desarrollados para los lenguajes de metamodelado como ECore nos ofrecen ayudas para desarrollar otros lenguajes. Estas directrices consideran únicamente grupos reducidos de características estructurales en el modelado de lenguajes o se centran en aspecto relacionados con los metamodelos pero que acaban siendo diferentes de los de la sintaxis abstracta.

Una vez ya hemos expuesto los beneficios y los problemas de esta clase de directrices, a continuación explicaremos lo más exactamente posible las pautas de los metamodelos MF usando un enfoque entidad-relación. Ya que son lo que facilitará la definición de modelación de lenguajes a través de metamodelos. Los ML se pueden clasificar en basados en conexión, que representan objetos interconectados, y basados en geometría, que consideran la distribución espacial de los elementos. Este trabajo estrecha su espacio con los ML basados en conexiones, al considerar que un grupo homogéneo de ML con ciertas características comunes permite dar una guía mas detallada para su diseño.

El análisis de los ML basados en conexión frecuentemente adopta una perspectiva entidad-relación (ER). Esta perspectiva abstrae los elementos del lenguaje como entidades relaciones o elementos auxiliares como extremos de la relación o propiedades. Esta guía también adopta el enfoque entidad-relación para organizar su conocimiento.

Las pautas que aquí seguimos consideran una colección de características estructurales presentadas en la sintaxis abstracta de los ML basados en conexiones. Específicamente considera la presencia de relaciones n-arias, atributos en las relaciones y atributos en los extremos de las relaciones. Mas adelante cuando estas directrices defines un framework para los metamodelos con diferentes alternativas para la representación de ER de estas características. Estas alternativas permiten crear metamodelos que satisfacen restricciones adicionales de la representación, como la adaptabilidad a los cambios, numero mínimo de elementos o facilidad de procesado para sus modelos. Para apoyar estas decisiones de diseño sobre el metamodelo, la guía ofrece ayuda sobre como diseñar el metamodelo que encaja mejor para un ML con ciertos requerimientos, como por ejemplo características estructurales y restricciones adicionales. Esta ayuda se presenta como una colección de preguntas para el diseñador que le llevan a una instanciación adecuada para el framework que define el metamodelo.

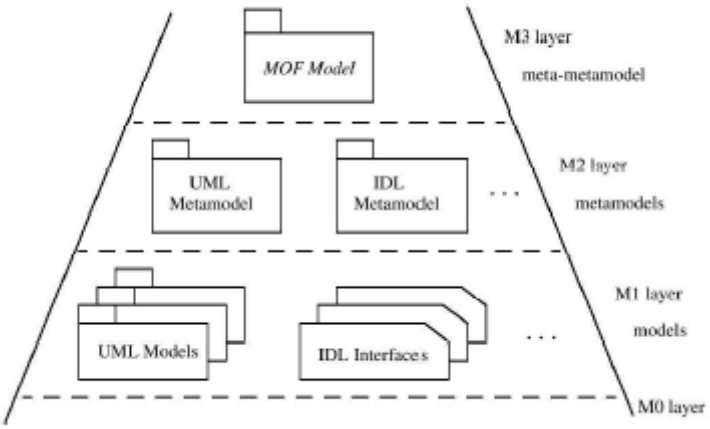
Actualmente existen dos lenguajes principales de meta-metamodelado, MOF y ECore. MOF es un lenguaje para estándares. Sin embargo, ECore ofrece una herramienta de soporte mejor con el Eclipse Modeling Framework (EMF). Por esta razón, y para facilitar la experimentación y diseminación, nosotros adoptamos ECore por su descripción y herramientas.

Bases del metamodelado

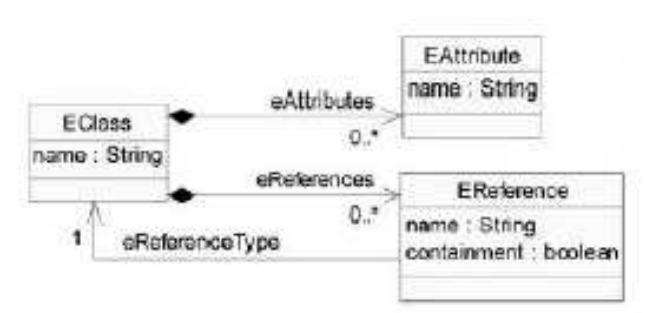
Los metamodelos son un mecanismo para definir sintaxis abstractas de ML, que determina los modelos validos de esos lenguajes. La OMG explica las relaciones entre los modelos y los metamodelos en la definición de ML con la arquitectura de metadatos MOF. Aunque esta arquitectura se puede manejar con un número arbitrario de capas de abstracción, normalmente se consideran las cuatro capas que son visibles:

- Capa del meta-metamodelo (M3). Es la capa de los lenguajes de meta-metamodelado como MOF o ECore.
- Capa de metamodelo (M2): es la capa donde se definen los lenguajes de metamodelos usando para ello los lenguajes de meta-metamodelado de la M3.
- Capa de modelado (M1): Contiene modelos específicos que se definen a través de los metamodelos en la M2.
- Capa de los objetos de usuario (M0): define los objetos reales del un problema, como instancias de los modelos de la M1.

Lo vemos en la siguiente figura:



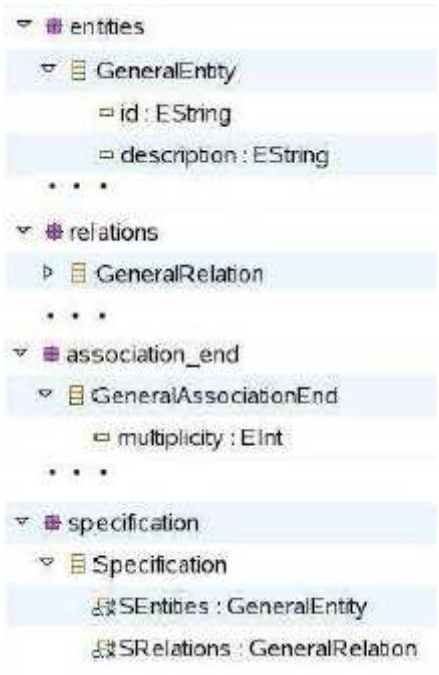
A continuación vemos los elementos principales del lenguaje usando ECore como metamodelos:



- *EClass*: Es el único elemento capaz de contener *EAttibutes* o *EReferrences*. Las instancias de *EClass* se pueden extender de otras instancias de *EClass*.
- *EAttribute*: Es un elemento que contiene los calores de los tipos primitivos, como enteros, strings o caracteres.
- *EReference*: Representa las relaciones binarias entre dos clases *EClasses*. La instencia de la fuente(source) *EClass* contiene la instancia de la *EReference*. El objetivo(target) de *EClass* es el *EType* de la instancia de *EReference*. Hay dos tipos de *EReferences*:
    - *constainment* (de contención) → El elemento que representa el contenedor se contiene un elemento que representa el objeto referenciado.
    - *non-constainment* → El contendor tiene una dirección del elemento que esta representando el elemento referenciado.

- *EPackage*: Representa un paquete con elementos del metamodelo.
- *ENum*: define un tipo de enumeración que contiene diferentes literales como *ENumLiterals*.

ECore es la base de Eclipse Modeling Framework (EMF). EMF permite la construcción de editores y otras aplicaciones basadas en modelos de datos. Proporciona librerías y plug-ins para producir colecciones de clases Java de los metamodelos, lo que hace posible visualizar y editar su correspondiente ML. A continuación mostramos los metamodelos generados con ECore y los tres editores que se generan de ellos con EMF.



En estos editores, los *EPackages* contienen otros *EPackages* o *EClasses*, y las *EClasses* contienen *EReferences* y *EAttributes*. El símbolo a la izquierda del nombre del elemento lo distingue de otros tipos difrence de elementos de ECore. Vemos ejemplos de *EPackages* (*entities* y *relations*), de *EClasses* (*GeneralEntity* y *GeneralRelation*), de *EReferences* (*SEntities* y *SRelations*) y de *EAttributes* (*id*, *description* y *multiplicity*).

Frameworks para metamodelos

Un Framework proporciona una estructura para organizar los contenidos del metamodelos de un ML, y posibilita la representación para sus entidades y relaciones. Mientras que propone una única representación para las entidades, hay diferentes representaciones para las relaciones y que satisfacen los diferentes requerimientos. Como vimos en la figura anterior el editor EMF contiende cuatro *EPackages* que corresponden con los diferentes elementos de una perspectiva ER de una ML basada en conexión. Estos paquetes definen la herencia de jerarquías de las *EClasses* con una única raíz abstracta para cada paquete. De acuerdo con la semántica de ECore, las *EClasses* heredan atributos de estas jerarquías. Por consiguiente, definir atributos en la raíz de una de estas jerarquías es un mecanismo para definir atributos para todos los elementos de un paquete dado, Los cuatro paquetes son:

- *Entities*: Contienen todas las entidades de un lenguaje de modelado. La raíz de la jerarquía es una entidad única y abstracta que se llama *GeneralEntity*.
- *Relations*: este paquete contiene todos los cuerpos de las relaciones del ML si fueran necesarios. Este es el caso, por ejemplo, en el que el ML incluye relaciones n-arias o atributos relacionales. La raíz de esta jerarquía es la *GeneralRelation EClass*.
- *Association_end* : Incluye todas las terminaciones de las relaciones de un ML.

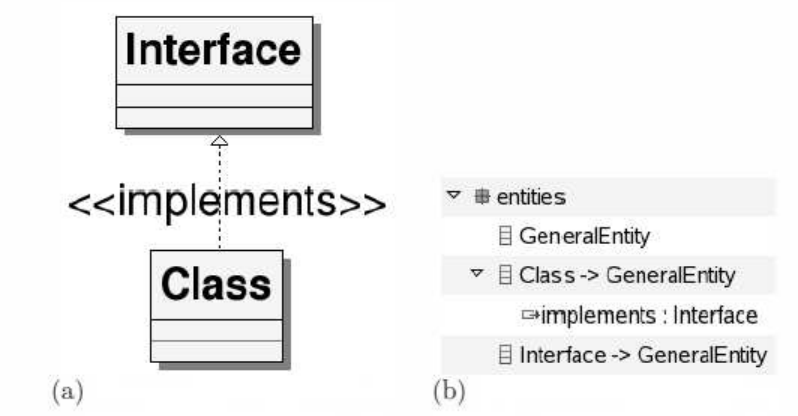
- Por ejemplo para los ML con atributos en sus terminaciones de relación se necesitan estos elementos. El paquete tiene una jerarquía cuya raíz es una relación única y abstracta llamada *GeneralAssociationEnd*.
- *Specification*: Este paquete contiene las *Specification EClass*. Sus instancias son los elementos raíz de los modelos. Una instancia de *Specification* contiene dos contenedores múltiples *EReferences*, los *SEntities* y los *SRelations*, con todas las entidades y relaciones del modelo de la capa M1 respectivamente.

El Framework considera únicamente la representa de una entidad como un *EClass* con *EAttributes*. Como hemos mencionado anteriormente las entidades *EPackage* contienes a estas *EClasses* y se organizan en una jerarquía en la entidad abstracta *GeneralEntity*.

Una posible representación es la *No-EClass* que describe las relaciones con un único *EReferences* no-contenedor. Para agrupar los tipos de entidades que pueden estar ligadas por una relación, el nivel M2 de metamodelado contiene una *EReference* para cada una de las posibles meta-entidades que la relación puede ligar. La *EReference* en esta representación se sitúa en la *EClass* que corresponde a la entidad fuente de la relación y que apunta hacia la *EClass* que representa la entidad objetivo de la relación.

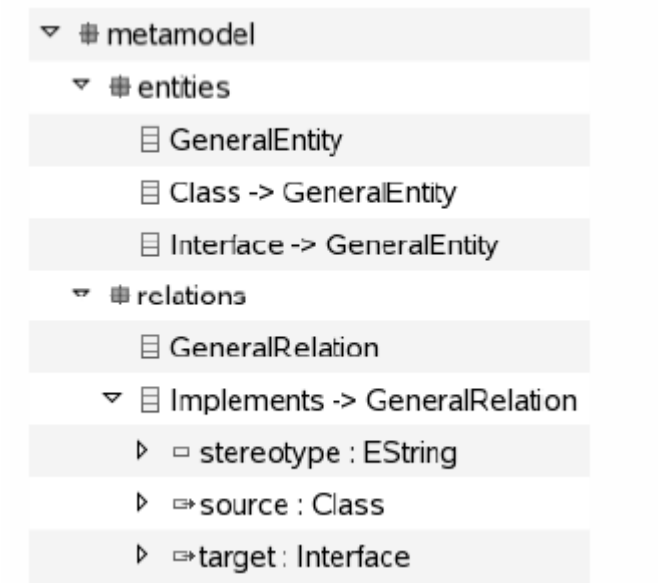
De acuerdo con la especificación *ECore*, estas opciones tienen varias consecuencias. Primero, una *EClass* puede ser señalada por más de una *EReference* contenedora. Dado que la *EReference* en esta representación es no-contenedora, se pueden relacionar varias relaciones a la misma entidad. Segundo, las *EReferences* de la capa M1 pueden conectar exactamente dos *EClasses* y no tienen atributos, así que esta representación solo es valida para representaciones binarias sin atributos.

Como ejemplo de esta representación tenemos la siguiente figura, que muestra parte del metamodelado UML con las relaciones entre clases e interfaces. La figura b describe la representación *No-EClass* de esta relación. Los *EReferences* aparecen en la clase *EClass* y señala a la interfaz *EClass*.



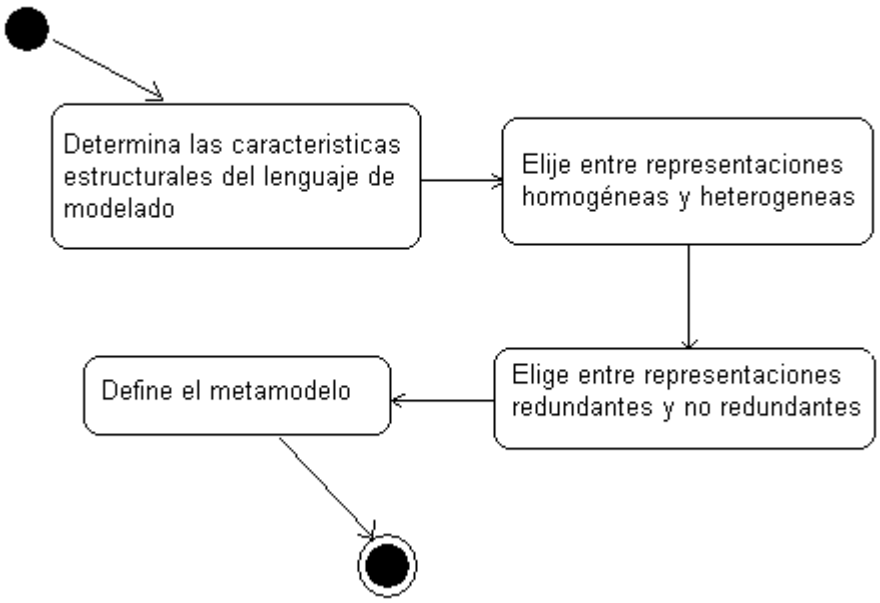
La representación del cuerpo *EClass* usa para cada relación una *EClass* y varias *EReferences*. La *EClass* representa el cuerpo de la relación. Contiene los atributos de la relación como *EAttributes* y los *EReferences* que conectan la relación con varias entidades. En el nivel de metamodelado, hay *EReferences* para cada una de las posibles meta-entidades que la relación puede unir, lo que agrupa los tipos de las posibles entidades a unir en el nivel de modelo. Con esta implementación, la representación de *cuerpo-EClass* es factible para representar relaciones n-arias con atributos.

La siguiente figura presenta las *EClasses* implementadas que representan las relaciones implementadas, lo que permite incluir un atributo "*stereotype*" en la relación. Las *EReferences* fuente y objetivo conectan la *EClass* de la relación con el cuerpo de las *EClasses* que representan la fuente y el objetivo de las entidades ligadas por la relación respectivamente.



Guía para la definición de metamodelados con ECore

Esta guía pretende ayudar al diseñador en la definición de un metamodelo para un ML dado el proceso sigue el siguiente diagrama de actividad:



Comienza determinando las principales características de un lenguaje con una colección de preguntas preliminares; luego, la guía propone un metamodelo para especificar el lenguaje de acuerdo con las elecciones del diseñador.

La primera actividad de la guía es especificar las características estructurales que caracterizan el ML. Las características posibles que la guía considera vienen del enfoque ER que se adopta en ella. Dado un ML cada relación puede tener diferentes características estructurales. El diseñador debe saber a conciencia en este punto cuantas relaciones tiene cada una de las características estructurales consideradas. Estas estimaciones deben considerar el estado actual del ML, pero también su posible evolución. De este modo la guía recomendará una representación que facilite el mantenimiento futuro del metamodelo.

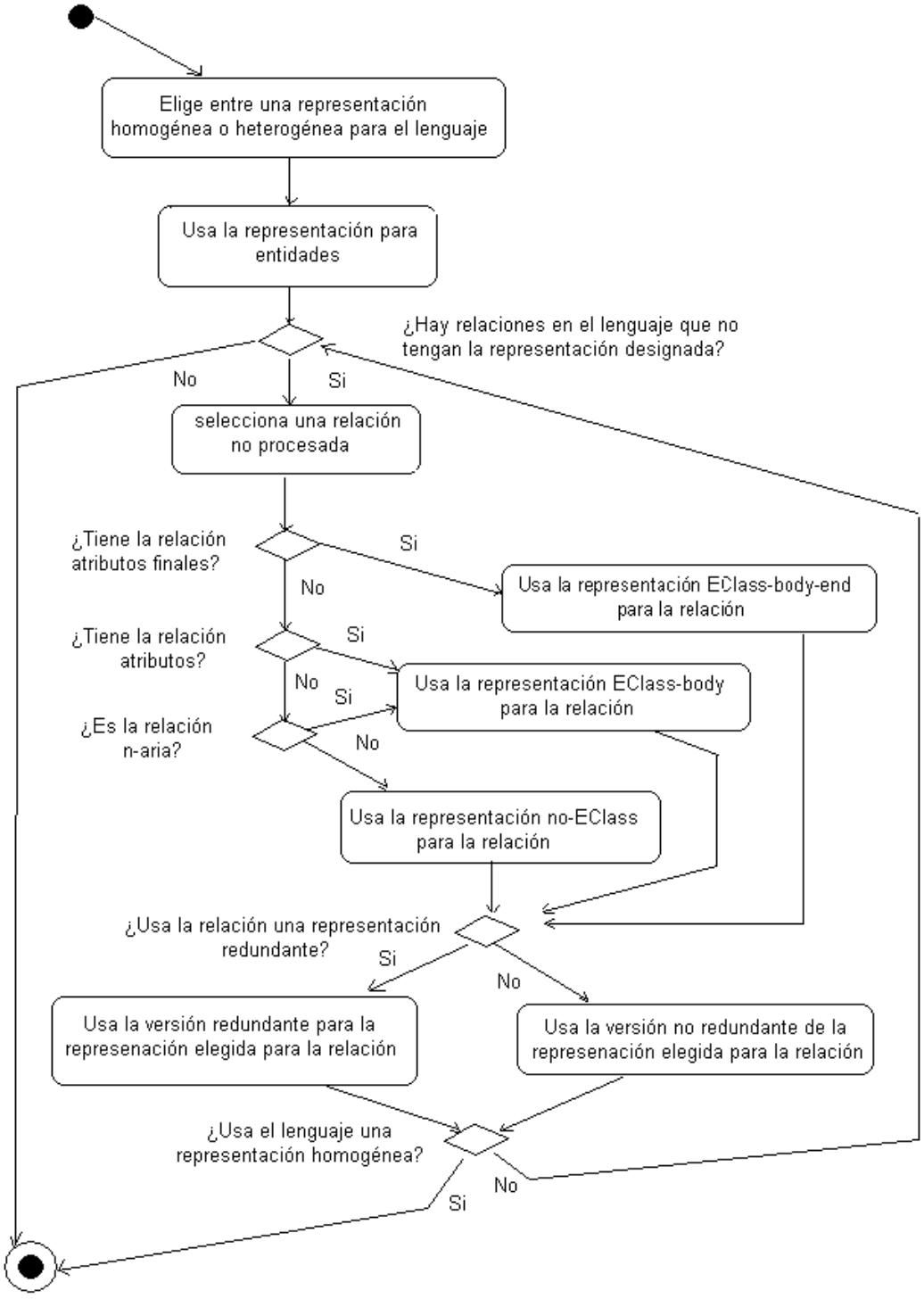
La segunda actividad en la guía elige entre representaciones homogéneas y heterogéneas para las relaciones. Esta decisión supone un equilibrio entre minimizar el número de elementos del metamodelo y facilitar en el cambio y el uso.

La guía recomienda usar representación homogénea para lenguajes en evolución. Esta representación permite a las relaciones tener más características estructurales en el futuro, ya que selecciona la representación más flexible para todas ellas. Además, los cambios en la representación de las relaciones se pueden aplicar uniformemente a todo el metamodelo sin importar los cambios que se produzcan. Por otra parte, la representación heterogénea es adecuada para los lenguajes de modelado estables, donde cada relación tiene una característica estructural diferente.

La tercera actividad elige entre representaciones redundantes y no redundantes. Como se explicó anteriormente las representaciones no redundantes usan una *EReference* para conectar dos *EClasses*, mientras que la representación redundante usa dos *EReferences* para el mismo fin. Esto no significa que la representación de una relación use solamente *EReferences*. Dependiendo de las características estructurales de una relación, su representación puede contener varias *EClasses*, *EReferences* y *EAttributes*.

Considerando los equilibrios que han de mantenerse, vistos ya anteriormente, el diseñador debe elegir la representación redundante si se necesita un acceso rápido a las relaciones conectadas con una entidad. La representación no redundante es una mejor alternativa cuando es importante reducir el número de elementos de modelado usados en el metamodelo. Esta última representación se recomienda también con algunas herramientas que los editores para ML generan automáticamente para sus metamodelos, como EMF. Si los editores generados carecen de mecanismos para manejar consistentemente los pares de *EReferences* en la representación tratada, este manejo será también una tarea del usuario, esto incrementará notablemente la carga del trabajo.

La última actividad construye el metamodelo para el lenguaje considerando las elecciones del diseñador en las actividades previas. Este proceso aparece en la siguiente figura. El metamodelo es construido como una instanciación del framework para metamodelos. Este proceso propone el uso de una única representación disponible para las entidades y elige la que mejor encaje para cada una de las relaciones de acuerdo con las características seleccionadas. En el caso de la representación homogénea, todas las relaciones adoptaran la misma representación. Esta representación será la más flexible de entre las requeridas por las relaciones en el lenguaje.


Elige entre una representación homogénea o heterogénea para el lenguaje
Usa la representación para entidades
¿Hay relaciones en el lenguaje que no tengan la representación designada?
No
Si
selecciona una relación no procesada
¿Tiene la relación atributos finales?
Si
No
Usa la representación EClass-body-end para la relación
¿Tiene la relación atributos?
Si
No
Usa la representación EClass-body para la relación
¿Es la relación n-aria?
Si
No
Usa la representación no-EClass para la relación
¿Usa la relación una representación redundante?
Si
No
Usa la versión redundante para la represenación elegida para la relación
Usa la versión no redundante de la represenación elegida para la relación
¿Usa el lenguaje una representación homogénea?
No
Si

Conclusiones

Los metamodelos están cada vez mas involucrados en la creación de sistemas de información [2]. El Model-Driven Development y el Domain-Specific Model Languages aborda el uso de metamodelos para definir la sintaxis abstracta de su ML y para generar herramientas de soporte para ellos.

Aquí se ha visto una breve guía para definir estos metamodelos. La guía se centra en los ML basados en conexión y adopta un enfoque entidad-relación para su análisis. Considera varias características estructurales que pueden aparecer en un ML. Además también tiene en cuenta requerimientos complementarios, como: las flexibilidad a través de la representación homogénea; utilizar el mínimo número de elementos en el metamodelo con la representación heterogénea y no redundante; y eficiencia en el proceso con la representación homogénea y redundante. Esta guía también presenta representaciones para elementos en ML que encajan con estas características. Finalmente proporciona un proceso de decisión para diseñar un metamodelo adecuado para las características previstas del ML.

A parte de su aplicación en el diseño del lenguaje la guía se ha definido basándose en el lenguaje ECore apoyándose en su uso en proyectos basados en sus herramientas.

4. Análisis

4.1. Descripción

4.1.1. Requisitos, alcance, viabilidad

Requisitos: Destalles sobre el proyecto que vamos a realizar.

El sistema constará 2 módulos básicos para gestionar la animación en sí:

- *API.* El módulo *API* implementa 3 interfaces. La primera permite proporcionar la descripción UML-AT de los elementos que componen el sistema. La segunda contiene la descripción gráfica de los elementos de la primera interfaz. Esta descripción considera aspectos como la imagen usada para su visualización, posición o interacciones visuales permitidas. La tercera interfaz especifica la animación como una secuencia de pares sujeto-tarea a ejecutar. Esta última interfaz puede admitir información adicional como repetición de tareas, tiempo de ejecución o incluso condiciones. La base de este interfaz para animación puede ser el propio UML-AT.

- *Robot.* El módulo tiene un comportamiento como al del *API,* pero la información relativa a las 3 interfaces se proporciona vía ficheros.

Los ficheros se especificarán con los correspondientes DTD o "XML Schema"[10]. Además se construirán editores para ellos, usando las facilidades del "Eclipse Modeling Framework “EMF"[11]. La definición de los lenguajes se basará en UML-AT, utilizado para definir sistemas de Teoría de Actividad.

El módulo API estará constituido a su vez por un módulo base capaz de interactuar con la plataforma elegida de visualización y una capa superior que transformara las peticiones de visualización a llamas a este módulo inicial. Ello permitiría sustituir la plataforma de visualización por otra sin más que modificar el módulo base. De la misma manera, el módulo *Robot* se construirá sobre el de *API* e incluirá separados los componentes para procesar los ficheros.

Sobre los módulos anteriores se construirá un componente de gestión del mundo virtual. El sistema se ocupará de actualizar el estado del mundo (datos y visualización) según los eventos que se produzcan en el entorno, tales como la ejecución de tareas por parte de los sujetos.

Finalmente habrá un componente visual de gestión del mundo virtual. Este componente permitirá acceder interactivamente al módulo *API* y arrancar el módulo de gestión. Este módulo permite cargar ficheros con descripciones de mundos y también especificar interactivamente estos componentes. Los sujetos, que son los elementos activos de la simulación podrán ser autónomos o interactivos. Los autónomos seleccionarán por sí mismo tareas válidas a ejecutar. En el caso de los interactivos serán los usuarios los que seleccionen las tareas válidas que podrán ejecutar.

Alcance: ¿Cuánto de lo que se podría hacer nos va a dar tiempo a hacer con el tiempo y la gente que tenemos?

- Hemos calculado que nos dará tiempo a realizar la mayor parte de los requisitos del proyecto con todas sus posibilidades. A parte de eso podríamos introducir la posibilidad de cargar diferentes plataformas de visualización, para facilitar así el estudio del modelo sociológico, ya que ésta se adaptaría al domino del ejemplo. En caso de tener completado todo lo anterior nos centraríamos en realizar diferentes archivos de prueba utilizando las interfaces para metamodelos y visualización, además de hacer una interfaz de usuario más vistosa.

Viabilidad: ¿Es realizable nuestro proyecto?

- Después de realizar un estudio sobre lo que vamos a realizar, las tecnologías que vamos a usar, el tiempo que tenemos, y los conocimientos de nuestros componentes, creemos que nuestro proyecto es bastante viable.

4.1.2 Casos de uso

Pulsar Simular (Cu 1)

Objetivo	Se inicia la simulación del modelo. En cada paso se mostrarán las instancias que existen y se reflejará, a través de colores, qué entidades persisten (porque todas sus instancias persisten), cuáles ya no existen y cuáles han ejecutado un cambio en el estado del mundo.
Prerrequisitos	Haber cargado un fichero de especificación del modelo y otro de visualización en la aplicación, a través de los botones existentes para tal efecto o a través del menú 'File'.
Actores	Usuario
Acciones	1. Hacer clic izquierdo con el ratón en el botón de reproducir.
Comentarios	Existen dos maneras de reproducir: paso a paso o cada ciertos segundos, especificados por el usuario a través del menú de opciones. Si no se indica el tiempo, por defecto está configurado para que se ejecute cada 4 segundos.

Pulsar Parar (Cu 2)

Objetivo	Se para la simulación de la especificación cargada.
Prerrequisitos	Haber iniciado la simulación.
Actores	Usuario
Acciones	1. Hacer clic izquierdo con el ratón en el botón de parar.
Comentarios	Una vez parada la simulación, al pulsar de nuevo en reproducir, se ejecutará la simulación de la especificación del modelo desde el principio

Pulsar Pausar (Cu 3)

Objetivo	Se pausa la simulación del modelo cargado.
Prerrequisitos	Haber iniciado la simulación a través del botón de ejecutar cada cierto tiempo.
Actores	Usuario
Acciones	1. Hacer clic izquierdo con el ratón en el botón de pausar.
Comentarios	Si se está ejecutando paso a paso, el botón de pausar está inhabilitado. Al reanudar la simulación (con cualquiera de los dos botones de simular), ésta continúa desde donde se había dejado en suspenso.

Salvar la visualización en una imagen (Cu 4)

Objetivo	Guardar la visualización actual en una imagen jpg, png o gif
Prerrequisitos	
Actores	Usuario
Acciones	1. Hacer clic en la opción de salvar imagen del menú ‘File’
Comentarios	Esta opción siempre está disponible (no es necesario haber cargado un modelo en la aplicación), pero su utilidad se basa fundamentalmente en que el canvas tenga el modelo cargado. Si no es así, el usuario obtendrá una imagen en blanco.

Cargar Fichero de Especificación del Modelo (Cu 5)

Objetivo	Cargar el fichero de especificación de modelo en la aplicación para poder proceder a su simulación.
Prerrequisitos	Haber creado un fichero .specification a través del editor de árbol de Eclipse.
Actores	Usuario
Acciones	1. Hacer clic izquierdo con el ratón en el botón de cargar fichero de especificación de la barra de herramientas o en la opción del menú 'File'. 2. Seleccionar el fichero .specification a través del diálogo que aparece.
Comentarios	Los posibles errores que contenga el fichero de especificación no se notificarán hasta que se pulse alguno de los botones de 'Play'. Dichos errores se reportarán en la barra de estado situada en la parte inferior de la aplicación.

Pasar especificación al Motor (Cu 6)

Objetivo	Se lleva a cabo primeramente el preprocesamiento del fichero para encontrar posibles inconsistencias en la especificación. En esta primera etapa se construye asimismo las estructuras que representan al modelo y a las instancias dentro de la aplicación.
Prerrequisitos	Haber cargado un fichero de especificación y de visualización y haber pulsado cualquiera de los botones de ejecución.
Actores	El Motor
Acciones	1. Se lleva a cabo el preprocesamiento. 1.1. Se carga en memoria el fichero de especificación. 1.2. Se construyen las estructuras que representan el modelo en la especificación y se notifican posibles errores según se *parsea* el documento. 2. Cada vez que el motor, a través del Gestor de Mundo, recibe la orden de ejecutar, realiza un paso de ejecución sobre el estado actual del mundo.
Comentarios	

Pasar estados al Gestor de Mundo (Cu 7)

Objetivo	El motor pasa el estado que se tiene que visualizar en ese momento. En el motor se almacena tanto el estado del mundo como las entidades que han producido un cambio en él.
Prerrequisitos	Que la aplicación esté ejecutando la simulación y que se haya ejecutado un paso en el motor gracias a la señal recibida desde la GUI.
Actores	Gestor de Mundo
Acciones	1. Obtener del motor el estado del mundo (instancias) y la lista de entidades que han producido un cambio en él.
Comentarios	

Pasar modelo de visualización al Wrapper (Cu 8)

Objetivo	El Wrapper obtendrá la representación visual del modelo que se cargue a través del fichero de visualización.
Prerrequisitos	Haber creado con el editor de árbol un modelo de visualización y haber pulsado cualquiera de los dos botones de ejecución.
Actores	Wrapper
Acciones	1. Se carga en memoria el fichero de visualización 2. Se realiza un *parseo* del fichero para cargar en una tabla la correspondencia entre los entes representables y su representación. Dichos entes serán las entidades, las relaciones y los 'association ends'.
Comentarios	

Pasar orden al Wrapper (Cu 9)

Objetivo	El gestor de mundo le pasará los objetos que se deben representar en ese momento al wrapper. Dichos objetos serán las entidades del modelo que, dependiendo del estado del mundo, se representarán de un color o de otro.
Prerrequisitos	Haber analizado el estado en el que se encuentra la representación, y adjudicado la representación que va a tener cada elemento de ese estado.
Actores	El Gestor de Mundo
Acciones	1. El gestor envía una orden con los objetos que han de ser representados.
Comentarios	

Recibir objetos a pintar (Cu 10)

Objetivo	El wrapper recibirá lo que tiene que verse por pantalla y ordenará a la librería de visualización que lo dibuje.
Prerrequisitos	Haber enviado la representación desde el gestor de mundo.
Actores	El Wrapper
Acciones	1. Almacenar lo que debe mostrarse. 2. Ordenar a la librería que lo pinte.
Comentarios	

Pruebas (Cu 11)

Se incluyen las pruebas realizadas con más detalle y capturas en la sección 5.3, más adelante.

Objetivo	Simular un fichero de especificación en la aplicación.
Prerrequisitos	Haber realizado un fichero de especificación y de visualización a través del editor en árbol de eclipse.
Actores	Usuario
Acciones	1. Ejecutar la aplicación. 2. Cargar fichero de especificación del modelo (Cu 5). 3. Cargar fichero de visualización. 4. Pulsar botón de ejecutar (Cu 1). 5.Opciones durante la simulación: a) Parar la simulación (Cu2). Al volver a ejecutar se reiniciará la simulación desde el comienzo. b) Pausar la simulación (Cu3), en el caso de que se esté ejecutando de manera automática c) Reanudar la simulación a través de cualquiera de las dos opciones de ejecución (Cu 1), siempre que la simulación se haya parado o pausado. 6. Al acabar la simulación se puede volver a pulsar ejecutar (Cu 1), con lo que se comenzaría desde el principio de nuevo o cargar otro fichero de especificación.
Comentarios	

4.1.3. Tecnologías

Las herramientas con las que se desarrollará el proyecto serán las siguientes:

Entorno de programación: Eclipse (http://www.eclipse.org).

Los proyectos JAVA que componen la aplicación han sido desarrollados en el entorno Eclipse. Esta elección de IDE sobre otros disponibles, ha sido principalmente por su facilidad a la hora de escribir código y por los plugins disponibles, EMF y ECore, que han sido imprescindibles en el desarrollo de este proyecto.

EMF es un framework que además genera código, utilizado para crear herramientas y aplicaciones basadas en modelos de datos estructurados, como es nuestro caso. La forma en que se especifican los modelos es a través de XMI (XML Metadata Interchange), que en resumen se reduce a ficheros XML con características especiales. A partir de la definición de dichos modelos, el framework permite generar código para modelar en forma de clases dicha definición, así como un sencillo editor – integrado como aplicación Eclipse – de especificaciones basadas en los modelos. EMF, además, permite definir los modelos de maneras más intuitivas para el usuario – como por ejemplo el editor gráfico que se incluye – que no es sino una forma de enmascarar el proceso tedioso de definir el fichero XMI mencionado anteriormente.

Lenguaje de programación: El lenguaje de programación base seleccionado ha sido JAVA.

Visualización básica: JGraph (http://www.jgraph.com)

A través de esta librería implementamos la visualización básica como diagrama UML-AT. JGraph es una librería *open source* que permite visualizar grafos dirigidos y que se integra de manera muy cómoda con Swing de JAVA.

Esta librería proporciona una manera rápida y sencilla de obtener una representación gráfica de modelos entendidos como grafos, como ocurre en nuestro caso. Utiliza el modelo Vista-Controlador y ofrece una representación para cada uno de los elementos que integran el grafo: los nodos, las aristas y el propio grafo. Además, añadir características especiales a dichos elementos – colores y formas de los nodos, aristas de distintas formas, etc. – se convierte en una tarea fácil gracias al sistema ideado para introducir cambios a través de tablas. Por otra parte, también ofrece la posibilidad al desarrollador de crear sus propias representaciones de nodos y aristas a través de un sistema jerarquizado de clases. La integración con Swing es inmediata, pues la clase que representa al grafo en su totalidad es parte de la jerarquía de Swing al extender la clase JAVA *JComponent*.

4.1.4. Riesgos

4.1.4.1. Análisis de riesgos

En las siguientes tablas mostramos los riesgos considerados y los planes para su tratamiento:

Riesgo	Probabilidad	Impacto
Incompatibilidad de horarios entre los miembros del grupo	Alta	Tolerable
Mala estimación en el tiempo necesario para completar las características fundamentales del proyecto	Alta	Serio
Falta de tiempo para realizar características secundarias	Alta	Tolerable
No poder realizar el proyecto en los puestos de laboratorio	Muy Alta	Tolerable
El personal no es capaz de adquirir los conocimientos	Moderada	Serio
La aplicación es demasiado lenta como para ser usable	Moderada	Serio
La aplicación es excesivamente grande	Moderada	Serio
Bajas permanentes del personal (por bajas médicas, abandono de la asignatura, etc.)	Baja	Serio

4.1.4.2. Estrategia de gestión de riesgos

Incompatibilidad de horarios entre los miembros del grupo

Probabilidad	Alta
Impacto	Tolerable
Descripción	Problemas a la hora de quedar para avanzar en el proyecto juntar partes y asistir a reuniones.
Consecuencias	El proyecto no avanza.
Cómo evitarlo	Intentar juntar los horarios de todos y quedar en algún momento libre para todos.
Cómo tratarlo	Utilizar el correo, y ceder cada uno, lo más posible para llegar a un acuerdo
Eliminado	Iteración Octubre.

Mala estimación en el tiempo necesario para completar las características fundamentales del proyecto

Probabilidad	Alta
Impacto	Serio
Descripción	Tardamos más en hacer lo propuesto de lo que estimamos.
Consecuencias	El proyecto no avanza, no acabamos a tiempo.
Cómo evitarlo	Enterarnos bien de lo que hay que hacer y estudiar la dificultad.
Cómo tratarlo	Dedicarle más tiempo.
Eliminado	Iteración de Mayo.

Falta de tiempo para realizar características secundarias

Probabilidad	Alta
Impacto	Tolerable
Descripción	Tardamos más en hacer lo propuesto de lo que estimamos y no nos queda tiempo para perfeccionar y añadir elementos más visuales por ejemplo.
Consecuencias	El proyecto queda menos vistoso.
Cómo evitarlo	Estimar bien el tiempo.
Cómo tratarlo	Dedicarle más tiempo.
Eliminado	Surgió a principios de mayo, se suplieron las carencias con una preparación y Modularidad para futuras ampliaciones, así como en refinar lo hecho anteriormente

No poder realizar el proyecto en los puestos de laboratorio

Probabilidad	Alta
Impacto	Tolerable
Descripción	No tenemos en los laboratorios instalados los programas, o plug-ins, necesarios para realizar nuestra aplicación
Consecuencias	No podemos trabajar en la facultad, que es cuando estamos los tres juntos.
Cómo evitarlo	Intentar pedir que lo instalen, aunque poco probable.
Cómo tratarlo	Trabajar en la facultad con nuestros propios ordenadores portátiles.
Eliminado	Iteración de Octubre

El personal no es capaz de adquirir los conocimientos

Probabilidad	Moderada
Impacto	Serio
Descripción	No conseguimos entender las tecnologías nuevas.
Consecuencias	No podemos realizar el proyecto como estaba previsto.
Cómo evitarlo	Pedir ayuda, ayudarnos entre nosotros, que el profesor nos facilite manuales, hacer más tutoriales.
Cómo tratarlo	Pedir más ayuda, intentar comprenderlo por otras fuentes.
Eliminado	Iteración de Abril. Problemas con la visualización, que finalmente se resuelven.

La aplicación es demasiado lenta como para ser usable

Probabilidad	Moderada
Impacto	Serio
Descripción	Cuando finalizamos la aplicación esta va visualmente muy lenta y se hace demasiado pesada para ser usable.
Consecuencias	El proyecto no sirve para lo ideado.
Cómo evitarlo	Hacer prototipos y pruebas.
Cómo tratarlo	Investigar y solucionar.
Eliminado	Iteración de Abril. Se comprueba y no surge.

La aplicación es excesivamente grande

Probabilidad	Moderada
Impacto	Serio
Descripción	Lo propuesto para hacer es demasiado grande y extenso para que de tiempo.
Consecuencias	El proyecto no se finaliza por completo, hay que quitar funcionalidad.
Cómo evitarlo	Hacer prototipos, buen diseño y buena planificación.
Cómo tratarlo	Intentar reducir funcionalidad sin que afecte a lo esperado por la aplicación.
Eliminado	No surgió

Bajas permanentes del personal (por bajas médicas, abandono de la asignatura, etc.)

Probabilidad	Baja
Impacto	Serio
Descripción	Dejamos de ser tres en el proyecto.
Consecuencias	Al ser menos no nos da tiempo a acabar.
Cómo evitarlo	Procurar no estar malos, motivación para no abandonar.
Cómo tratarlo	Convencerlo para que regrese.
Eliminado	No ha surgido.

4.1.5. Gestión de la configuración

Todos los archivos, tanto de documentación como de código, son objeto de control de la gestión de configuración.

Para la nomenclatura de los documentos y clases se elegirán nombres claramente identificativos y descriptivos de la información que contienen. En cuanto a los archivos de documentación se incluirá la versión en el archivo .zip generado cada vez que se produzca un cambio.

Para controlar los cambios y no perder las versiones anteriores se guardarán siempre las versiones de todos los archivos tanto de documentación como de código hasta el final de la iteración en curso hasta establecer un documento final que será la versión definitiva de ese documento para esa iteración.

Dado el reducido número de integrantes del grupo y la facilidad de coordinación en que se traduce, sumado a que proporciona una gran facilidad de revisión de cualquier información en todo momento, todos los miembros deberán tener una copia de los archivos. Así en el momento en el que un documento sea modificado se enviará por correo a los otros dos integrantes. De este modo todos tendrán una versión actualizada de los documentos en todo momento.

Para realizar cambios sobre un archivo que tengan que modificar varias personas antes de alcanzar su aspecto final se deberá establecer un turno de modificación para no provocar pérdidas e inconsistencias de la información. De esta manera será necesario avisar a las personas correspondientes antes de modificar el archivo, y mientras no se reciba un nuevo aviso de que el archivo vuelve a estar disponible (adjuntando la nueva versión del archivo) no podrá ser modificado por nadie más.

Estándares de código

Para los archivos de código no será necesario incluir en el nombre la versión ya que como se ha explicado anteriormente cada vez que se realice algún cambio en el proyecto, se realizará un archivo .zip indicando la versión y fecha de última modificación (en formato dd/mm/hh), e inmediatamente se enviará por correo esta modificación al resto de integrantes ya que hemos considerado que somos un número reducido, no nos hace falta utilizar ninguna herramienta de control de versiones.

Cada clase debe tener un nombre claramente explicativo y representativo, así como todas las variables y métodos que se usen para el desarrollo del código. El nombre de las clases deberá comenzar siempre con mayúscula, y el de las variables y métodos con minúscula, el nombre de los atributos de clase será también en minúscula y estará precedido por un "_". Aún así se deberá incluir antes de cada uno de estos cuatro elementos nombrados una breve explicación del propósito y función que desarrollan dentro del programa, cuando no sean éstas deducibles de su nombre. Esta explicación será siempre en forma de comentarios de código colocados una línea antes que la declaración del elemento que describen.

Estándar de documentación

El nombre que identificará a los documentos debe ser a la vez claramente representativo y descriptivo de la información que contiene. Cada nueva sección aparecerá en un nuevo comienzo de hoja.

En cuanto a la presentación del texto en el documento será la siguiente:

- Fuente para todo el documento y las cabeceras: Times New Roman.
- Color de letra para todo el documento y las cabeceras: negra.
- Formato texto normal: tamaño 12, alineado a la izquierda.
- Formato títulos principales: tamaño 22, negrita, alineado a la izquierda.
- Formato títulos secundarios: tamaño 18, negrita, alineado a la izquierda.
- Formato títulos secundarios: tamaño 12, negrita, subrayado, alineado a la izquierda.
- Tabulación inicial en la primera línea de cada párrafo.
- Interlineado de 1 línea.
- Numeración de página en la parte inferior central.
- Encabezado con el texto “Sistemas Informáticos 2008/2009” en la parte superior izquierda y el nombre del documento en la parte superior derecha, ambos con letra Times New Roman, color negro, tamaño 12.
- Para palabras en inglés se utiliza la cursiva.
- Para las listas de enumeraciones se usará la herramienta del editor de texto correspondiente (Microsoft Word)

4.1.6 Requisitos del sistema

Usuario

Esta aplicación no requiere una configuración especial para el sistema ni unos requisitos especiales dado que se proporciona un archivo autoejecutable con el que se puede usar la herramienta sin ningún tipo de problema.

El único requisito exigido es tener instalada la máquina virtual de Java (JVM) dado que el propio archivo comprimido de Java la necesita para poder ejecutarse. Para poder instalar la JVM es necesario bajarse de la página de Sun http://www.java.com/es/download/manual.jsp) el instalador correspondiente a alguna de las versiones de Java Runtime Environment (JRE) iguales o posteriores a la versión 1.5.

Si el usuario pretende crear un fichero de especificación de un modelo propio, debe tener instalado Eclipse y ejecutar el proyecto metalanguage.editor, que forma parte de nuestra aplicación, como aplicación Eclipse. Después deberá crearse un nuevo proyecto general y crear dentro de él ficheros *.specification*. Para conocer con más detalle la forma de generar una especificación, ver la sección de manual del usuario en el apéndice final.

Desarrollador

Existen dos tipos de posibles desarrolladores para este proyecto:

- Extensiones del lenguaje UML-AT:
  Tener Eclipse Modeling Framework, no es necesario instalarlo. Modificar los ficheros *.ecore* y *.ecorediag*, dependiendo de si se desea hacerlo a mano en el XML (fichero ecore) o si se desea realizarlo mediante el editor gráfico (fichero ecorediag). Las modificaciones realizadas en el fichero .ecorediag tienen efecto en el fichero .ecore, pero no al contrario.

- Extensiones de la aplicación:
  Tener instalado cualquier IDE que soporte JAVA. Si el desarrollador pretende crear una visualización distinta, deberá extender la clase Wrapper (ver arquitectura y diseño) y adecuar cada unas de las funciones a la visualización concreta.

4.2. Planificación

En cuanto al desarrollo del trabajo lo dividiremos por iteraciones de duración variable dependiendo de los objetivos asignados a cada una de ellas. En general todas durarán entre 3 y 4 semanas e irán marcadas por una fecha de comienzo y finalización, para la cual deben estar terminados y conseguidos los objetivos fijados.

Hemos contado con periodos en los que no podemos trabajar tan a fondo en el proyecto debido a exámenes, periodos festivos… Así partimos del l de octubre de 2008 hasta el 31 de mayo de 2009, y contamos con los periodos festivos como media semana de trabajo por una semana completa de 7 días. En cuanto a las fechas de exámenes hemos decidido excluirlas de la planificación por la falta de tiempo.

4.2.1. Plan de fase.

Fecha de los hitos importantes:

Objetivo del ciclo (fase de inicio): 1 Octubre 2008
Arquitectura del ciclo (final de la fase de elaboración, arquitectura completa, prototipo básico): finales de febrero
Capacidad operativa (final de la fase de construcción): finales de semana santa
Fase de transición (arreglos, y memoria): mediados de mayo.
Entrega del producto: finales de mayo.

- Recursos humanos:
  3 personas (integrantes del grupo)
- Testeadores (voluntarios externos al grupo de desarrollo)

4.2.2. Plan de Iteración.

Nuestras iteraciones previstas son:

- **Primera iteración** (Iteración de Octubre)
  - Objetivos: Requisitos, riesgos, especificación, casos de uso, estudio de tecnologías.
  - Periodo: 1 octubre - 25 octubre
  - Estado: terminada
  - Evaluación: completado. Se entiende qué va a hacer nuestra aplicación

- **Segunda iteración** (Iteración de Octubre- Noviembre)
  - Objetivos: Definición del lenguaje UML-AT, generación del archivo ECore y utilización del EMF para generar el Editor de Árbol y las Clases JAVA que se utilizarán para crear ejemplos de simulación en la aplicación. Utilizar el editor para generar un ejemplo básico de posible simulación en el prototipo inicial. Realizar estos mismos pasos pero con el metamodelo de visualización.
  - Periodo: 26 octubre - 12 noviembre
  - Estado: terminada
  - Evaluación: completado. Se tiene un lenguaje de metamodelo sólido del lenguaje UML-AT. El metamodelo de la visualización queda de pendientes revisiones, según se vaya a realizar finalmente dicha visualización.

- **Tercera iteración** (Iteración de Noviembre- Diciembre)
  - Objetivos: Diseño del Motor y Gestor de Mundo. Diseño de la arquitectura y funcionalidad básica que tendrán estos paquetes. Esto incluye: diseño de las clases básicas que tendrán, el motor recibirá como entrada las clases java generadas de ejemplo en la iteración anterior, las identificará y organizará una máquina de estados, a través de ella le irá administrando al gestor de mundo la situación en la que está el mundo actual separados por cuantos de tiempo. El gestor de mundo recibirá inicialmente una única situación para visualizar (recibida del motor), a su vez recibirá también como entrada cómo representar cada situación (por medio de las clases java que se crearon de ejemplo en la iteración anterior usando el editor de árbol dedicado a la visualización). Como salida producirá instrucciones para el Wrapper.
  - Periodo: 13 noviembre – 31 diciembre
  - Estado: terminada
  - Evaluación: se tiene una arquitectura sólida.

- **Cuarta iteración** (Iteración de Enero – Febrero)
  - Objetivos: Preparar la fase de construcción del Wrapper, crear un prototipo, que permitirá a través de las indicaciones del gestor de mundo crear instrucciones de creación de esos objetos.
  - Periodo: 1 enero - 22 febrero
  - Estado: terminada
  - Evaluación: los módulos están bien comunicados entre si, por lo que el Wrapper recibe adecuadamente toda la información necesaria.

- **Quinta iteración** (Iteración de Febrero – Marzo)
  - Objetivos: Prototipo de la GUI que utilizará el usuario, permitirá cargar un ejemplo ya creado, pulsar play y pararlo. Revisión de errores y problemas de diseño. Planificar la construcción en detalle.
  - Periodo: 23 febrero - 13 marzo
  - Estado: terminada
  - Evaluación: la estética de este prototipo será sencilla, sólo para cubrir funcionalidad.

- **Sexta iteración** (Iteración de Marzo a Abril)
  - Objetivos: Estudio del motor gráfico, utilización y conexión a la aplicación.
  - Periodo: 14 marzo - 14 abril
  - Estado: terminada
  - Evaluación: Se consigue visualizar un grafo del diagrama UML-AT realizado.

- **Séptima iteración** (Iteración de Abril - Mayo)
  - Objetivos: Fase de construcción y ampliación de lo hecho en las iteraciones 2-6 dando funcionalidad completa.
  - Periodo: 15 abril – 10 mayo
  - Estado: terminada
  - Evaluación: se permite representar varias situaciones simultáneamente.

- **Octava iteración** (Iteración de Mayo)
  - Objetivos: Realizar la fase de transición. Probar el proyecto y corregir errores. Ampliar algún detalle. Realizar la memoria.
  - Periodo: 11 mayo – 31 mayo
  - Estado: terminada
  - Evaluación: finaliza la parte básica del proyecto.

5. Arquitectura

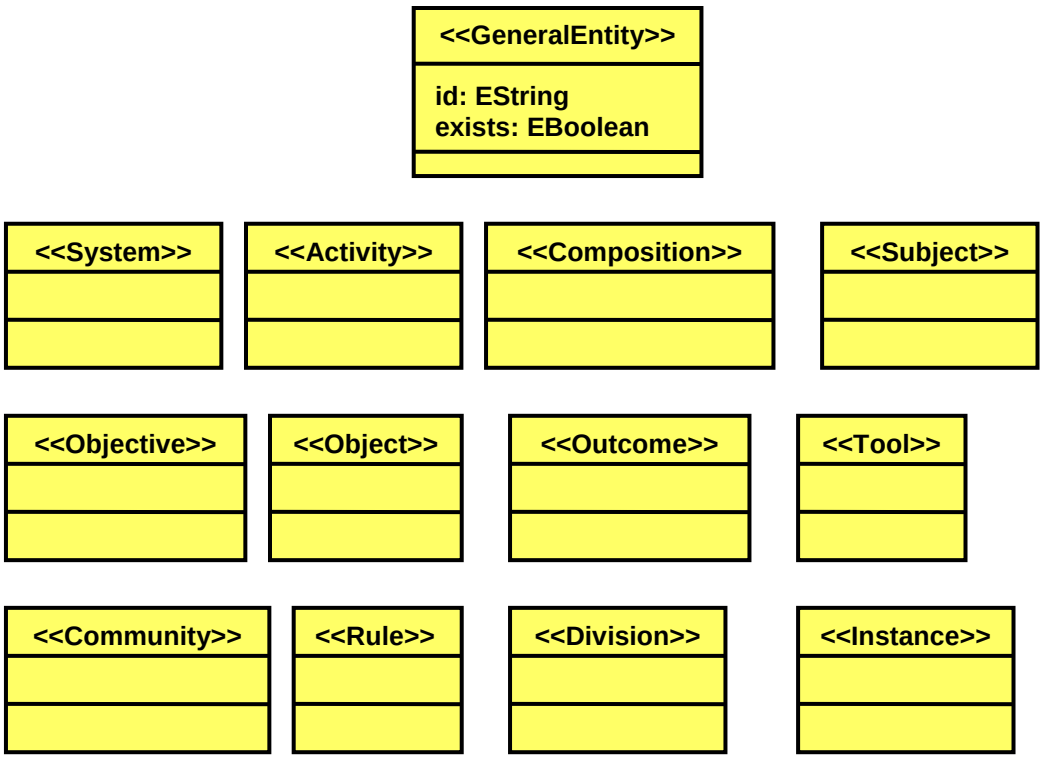
5.1. Estructura y diseño

La aplicación implementada presenta una funcionalidad básica, la cual se puede resumir en el preprocesamiento de los ficheros de las especificaciones y la simulación por parte del motor. Además, nuestra aplicación permite la visualización de dicha simulación con diversas plataformas gráficas, gracias a la estructura de clases realizada, la cual soporta el acoplamiento con distintas tecnologías.

5.1.1. Definición del metamodelo

El primer paso, realizado gracias al Eclipse Modeling Framework (EMF), consistió en la especificación del metalenguaje de UML-AT (ver página 20). Definimos 4 EPackages:

- Entities: contiene las entidades que conforman el lenguaje. Cada una de ellas está modelada como una EClass. Todas heredan de una EClass llamada 'GeneralEntity' que contiene dos EAttributes: *id* (el nombre de la entidad) y *exists* (que indica si la entidad existe o no). Las entidades que hemos modelado, y que pertenecen al lenguaje de UML-AT son las siguientes:



- Relations: contiene las relaciones definidas en UML-AT como EClasses. Hemos establecido la siguiente jerarquía: todas las EClasses de relaciones derivan de una abstracta llamada GeneralRelation; las relaciones cuyo source es una entidad System derivan de otra llamada NormalRelation, que a su vez deriva de GeneralRelation; las relaciones cuyo source es una entidad Rule derivan de otra llamada RuleRelation, que a su vez deriva de GeneralRelation. El resto, si no derivan de alguna de esas dos mencionadas de manera especial, derivan de GeneralRelation. El hecho de modelar esas EClasses intermedias tiene un significado inmediato a la hora de modelar los association end. De esta manera, no necesitamos association ends redundantes. Esto lo veremos en el siguiente punto. La EClass GeneralRelation contiene el EAttribute *stereotype*, para estereotipar cada relación. La EClass Contribution, tiene además otro EAttribute adicional, *contribution* de tipo EnumContribution, definido por nosotros también como una EEnum y que, en principio, tiene los valores *guarantee*, *essential*, *positively*, *negatively*, *undefined* e *impide*. La EClass DivisionOfLabour, por su parte, también contiene otro EAttribute, *relation*, el cual es de tipo EnumRelation, definido por nosotros también como EEnum y que sólo contiene el valor *superior*.

- Association_end: este EPackage contiene los association ends correspondientes a los sources y targets de todas las relaciones. Al igual que en los anteriores EPackages, tenemos una EClass abstracta de la que derivan el resto: GeneralAssociationEnd, que contiene el EAttribute *multiplicity*, de tipo Eint. Como se explicaba en el punto anterior, al haber modelado las relaciones que tenían algo en común bajo una misma EClass a partir de la cual derivaran todas, podemos asociar mediante una EReference la clase abstracta de la que derivan con una EClass que modele el association end común para todas. Por ejemplo. En el caso de las NormalRelation, todas comparten que tienen un System como source, por lo que hemos definido una EClass SystemSourceAssociationEnd con la cual se relaciona la EClass NormalRelation mediante una EReference llamada source. Esto nos ahorra tener que crear una EClass por cada source de cada relación que tenga un System en esa posición. De igual forma hemos procedido con las relaciones que tienen una entidad Rule en su source.
  Por último, las relaciones para las que el lenguaje define que tienen un *artefacto* en su *source* o en su *target*, las hemos modelado en sus association end con dos EClasses: SourceAssociationEnd y TargetAssociationEnd. Las EClasses referentes a las relaciones que se encuentren en el supuesto anterior, tendrán una EReference hasta uno de estas EClasses del paquete Association_end.
  De las EClasses que modelan association ends parten EReferences hasta las EClasses que modelan las entidades. Cada association end se dirige a su entidad correspondiente.

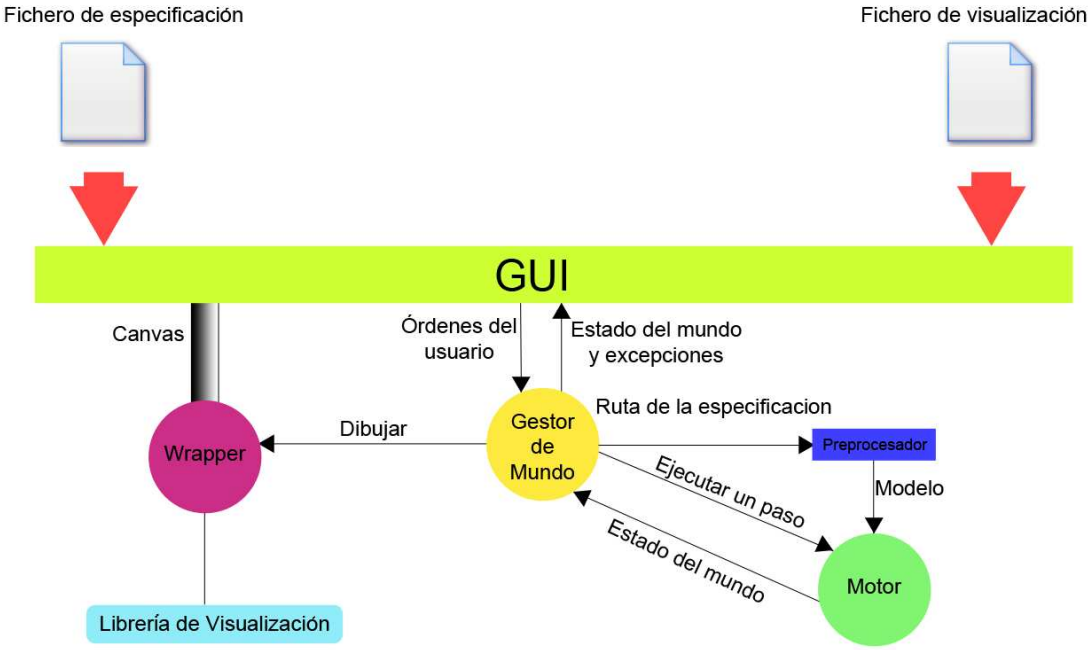
- Specification: por último, este EPackage contiene una EClass llamada Specification que modela la especificación de un modelo y que contiene dos EReferences hasta la GeneralEntity y la GeneralRelation respectivamente. Cada una de estas EReferences tiene cardinalidad 0..*, con lo que indicamos que una especificación puede tener un número arbitrario tanto de relaciones como de entidades.

Mediante la siguiente tabla vamos a describir las relaciones del lenguaje UML-AT de manera resumida. En primer lugar se encuentra la fuente (*source*) de cada relación, el nombre de esta en el medio y por último el objetivo (*target*). En el caso de relaciones que no son binarias (en concreto, *divisionOfLabour* y *surpass*) se han especificado todos sus *target*. Y ya por último, recordar que el empleo del término "Artefacto" se refiere a cualquier entidad del lenguaje:

SOURCE	**RELACIÓN**	**TARGET**
System	activitySystem	Activity
	executedBy	Subject
	try	Objective
	transform	Object
	use	Tool
	produce	Outcome
	consume	Artefacto
	include	Artefacto
	accomplishedBy	Community
	ruledBy	Rule
	organizedBy	Division
Rule	ruleContext	Artefacto
	rulePositive	Artefacto
	ruleNegative	Artefacto
Composition	input	Object
	output	Outcome
Division	divisionOfLabour	Subject | Community
Subject	purgue	Objective
	surpass	Objective | Objective
Artefacto	instanceOf	Instance
	contribute	Artefacto

5.1.2. Partes de la aplicación

En el siguiente gráfico se muestran las distintas partes de la aplicación y la interacción que se produce entre ellas:



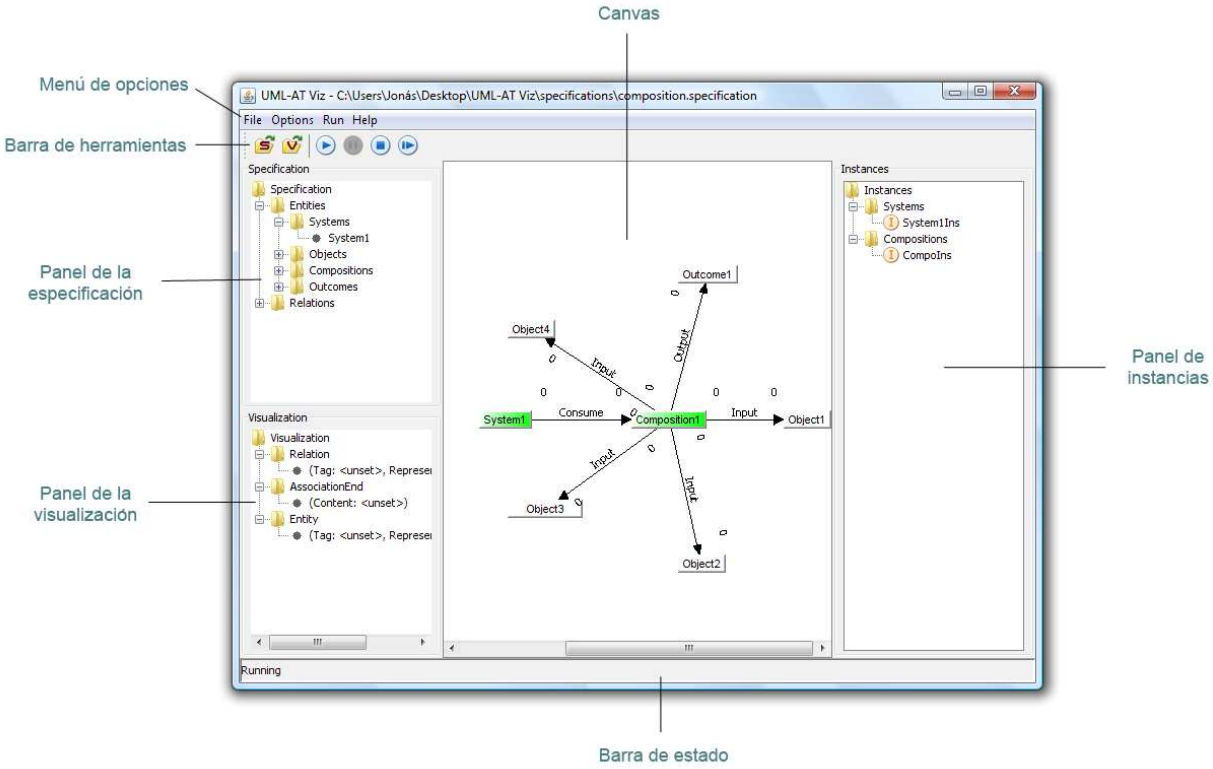
A lo largo de toda esta sección, se explicarán cada unas de las partes, cuáles son sus principales puntos a destacar y la tarea que realizan dentro de la aplicación. Este esquema, no obstante sirve como representación para entender la relación existente entre los diversos módulos en el caso de que la explicación textual resulte confusa.

Pasando ya a la implementación, nuestra aplicación se divide en 4 partes bien diferenciadas:

5.1.2.1. La GUI

La parte más externa de la aplicación y que el usuario puede visualizar en primera instancia es su interfaz gráfica. Consta de las siguientes partes:

En el panel de la especificación, a la izquierda, el usuario podrá tener acceso en todo momento de la simulación al modelo especificado en el fichero cargado. De igual forma, podrá informarse del estado del mundo (recordemos, las instancias que existen en un determinado paso de ejecución) mediante el panel de instancias situado a la derecha. El centro de la ventana está reservado a la visualización propiamente dicha y la identificaremos como el *canvas*. En la parte de abajo se encuentra la barra de *status*, en la que aparecerán los mensajes pertinentes, tanto de error como de confirmación del estado actual.



A través de la GUI el usuario podrá cargar un fichero de especificación de un modelo concreto y cargar el fichero de visualización a través de los botones de la barra de herramientas o de los menús. Asimismo, una vez cargados dichos ficheros, el usuario tiene control sobre la ejecución mediante los botones de la barra de herramientas o a través del menú 'Run'. Dichos botones tienen la siguiente funcionalidad:

- PLAY: mediante este botón se ejecuta la simulación de manera automática. Un paso de ejecución se realizará cada ciertos segundos, especificados mediante la opción correspondiente del menú 'Options'. Por defecto, un paso de simulación ocurre cada 4 segundos.
- PLAY STEP BY STEP: como su nombre indica, ejecuta la simulación paso a paso. En este caso es el usuario quien debe ir pulsando en el botón para que la simulación avance.
- STOP: para la simulación. Al volver a pulsar cualquiera de los dos botones de play, la simulación comenzará desde el inicio.
- PAUSE: pausa la simulación. Cuando se pulsa alguno de los botones de play, la simulación se reanuda donde se había detenido.

En cuanto al propio diseño de la GUI, esta clase contiene dos objetos importantes: el Wrapper y el Gestor de Mundo (ambos serán descritos más adelante). A través del Wrapper obtendrá la visualización y la 'empotrará' en su canvas. El Gestor de Mundo es creado cada vez que se carga una nueva especificación y se le encarga, a través de la GUI, de comunicar al Motor (ver siguiente punto) que realice la simulación, que la pare o que la pause. El Gestor de Mundo actúa como un hilo de ejecución independiente de

la GUI para que no haya interferencias entre el procesamiento de la especificación y la interacción del usuario con la GUI. El Wrapper es creado una sola vez y es la GUI la encargada de pasárselo al Gestor de Mundo cada vez que éste se crea para que pueda comunicarle los elementos que debe visualizar en cada momento.

5.1.2.2. El Motor

Es la parte dedicada exclusivamente a procesar la especificación del modelo que tiene como entrada el programa. Se encarga de ejecutar los cambios pertinentes en el estado del mundo, entendiendo por esto qué instancias existen en un determinado paso de ejecución. El motor actuará tras un pequeño preprocesamiento del fichero de entrada en el cual se vigilará la coherencia de la especificación. En concreto, se le podrá notificar al usuario lo siguiente:

- El fichero está vacío
- A la hora de especificar una relación, se le ha olvidado especificar su *source* o sus posibles *targets*.
- Ha declarado dos relaciones *ActivitySystem* con el mismo System (*source*) y distinto Activity (*target*). Esto es incoherente, ya que un System sólo puede tener una Activity.
- Ha declarado dos relaciones *Output* con la misma Composition (*source*) y distinto Outcome (*target*). Resulta incoherente también, por la misma razón que antes.
- Ha declarado una relación *InstanceOf* en la que el *source* es una instancia. Una Instance sólo puede ser instancia de otra entidad que no sea a su vez instancia.

Además, en este preprocesamiento se construirán distintos ArrayList en los que se ubicarán los elementos del modelo en forma de objetos. Concretamente, tendremos un ArrayList por cada una de las siguientes: entidades, relaciones, association ends, instancias e instancias que pueden cambiar el estado de mundo. El ArrayList de instancias merece mención especial, ya que se utilizan objetos de la clase ExtendedInstance en lugar de Instance sin más. Por tanto, al preprocesar las instancias creamos un objeto de la clase ExtendedInstance que albergará a la instancia y a la entidad de la cual es instancia, a la que hemos identificado como ‘padre’. El motivo de usar esa otra clase creada por nosotros y no Instance a secas es poder tener acceso al padre en cualquier momento. Esto resulta muy útil a la hora de hacer la ejecución con el motor, ya que no podemos olvidar que lo que realmente se “ejecuta” son las instancias y no el modelo como tal.

El ArrayList de instancias que pueden efectuar un cambio en el mundo también se ha modelado con un fin especial. En UML-AT las únicas relaciones que producen verdaderos cambios en el estado del mundo (entendiendo por esto, como se ha mencionado antes, las instancias que existen en un paso de ejecución) son Produce, Consume, Input y Output. Según la especificación del lenguaje, dichas relaciones son ejecutadas por entidades System o Composition. Por ello, ideamos una nueva clase, a la que llamamos Changer y que albergará las instancias de System o Composition junto con sus relaciones ‘cambiantes’, o sea sus Produce y Consume en el caso de los Systems y Output e Input en el de los Compositions. De esta manera, construimos en el preprocesador un ArrayList de Changer, de forma que al procesar el modelo con el

motor sólo nos fijemos en dichos 'changers', ya que el resto de instancias sabemos que no producirán cambios en el mundo.
Además de estos ArrayList se construirán dos estructuras más que servirán de apoyo a la visualización: un grafo dirigido, modelado mediante una matriz de adyacencia con ciertas peculiaridades, y un ArrayList que contiene árboles. Ambas estructuras se explicarán más adelante, en la parte concerniente al wrapper.

Un paso de ejecución en el motor de puede representar mediante el siguiente fragmento de pseudocódigo:

```
para cada elemento e del ArrayList de Changers
     relCambiantes ← e.extraerRelacionesCambiantes
     para cada elemento r de relCambiantes
          ejecutarCambio(r)
     fpara
fpara
```

Es decir, extraemos cada uno de los elementos del ArrayList de Changer construido en el preprocesador (recordemos que estos elementos pueden ser bien Compositions o bien Systems), accedemos al conjunto de sus relaciones y para cada una de sus relaciones cambiantes (que puede ser un número arbitrario) se ejecuta el cambio que proceda con dicha relación. En el caso de los Systems, su relación Produce hace que existan nuevas instancias en el mundo, que previamente podían no existir, es decir, las 'crea'. Por el contrario, sus relaciones Consume transforman el mundo haciendo que instancias que existían dejen de existir, o dicho de otro modo, las 'destruye'. En cuanto a las Compositions, sus relaciones Input crean la propia instancia de la *composition* a partir de la existencia de las instancias de los objetos especificados en cada Input. Una vez exista la instancia de la *composition*, ésta creará la instancia del Outcome especificado en su relación Output.

5.1.2.3. El Wrapper

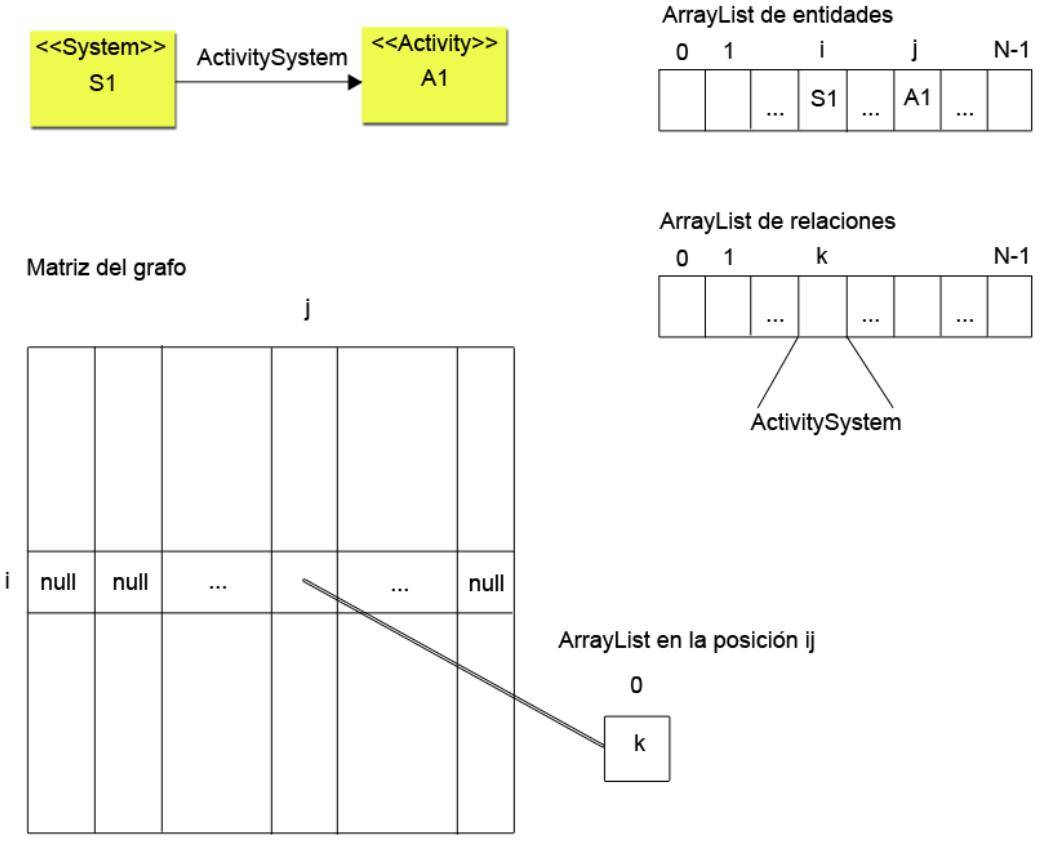
Esta clase es abstracta y es la encargada de ofrecer una funcionalidad básica de visualización. Su cometido es establecer un punto de acoplamiento entre la GUI de nuestra aplicación y cualquier plataforma o librería de visualización. En nuestras pruebas, hemos podido usar distintas librerías, desde OpenGL para JAVA (jogl), pasando por distintas librerías de grafos e incluso el motor gráfico 3D escrito en JAVA jMonkey. Esto permite que la aplicación esté abierta a cambiar la visualización de manera sencilla.

Para implementar una visualización concreta, tan sólo es necesario extender la clase Wrapper y definir sus métodos abstractos *getCanvas*, *resize* y *create*. El primero de los métodos sirve para establecer la unión entre la GUI y la librería o plataforma de visualización. En este método se le pedirá a dicha librería un componente de Swing (JComponent) para poder 'empotrarlo' en el canvas de la GUI. En general es una operación muy sencilla, ya que algunas librerías de visualización (como es el caso de jMonkey, o Grappa) permiten definir un canvas sobre el que se dibujará el contenido de la visualización. Dicho canvas puede ser devuelto a la GUI para establecer un camino a través del cual poder visualizar el modelo de la manera oportuna. En el caso de que la librería no contenga una forma específica de obtener un componente Swing, siempre se

puede definir uno de manera manual en el cual dibujar los componentes que nos ofrezca la librería y pasárselo a la GUI.

El segundo de los métodos se utiliza para implementar los cambios que sean oportunos en la visualización cuando se produzca un evento resize en la GUI. La GUI se encargará de invocar el método resize del Wrapper, que deberá ser implementado en la clase concreta que extienda del wrapper abstracto.

Por último, el método *create* es el que inicializa la visualización y crea todas las estructuras y objetos necesarios, insertándolos en el canvas que ha sido pasado a la GUI. Para favorecer la visualización, se pasan 4 ArrayList a través de esté método: el primero de ellos se trata del ArrayList de entidades del modelo; el segundo, el ArrayList de relaciones del modelo. Por si esto no es toda la información que la librería de visualización pueda necesitar, se pasan dos ArrayList adicionales: una matriz cuadrada de dimensiones *número_entidades* x *número_entidades* representando el modelo como un multigrafo, en la cual se indica si entre la entidad *i* y la *j* (contando que dichos índices son los del ArrayList de entidades pasado también como parámetro) hay una relación. Como entre dos entidades puede haber un número arbitrario de relaciones, en lugar de modelarla como una simple matriz de enteros, cada una de las posiciones alberga un ArrayList de enteros, de manera que si no hay relación entre la entidad i y la j, en la posición Grafo[i][j] el ArrayList será nulo. Si por el contrario existe una o más relaciones, el ArrayList será una lista de enteros en la que cada uno indicará la relación existente entre ambos. La manera en que se indicará es mediante el índice que tiene la relación dentro del ArrayList de relaciones pasado también como parámetro. Para despejar dudas acerca de esto, observemos la siguiente imagen:



Como vemos en la imagen, se supone que existen dos entidades S1 y A1 que están relacionadas. Cada una de ellas tiene índices i y j, respectivamente, dentro del ArrayList de entidades que es construido por el preprocesador del motor. Además, la relación existente entre ambos está indexada en el ArrayList de relaciones con índice k. Por ello, en la matriz del grafo se tendrá que en la posición Matriz[i][j] habrá un ArrayList de una posición que contendrá el índice 'k'. Si hubiese más relaciones ente esas dos entidades, dicho ArrayList sería mayor. Vemos además que entre la entidad i-ésima y la entidad 0, por ejemplo, no hay relaciones, con lo que el ArrayList entre ambas es nulo.

El siguiente ArrayList que se pasa como parámetro es un ArrayList de árboles. Se trata de otra forma de ver el modelo, entendiendo que es un multigrafo dirigido en forma de árbol – aunque puede contener ciclos, pero como idea abstracta y de manera general es bastante parecido a un árbol – que en principio puede tener varias componentes conexas. Cada uno de los elementos de ese ArrayList será cada una de las componentes conexas, comenzando por la raíz de ese pseudoárbol. La utilización de todas estas estructuras es opcional, pudiendo utilizarlas todas o sólo una parte de ellas, o incluso una combinación. El motivo de pasarlas todas como parámetros es el de facilitar la visualización.

Además de estos tres métodos obligatorios de definir, la clase abstracta define 2 métodos de carácter general para dibujar, también abstractos, con los que se puede indicar al elemento de visualización que realice un cambio en dicha visualización. Concretamente, los dos métodos tienen la siguiente cabecera:

```
public abstract void draw(String entity, Object obj);
public abstract void draw(int entity, Object obj);
```

La clase derivada de Wrapper implementará el método *dibujar* que más le convenga. Con el primero de los métodos podemos referirnos a una entidad por su nombre a la hora de realizar un cambio en su visualización. Con el segundo, podemos referirnos a la entidad mediante un entero. La utilización de uno u otro método dependerá de cómo se realice la visualización, no siendo necesario que se redefinan los dos, sino el que más convenga. El segundo parámetro de ambos es un objeto a través del cual especificamos el cambio producido en la visualización. De esta manera, podemos incluir cualquier tipo de información a la hora de visualizar la entidad especificada, como por ejemplo, cambiar el color de una entidad pasando un objeto de tipo Color, que será procesado en la plataforma de visualización de la manera conveniente.

5.1.2.4. El Gestor de Mundo

Por último, la clase Gestor de Mundo (WorldManager) se encarga de dirigir toda la aplicación. Por un lado, tiene el cometido de indicar al motor que ejecute un paso de ejecución y por otra parte, le indica al wrapper lo que tiene que dibujar tras un paso de ejecución. Típicamente, lo que le dirá que dibuje serán las instancias existentes o las entidades cuyas instancias han realizado un paso de ejecución. Además, el gestor de mundo tiene acceso al panel de instancias para poder actualizar el estado del mundo tras cada paso de ejecución. De igual forma, tiene acceso a la especificación a través del motor, siendo éste uno de sus atributos. El gestor es el encargado de inicializar el motor e indicarle la ruta del archivo que contiene la especificación y del cual deberá realizar un preprocesamiento, como se ha indicado en la sección correspondiente.

La GUI contendrá en el momento en que se cargue una especificación un nuevo objeto de la clase 'WorldManager'. Ésta está diseñada como un hilo de ejecución aparte, de manera que si el usuario, a través de la GUI, quiere detener, pausar o reanudar la ejecución, se traducirá en que dicho *thread* se detenga (*stop*), pause (*suspend*) o reanude (*resume*). La ejecución básica del método *run* del *thread* del gestor de mundo es la siguiente:

```
mientras cierto hacer
     si puedoEjecutar entonces
          motor.ejecutarPaso();
          actualizarVisualización();
          puedoEjecutar := falso;
     fsi
fmientras
```

Como vemos, se trata de un bucle infinito en el que se ejecuta un paso de simulación y se actualiza la visualización. El control de los pasos de simulación se realiza mediante la variable booleana *puedoEjecutar*, la cual será modificada a través de la GUI. Existen dos formas de modificar dicha variable: si la ejecución se realiza mediante el botón de play, un temporizador situado en la GUI realizará la tarea de cambiar la variable invocando el correspondiente método del gestor de mundo; si por el contrario se realiza ejecución paso a paso, se invocará el método del gestor encargado de cambiar el estado de dicha variable cada vez que se pulse el botón. Como se puede apreciar, una vez que el bucle infinito tiene 'permiso' para entrar dentro del *if*, se ejecuta el paso de simulación y de visualización y se vuelve a poner la variable de control a falso, esperando una nueva indicación de la GUI, que en última instancia, se refiere a una indicación del propio usuario. El motor, como ya hemos indicado, es un atributo del gestor.

5.1.3. Clases auxiliares

Para poder llevar a cabo toda la aplicación, se han creado una serie de clases a modo de utilidades y que sirven tanto para tareas de computación del modelo como de visualización. Son las siguientes:

5.1.3.1. ExtendedInstance

A la hora de procesar el modelo, el motor debe fijarse únicamente en las instancias definidas, pero cuando quiere averiguar de qué entidad es instancia una concreta, no hay forma de saberlo si no es agrupando a instancia y a la entidad dentro de una misma estructura. El motor, a la hora de ejecutar cambios sobre el mundo, recorre el ArrayList de instancias que le proporciona el preprocesador, pero no sabe si la instancia que va a mirar en cada momento es la adecuada o no. Por ejemplo, imaginemos que se quiere ejecutar una relación Consume entre una entidad System y una entidad Object. El motor recorrerá el ArrayList de instancias, pero no sabrá cuál es instancia de un Object. Por ello, hemos introducido esta clase, para aunar en una misma estructura a la instancia y a la entidad de la que lo es, a la que hemos llamado 'padre'. El motor, en lugar de procesar el ArrayList de instancias, lo hará con el de ExtendedInstance, otorgado también por el preprocesador. Los atributos de una ExtendedInstance son:

```
Instance _instance;
GeneralEntity _parent;
```

5.1.3.2. Changer

Esta clase se utiliza para poder representar una entidad que puede realizar cambios en el estado del mundo. Recordemos que las únicas entidades que pueden realizar un cambio en el mundo son System y Composition. La clase Changer contiene los siguientes atributos:

```
ExtendedInstance _changer;
ArrayList<GeneralRelation> _changerRel;
ArrayList<GeneralRelation> _descriptiveRel;
ArrayList<Instance> _arrayObject;
```

El primer atributo se corresponde con la instancia en sí. El hecho de utilizar la clase ExtendedInstance en lugar de la propia Instance es poder discernir si el *changer* es una Composition o un System.

El segundo atributo es el conjunto de todas las relaciones cambiantes para esa entidad, según se ha especificado en el modelo. El tercer atributo es el conjunto de relaciones que no cambian el mundo para esa entidad.

Por último, el cuarto atributo se utiliza sólo en caso de que el *changer* sea una Composition y se refiere a las instancias definidas para sus relaciones Input.
La construcción de estos tres ArrayLists es una tarea concerniente al preprocesador, por lo que al motor le resulta transparente saber cómo se han elaborado.

A la hora de procesar el modelo con el motor, se recorrerá un ArrayList de Changers, de manera que sólo se ejecuten los cambios por las instancias que pueden efectuarlos, extrayendo de cada *changer* sus relaciones cambiantes. De esta manera, nos ahorramos tiempo de procesamiento, pues ejecutamos directamente las relaciones cambiantes de las únicas instancias que pueden efectuar algún tipo de cambio en el mundo.

5.1.3.3. Graph

Esta clase se utiliza para modelar la matriz de adyacencia del multigrafo del modelo (ver Wrapper). Sus atributos son:

```
ArrayList[][] _graph;
int _numEntities;
int _numRelations;
```

El primer atributo se refiere a la matriz de adyacencias tal como se ha explicado en la sección del Wrapper. Los siguientes atributos se refieren al número de entidades (que coincide con las dimensiones de la matriz) y el número de relaciones. Éste último no se refiere al número de relaciones real, sino al número de relaciones registradas en la matriz. Esta pequeña diferencia se ilustra mediante las dos relaciones ternarias presentes en la definición del lenguaje (Surpass y DivisionOfLabour) en las que hay que

relacionar al *source* con dos *targets*, con lo que se contaría una relación más por cada una de ellas.

5.1.3.4. VizTree

Con esta clase se modela el árbol de visualización explicado en la sección del Wrapper. Sus atributos son:

```
int _numEntity;
boolean _isRoot;
int _numNodes;
ArrayList<VizTree> _children;
ArrayList<Integer> _relations;
```

El primer atributo se refiere al índice que tiene la entidad dentro del ArrayList de entidades generado por el preprocesador del motor. El segundo de los atributos informa de si el nodo del árbol es la raíz. El siguiente atributo se refiere al número de nodos que contiene el subárbol formado por ese nodo y sus hijos. El primero de los ArrayList que aparece como atributo es una lista de los hijos del nodo y, por último, el segundo ArrayList es una lista de enteros que nos indica que entre el nodo raíz y el hijo i-ésimo ocurre la relación de índice j, según el ArrayList de relaciones construido en el preprocesador del motor. Como es obvio, tanto el ArrayList de hijos como el de los índices de las relaciones tendrán el mismo tamaño.

5.1.3.5. Queue

Se trata de una cola de propósito general. En concreto, dentro de la aplicación, se utiliza para procesar los árboles de visualización por niveles.

5.2. UML

En esta parte se presentarán los diagramas de clase y de secuencia de los principales componentes y acciones de la arquitectura de la aplicación.

5.2.1 Diagramas de Clase

Los siguientes diagramas UML de clase corresponden a las principales clases que integran la aplicación. Las únicas clases que no se incluyen son las concernientes a la visualización básica que hemos implementado, ya que la aplicación es abierta a que la visualización sea objeto de cambio. Por ello, consideramos que es de mayor relevancia presentar la estructura de clases del núcleo del sistema.

5.2.1.1 Motor

Como se aprecia en el diagrama, el motor se divide en dos módulos: el motor propiamente dicho y el preprocesador, en el cual se hace un parseo inicial del fichero de especificación en busca de posibles fallos y con la finalidad de construir las estructuras que representen al modelo dentro de la aplicación.

De entre sus atributos, los llamados *_entities* y *_relations* se corresponden con las entidades y relaciones especificadas en el fichero y son creados por el preprocesador. El atributo *_world* se refiere a las entidades de las instancias que han provocado un cambio en el mundo y son las utilizadas tras cada paso de procesamiento para indicar a la visualización quién ha sido el responsable de los cambios producidos.

Pasemos a describir los principales métodos:

- Motor: además de las evidentes accesoras, se tienen los siguientes métodos

  ```
  public boolean executeStep()
  ```
  Es el método invocado por el gestor de mundo para ejecutar un paso de simulación. Dentro de él se recorre el ArrayList de Changers y para todos aquellos Changers que existan en un momento determinado en el mundo, se ejecutan sus relaciones cambiantes. Devuelve true o false dependiendo de si ha tenido éxito.

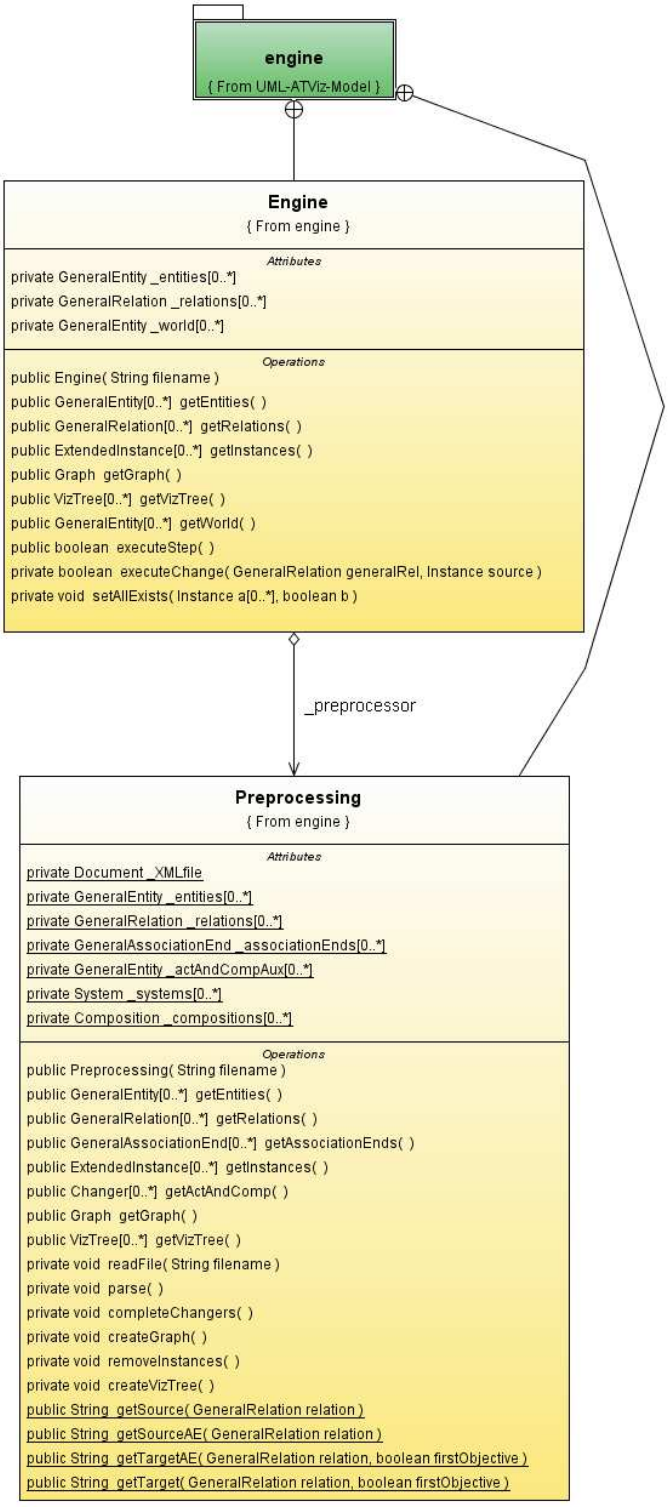
  ```
  private boolean executeChange(GeneralRelation
  generalRel, Instance source)
  ```
  Se invoca en el método anterior por cada posible relación cambiante. Dicha relación es pasada como parámetro al método, así como la instancia que lo invoca.

- Preprocesador: una vez más, aparte de las accesoras, encontramos los siguientes métodos

  ```
  private void readFile(String fileName)
  ```
  Abre el fichero cuya ruta se pasa como parámetro y lo guarda en un objeto de tipo Document, el cual es atributo de la clase.

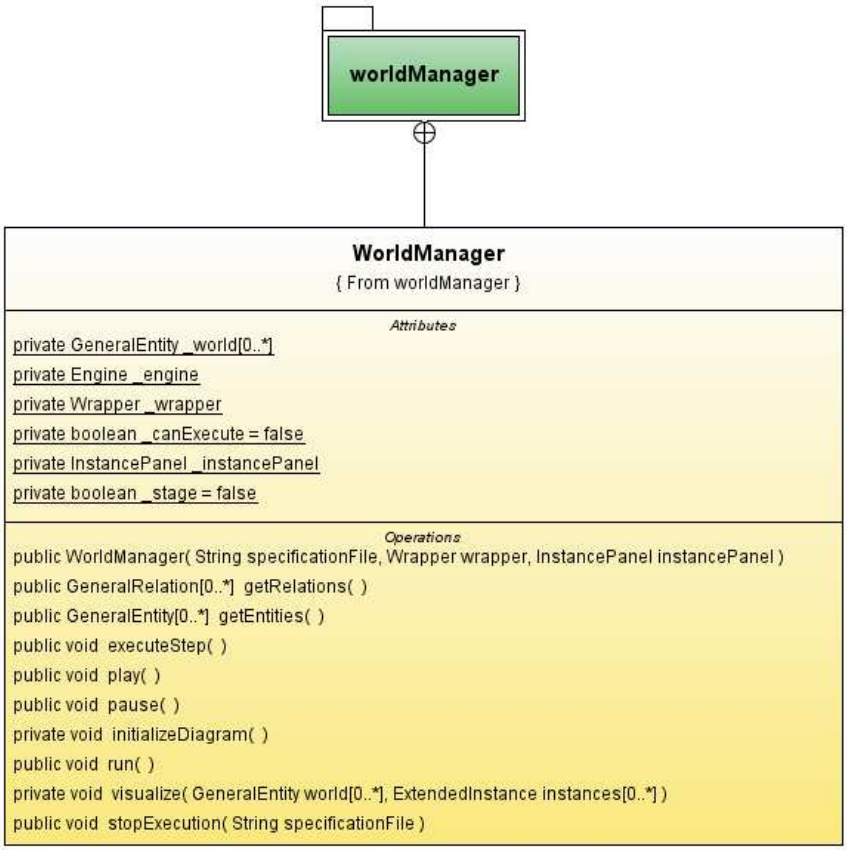
`private void` `parse()`
Realiza el parseo del fichero de especificación y construye los ArrayList de entidades, instancias, relaciones, association ends y changers.

`private void` `createGraph()`
`private void` `createVizTree()`
Generan las estructuras mediante las cuales se puede visualizar el modelo (ver Wrapper).



5.2.1.2 Gestor de Mundo

El gestor contiene al motor y al wrapper como atributos y extiende la clase 'Thread' para que, como se ha explicado anteriormente, el procesamiento de la especificación por parte del motor no interceda con la interacción del usuario con la aplicación a través de la GUI.



Sus principales métodos son:

`public void run()`
Es el método run de la clase Thread. La explicación más detallada de su implementación en el gestor se encuentra en la sección anterior.

`private void initializeDiagram()`
Como su propio nombre indica, inicializa la visualización, invocando al método *create* del Wrapper.

`private void visualize(ArrayList<GeneralEntity> World, ArrayList<ExtendedInstance> instances)`
Mediante este método se invoca a la visualización en el propio Wrapper. Para ello, se utilizan los dos ArrayList que se pasan como parámetros. El primero de ellos – World – contiene las entidades cuyas instancias han provocado un cambio en el mundo. El segundo, es el conjunto de las instancias que hay en el mundo, las cuales habrá que procesar dependiendo de si existen o no en un paso de ejecución.

```
public void play()
public void pause()
public void stopExecution(String specificationFile)
```
Son invocados en la GUI, debido a las órdenes del usuario. Sirven para parar, pausar o reanudar el *thread* del gestor de mundo.

5.2.1.3 GUI

Para este diagrama UML se ha prescindido de los elementos de Swing que forman la GUI principal, tales como botones, menús o la barra de herramientas por su elevado número y porque no tienen una relevancia especial a la hora de entender su funcionamiento interno. La GUI contiene un ModelPanel y un InstancePanel como atributos.

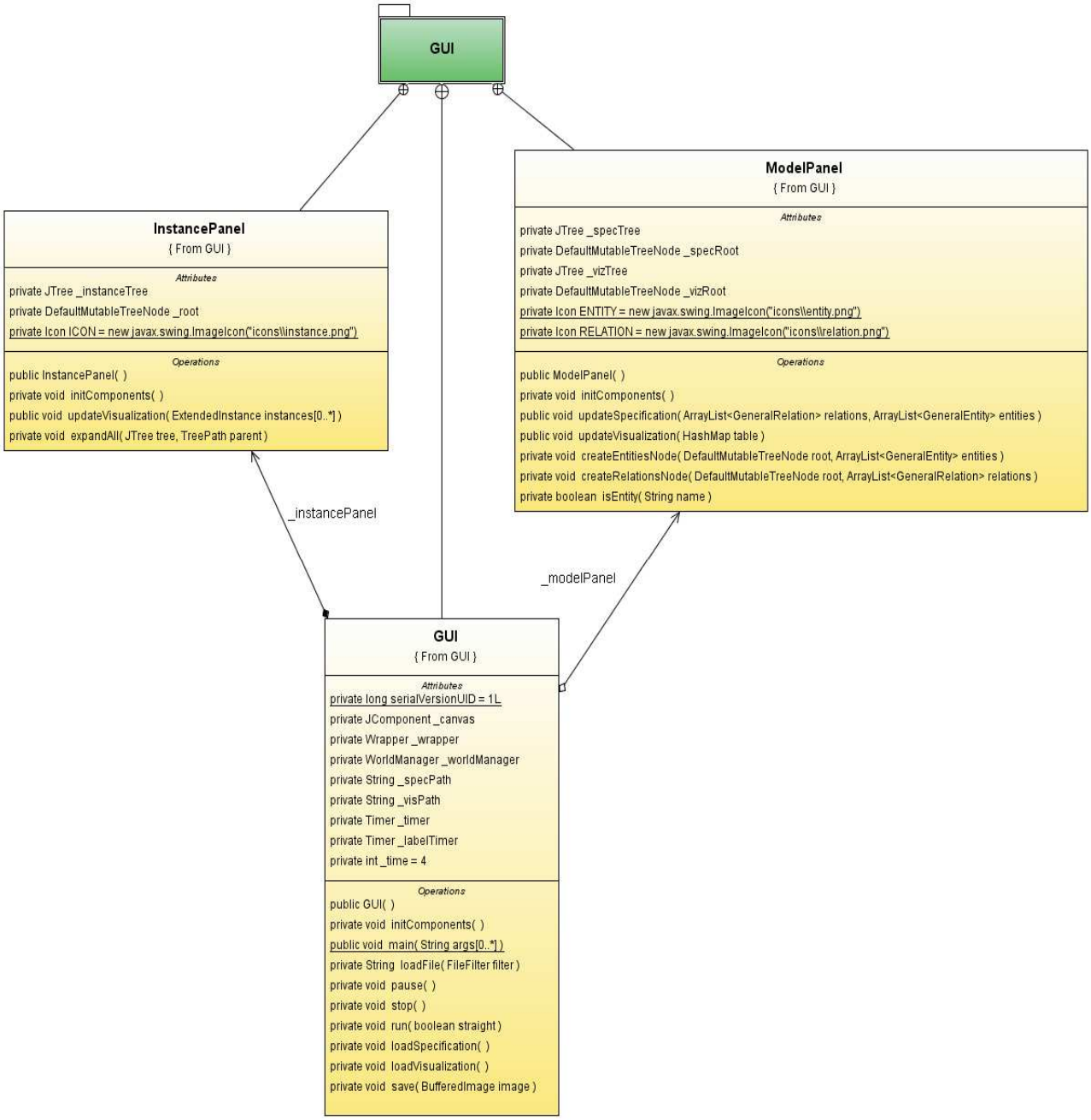
Los principales métodos de estas clases son:

- InstancePanel y ModelPanel: destacamos únicamente los métodos update, en los que se actualiza el contenido de los árboles en los que se muestran las instancias existentes en el mundo en el primer caso o el modelo de la especificación en el segundo.

- GUI: dos de sus principales métodos son

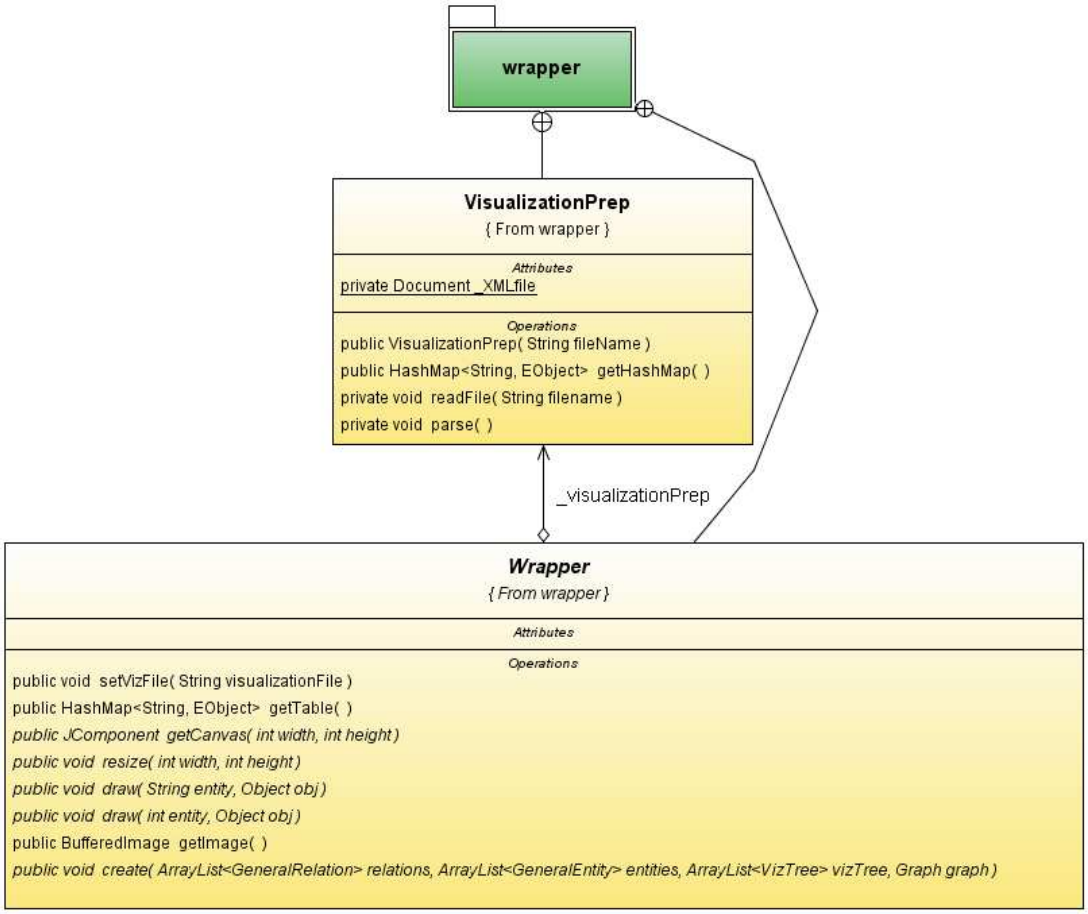
  ```
  private void loadSpecification()
  ```
  Muestra un JFileChooser mediante el cual se puede indicar el fichero de especificación que se desea cargar.

  ```
  private void run(boolean straight)
  ```
  Crea el gestor de mundo si es la primera ejecución. El parámetro indica si la ejecución será todo seguido (straight = true) o paso a paso (straight = false).


GUI
InstancePanel
{ From GUI }
Attributes
private JTree _instanceTree
private DefaultMutableTreeNode _root
private Icon ICON = new javax.swing.ImageIcon("icons\\instance.png")
Operations
public InstancePanel( )
private void initComponents( )
public void updateVisualization(ExtendedInstance instances[0..*])
private void expandAll(JTree tree, TreePath parent)
ModelPanel
{ From GUI }
Attributes
private JTree _specTree
private DefaultMutableTreeNode _specRoot
private JTree _vizTree
private DefaultMutableTreeNode _vizRoot
private Icon ENTITY = new javax.swing.ImageIcon("icons\\entity.png")
private Icon RELATION = new javax.swing.ImageIcon("icons\\relation.png")
Operations
public ModelPanel( )
private void initComponents( )
public void updateSpecification(ArrayList<GeneralRelation> relations, ArrayList<GeneralEntity> entities)
public void updateVisualization(HashMap table)
private void createEntitiesNode(DefaultMutableTreeNode root, ArrayList<GeneralEntity> entities)
private void createRelationsNode(DefaultMutableTreeNode root, ArrayList<GeneralRelation> relations)
private boolean isEntity(String name)
_instancePanel
_modelPanel
GUI
{ From GUI }
Attributes
private long serialVersionUID = 1L
private JComponent _canvas
private Wrapper _wrapper
private WorldManager _worldManager
private String _specPath
private String _visPath
private Timer _timer
private Timer _labelTimer
private int _time = 4
Operations
public GUI( )
private void initComponents( )
public void main(String args[0..*])
private String loadFile(FileFilter filter)
private void pause( )
private void stop( )
private void run(boolean straight)
private void loadSpecification( )
private void loadVisualization( )
private void save(BufferedImage image)

5.2.1.4 Wrapper

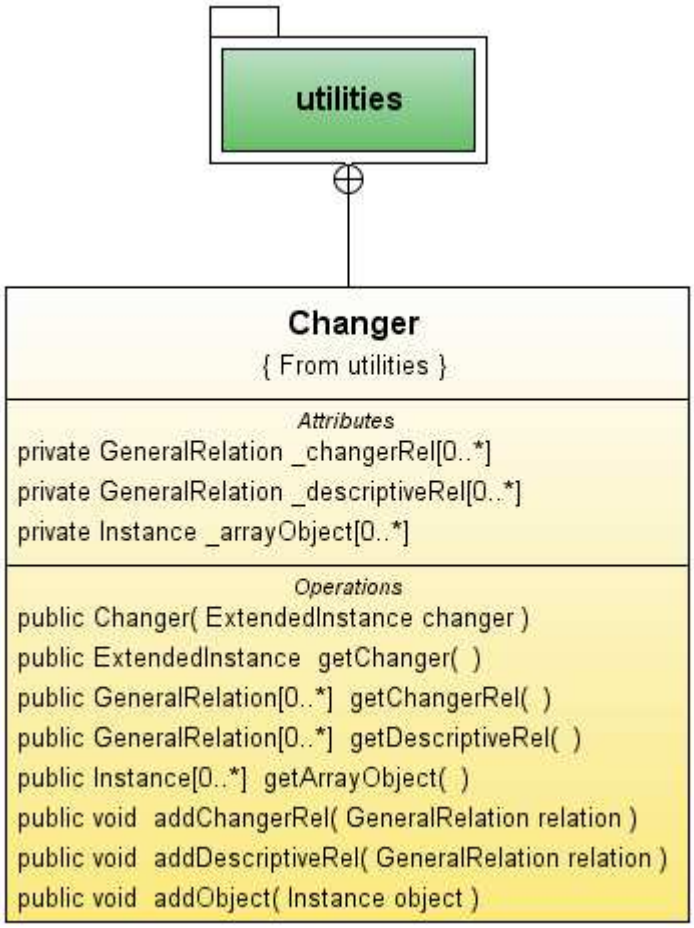
Los métodos que conforman esta clase abstracta fueron explicados en la sección anterior.



5.2.1.5 Changer

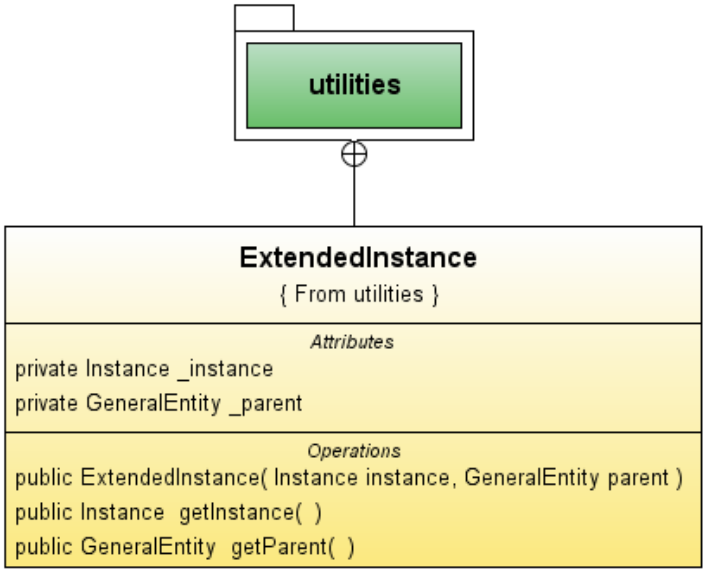
Como ya se ha explicado anteriormente, esta clase modela una instancia que puede cambiar el estado del mundo junto con el conjunto de sus relaciones cambiantes y no cambiantes.

Como se puede observar en el diagrama, sólo dispone de accesoras y mutadoras junto a los atributos.



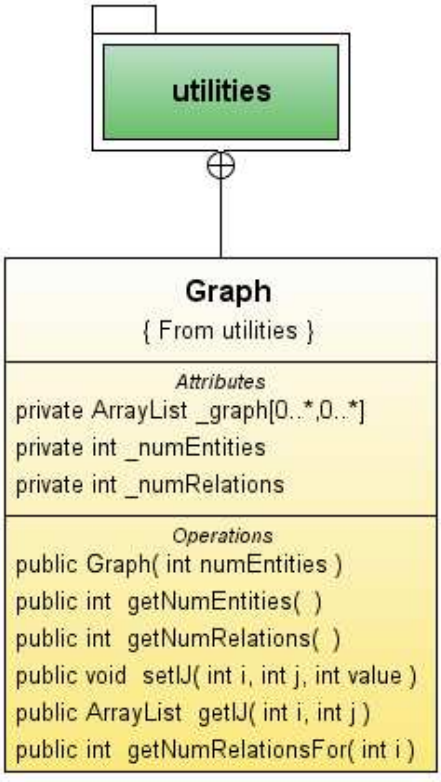
5.2.1.6 ExtendedInstance

Para entender el significado de esta clase, se debe consultar la sección anterior. El diagrama UML de la clase ExtendedInstance es el siguiente:



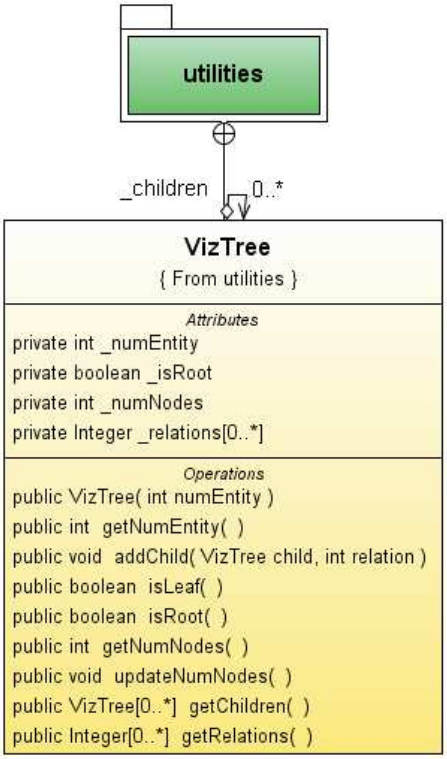
5.2.1.7 Graph

Mediante esta clase se modela el multigrafo que representa a la especificación del modelo. Su uso está descrito en la sección anterior.



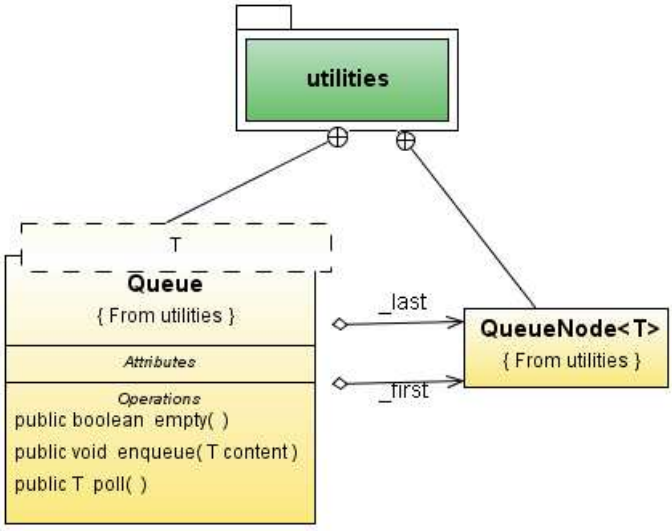
5.2.1.8 VizTree

Como ya se explicó en la anterior sección, este árbol se utiliza para apoyar a la visualización. Su diagrama de clases es el siguiente:



5.2.1.9 Queue

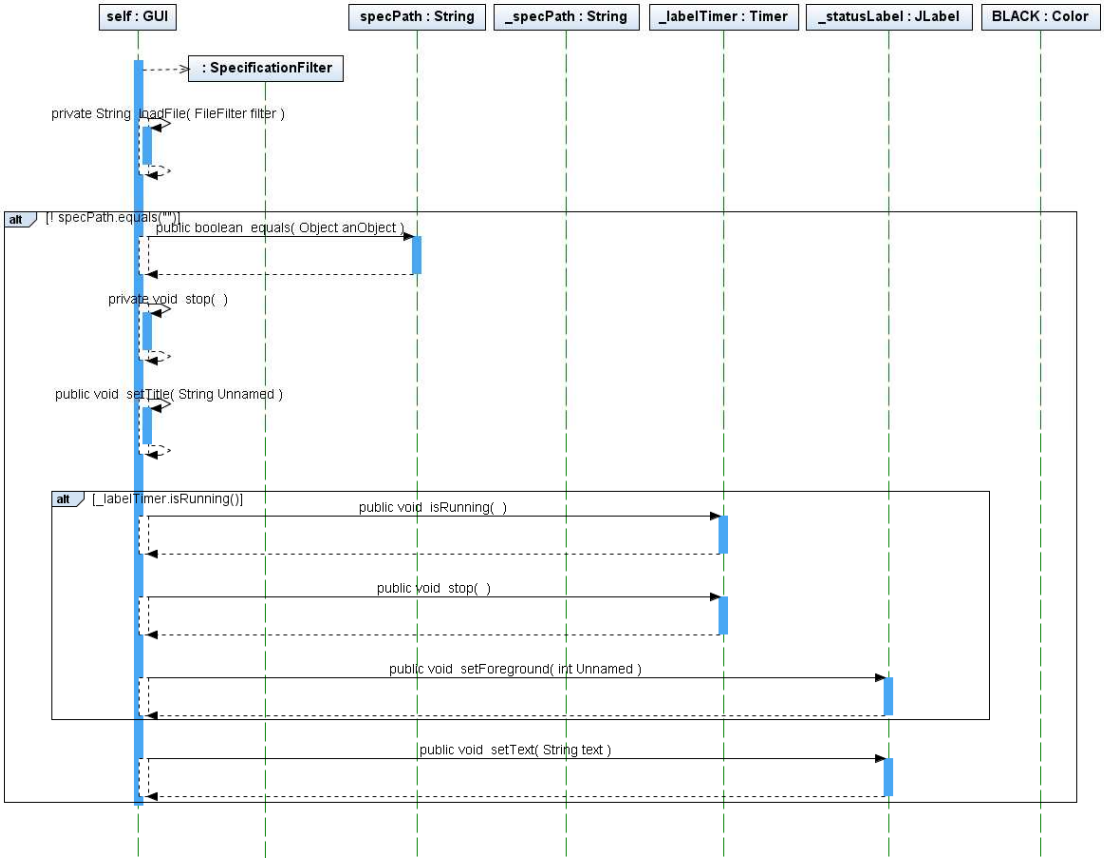
No se trata sino de una cola genérica. Como ya se indicó, en la aplicación se utiliza para recorrer el árbol de visualización por niveles.



5.2.2 Diagramas de Secuencia

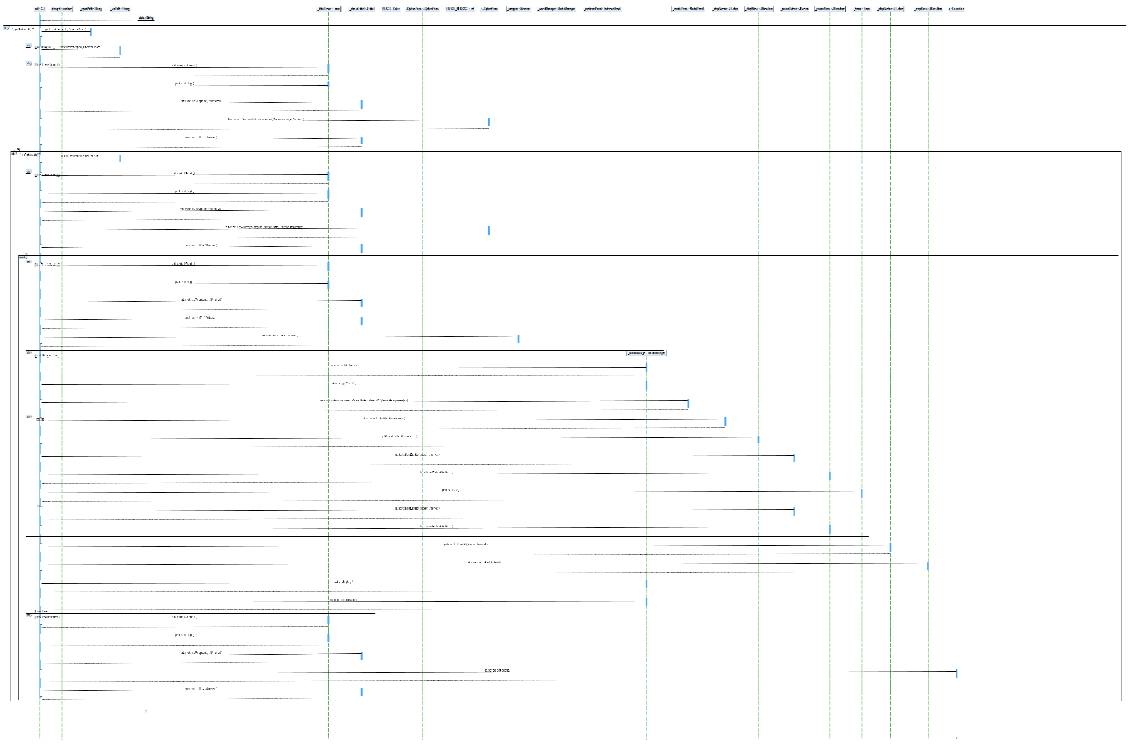
5.2.2.1. LoadSpecificationFile

En este primer diagrama de secuencia, ilustramos la operación 'loadSpecificationFile'. A través de ella se carga en el sistema el nombre del fichero de especificación que después se le pasará al motor para que lo analice con el preprocesador. Además, también se notifica por pantalla si se ha cargado correctamente o no a través de la barra de estado situada en la parte inferior de la GUI:

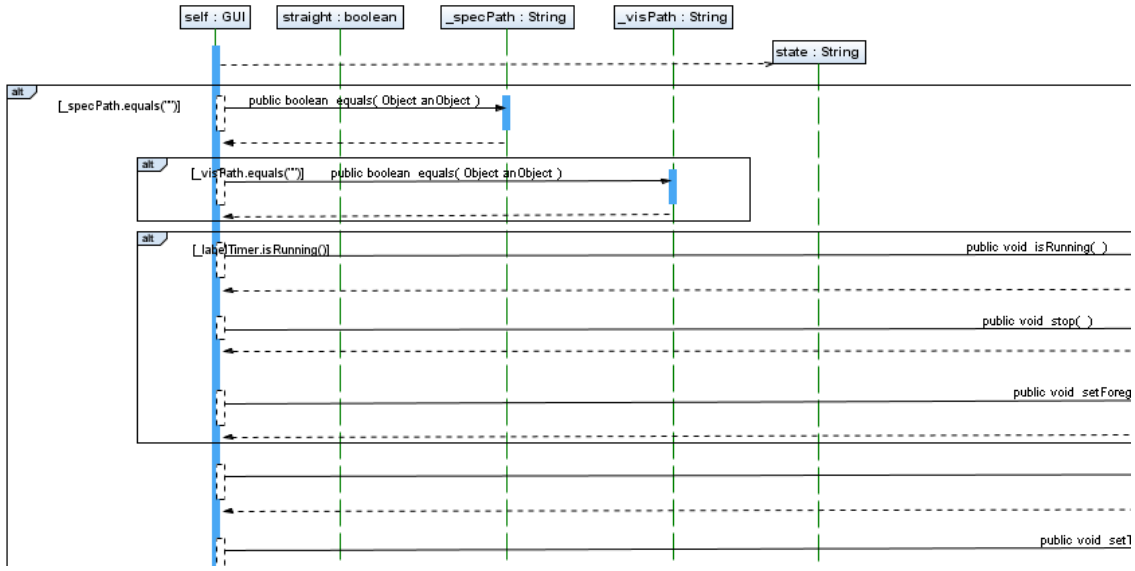


5.2.2.2. Run

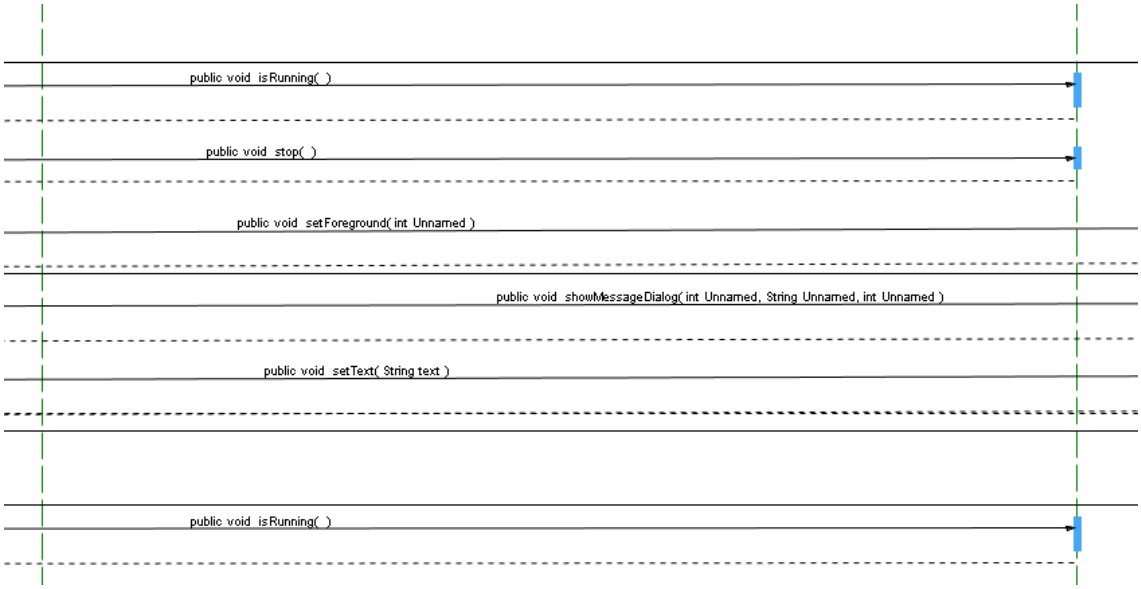
El siguiente diagrama, si bien no es muy útil su lectura por sus dimensiones – incluso en formato apaisado no se aprecia bien el texto – sí que nos sirve para ilustrar la complejidad de la operación *run* cuando se pulsa el botón de 'Play'. Al invocar el método *run*, ponemos en marcha el motor a través del 'WorldManager' y se van sucediendo los pasos de ejecución, reflejándose externamente a través de la GUI.



Aquí podemos ver unos fragmentos del diagrama:

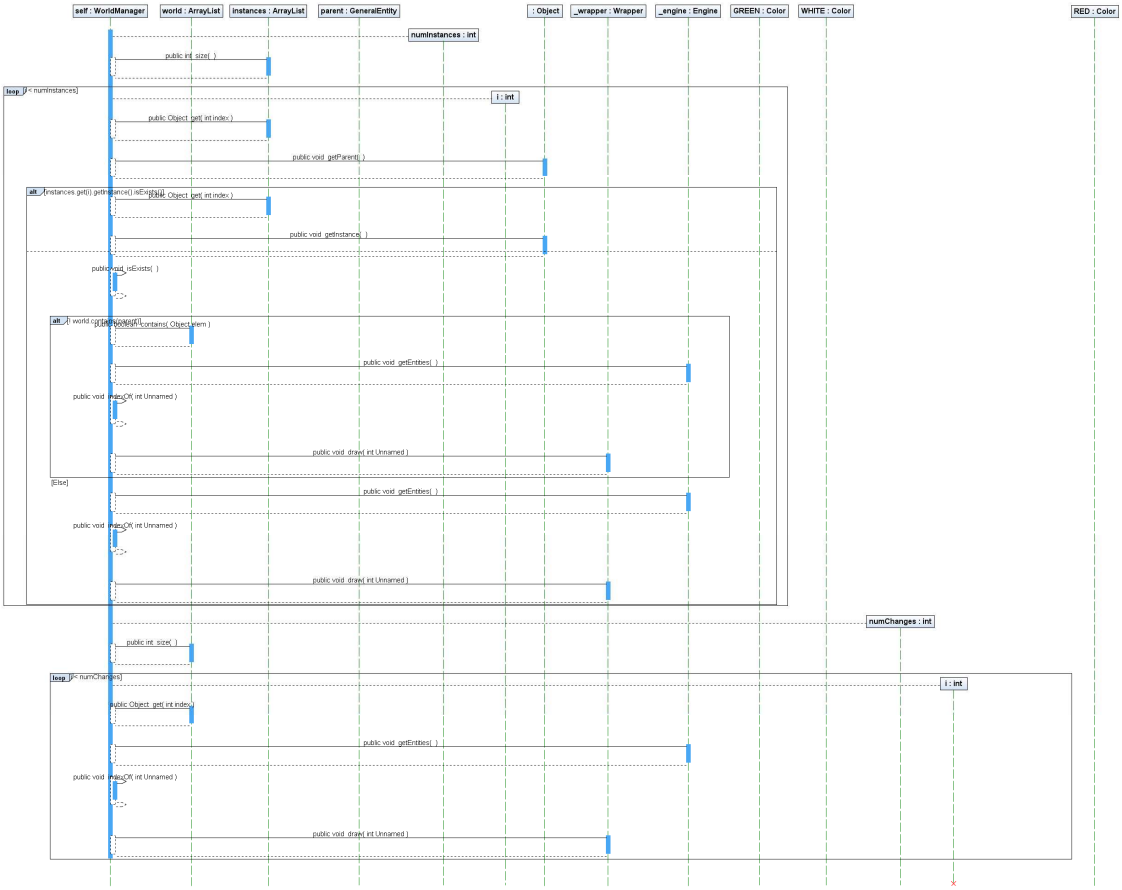



public void setT
[Else]
alt
[_visPath.equals("")]
public boolean equals(Object anObject)
alt
[_label.Timer.isRunning()]
public void isRunning( )
public void stop( )
public void setForeg
public void setT
assert
alt
[_label.Timer.isRunning()]
public void isRunning( )


public void isRunning( )
public void stop( )
public void setForeground(int Unnamed)
public void showMessageDialog(int Unnamed, String Unnamed, int Unnamed)
public void setText(String text)
public void isRunning( )

5.2.2.3. Visualize

Por último, mediante el siguiente diagrama de secuencia ilustramos la operación *visualize* perteneciente al 'WorldManager'. Con este método se consigue visualizar el estado del mundo, como su nombre indica. Se invoca cada vez que ha habido un paso de ejecución por parte del motor, el cual brinda al 'WorldManager' la información relativa a lo que tiene que visualizar, a saber: las instancias que existen y no existen (estado del mundo) y las instancias que han provocado un cambio en el mundo en ese paso de ejecución.



5.3. Pruebas

Para concluir esta sección queremos ilustrar con un ejemplo de ejecución nuestra aplicación. Remitimos asimismo al apéndice situado al final de esta memoria en el que se explica el manejo del sistema.

5.3.1. Capturas de pruebas

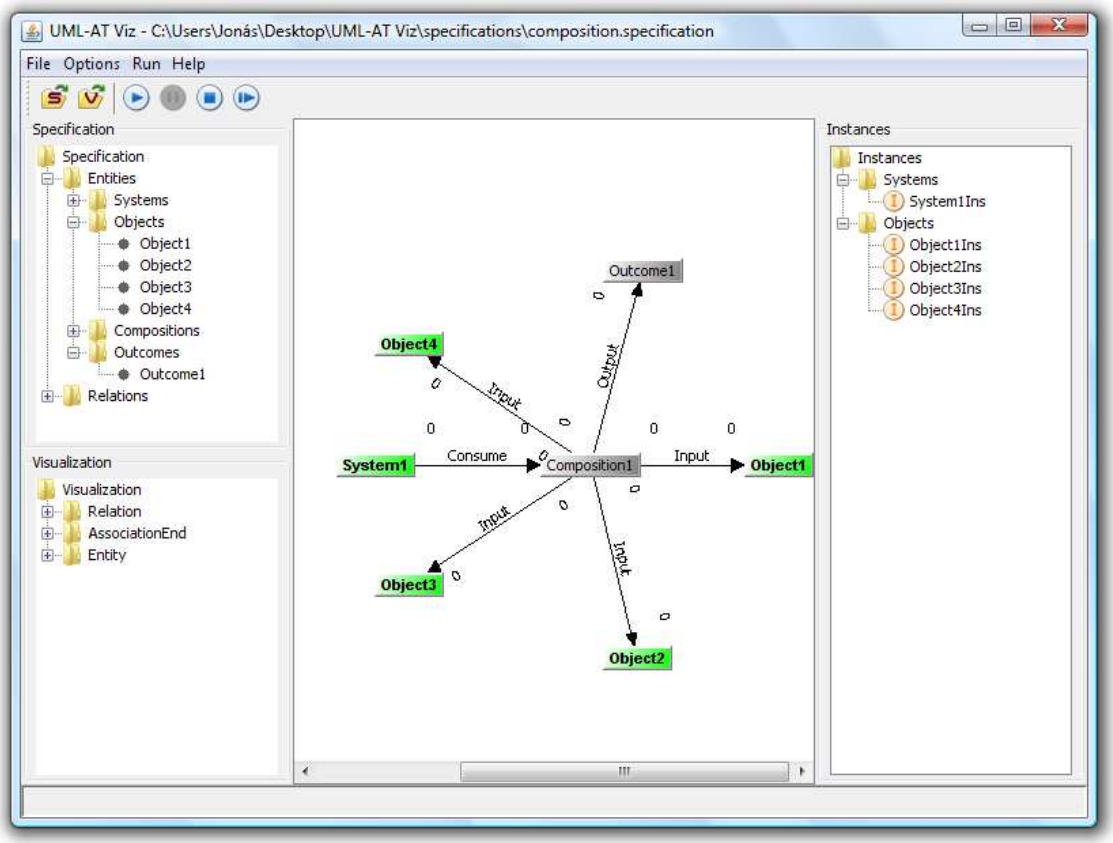
El fichero que vamos a cargar en la aplicación contiene una especificación muy sencilla, que nos sirve para mostrar el funcionamiento del entorno. Concretamente, dicha especificación es la siguiente:

System(system1) Object(object1) Object(object2)
Object(object3) Object(object4) Composition(composition1)
Outcome(outcome1)

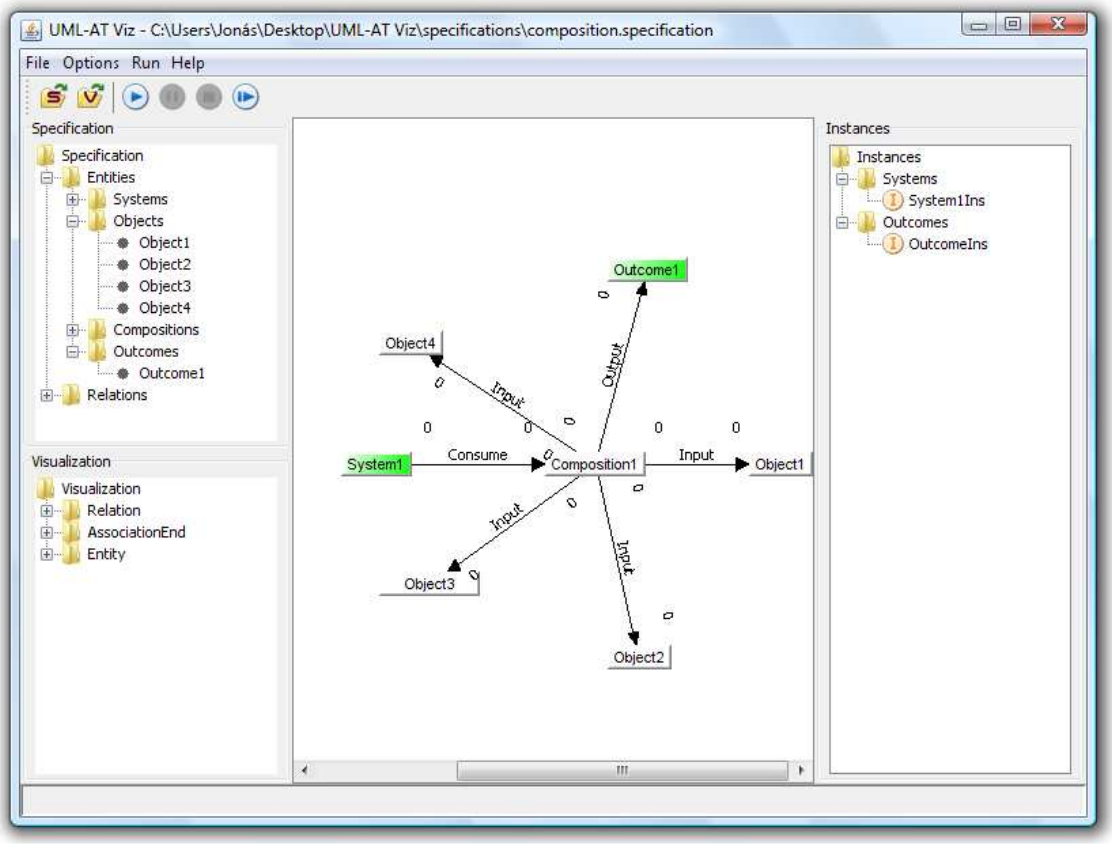
Input(composition1, object1) Input(composition1, object2)
Input(composition1, object3) Input(composition1, object4)
Output(composition1, outcome1) Consume(system1, composition1)

Como podemos ver, hay 4 'Object' que sirven de *input* a una 'Composition', un 'Outcome' resultado de esa composición y un sistema que consume la composición.

Procedemos a cargar el fichero en la aplicación, junto con el fichero de visualización básico y pulsamos ejecutar paso a paso:



Como muestra la imagen, se construye la visualización en el canvas y los paneles de especificación, visualización e instancias situados a ambos lados. Podemos pulsar ahora el botón de ejecutar y comprobar que cada 4 segundos ocurre un paso en la simulación. El usuario podrá observar en cada paso cómo cambian los colores de las entidades. Para comprender el significado de cada uno de estos colores, remitimos al apéndice I situado al final del documento, en el punto 8.3 (Manejo básico de la aplicación). En la siguiente imagen se muestra el final de la ejecución, en la que solamente existen dos entidades (ambas en color verde). Además, en el panel de instancias, sólo persisten las dos instancias de dichas entidades:



Una vez terminada la simulación podemos pulsar el botón de parar y volver a ejecutarla desde el principio o bien cargar otra simulación. Como se puede observar en el panel de la derecha, la evolución del estado del mundo en cada paso de simulación se refleja inmediatamente, con lo que podemos tener información exacta sobre qué instancias existen y cuales no. Además, el acompañamiento gráfico es otra fuente de referencia a la hora de analizar la especificación.

5.3.2. Conclusiones de pruebas

En general, tras las pruebas realizadas a lo largo del desarrollo de la aplicación y que aquí hemos querido ilustrar con el anterior ejemplo, concluimos que la herramienta realiza una tarea eficaz de procesamiento del modelo. Hemos reportado pocos fallos durante el tiempo de desarrollo y los que han surgido fueron rápidamente resueltos. Uno de los puntos clave para poder corregir dichos errores fue la introducción de la

visualización, la cual servía de perfecta ilustración a lo que ocurría por dentro de la aplicación.

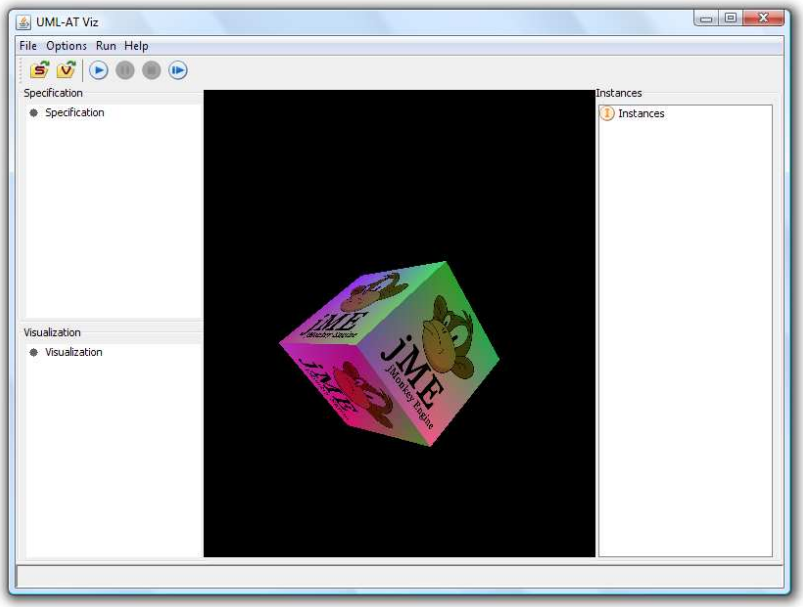
La aplicación fue probada con diversos ejemplos de prueba, con especificaciones más grandes incluso, de las que obtuvimos resultados satisfactorios. Sin embargo, únicamente podemos encontrar una pequeña nota negativa en la visualización, debido a la librería que hemos utilizado, jGraph, la cual presenta ciertos errores de vez en cuando a la hora de introducir cambios en el grafo construido. Como se puede apreciar tras algunas ejecuciones, la librería comete fallos al visualizar y en las pruebas, cuando cometía esos fallos, lanzaba excepciones por consola. Sólo queremos destacar ese punto negativo resaltando vehementemente que no es concerniente a nuestro desarrollo y que, como hemos explicado anteriormente, la visualización básica es fácilmente modificable por cualquier otra librería o motor gráfico, siendo éste quizás uno de los puntos más fuertes en los que se basa la aplicación.

6. Conclusiones

De los objetivos que nos propusimos al comenzar este proyecto hemos conseguido cumplir los siguientes:

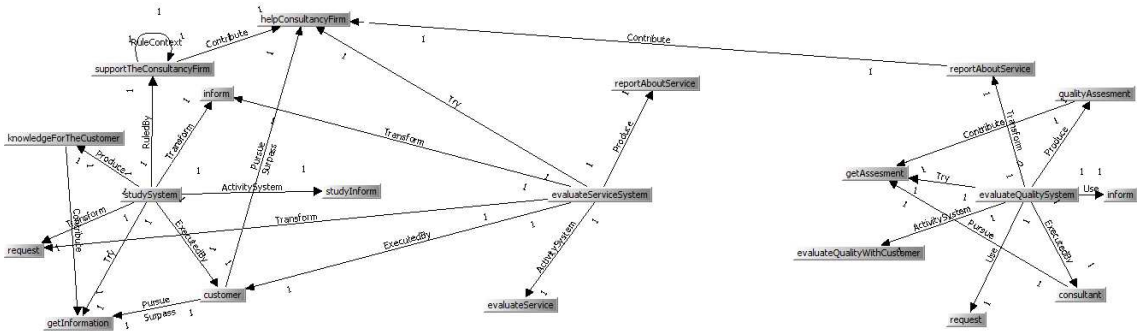
- Definición y modelado del lenguaje UML-AT gracias al EMF
- Modularización de los componentes que conforman la aplicación
- Fácil acoplamiento entre la aplicación y distintos tipos de visualización concedida por librerías o medios externos

Cabe mencionar que uno de los objetivos que teníamos pensado terminar ha quedado pendiente por falta de tiempo, tratándose éste de ampliar la visualización básica para que se represente el mundo virtual en 3D. No obstante, reiteramos que es posible realizarlo de manera sencilla – al menos en cuanto a acoplamiento con nuestra aplicación se refiere – gracias a la estructura de clases realizada en la implementación. De hecho, como se puede apreciar en la siguiente figura, el acoplamiento con el motor gráfico jMonkey fue un éxito:



Por otra parte, quedó descartada por falta de tiempo una opción mediante la cual el usuario pudiera guardar la ejecución en curso, con lo que se podría haber salvado el estado del mundo actual, así como los ficheros de visualización y especificación de los que partía la simulación. De esta forma, se podría haber otorgado al usuario la posibilidad de guardar simulaciones largas y poder recuperarlas en distintas sesiones de la aplicación. Además, dichas simulaciones salvadas presentarían una total transparencia a la visualización concreta. La motivación de esta funcionalidad es la de evitar que ejecuciones arbitrariamente grandes deban ser simuladas desde el inicio cada vez que se utiliza la aplicación.

Al margen de los objetivos principales para el desarrollo de este proyecto hemos implementado algunas funcionalidades adicionales que nos parecen muy útiles, como son los distintos controles sobre la ejecución del modelo, pues permiten al usuario un perfecto entendimiento del estado de la simulación, pudiendo avanzar, parar o pausar la simulación o la opción de poder salvar en un fichero de imagen la visualización actual, con propósito general y así evitar al usuario tener que realizar capturas de pantalla para extraer la visualización; con esto se otorgaría una manera de obtener la visualización del modelo en formato imagen y así tener la posibilidad de utilizarla en documentos, memorias, artículos, etc., ilustrando el modelo.



6.1. Aportaciones

En este trabajo hemos abordado el tema del proceso de modelado basado en agentes explicando varios de los problemas que esto trae consigo. La solución que hemos propuesto para abordar dichos problemas es la introducción de un lenguaje de alto nivel, metódico y formal, que fuera más cercano al dominio del experto y en forma de diagramas fácilmente entendibles, para su fácil utilización, comprensión y estudio. Para hacer todo esto, se ha implementado una aplicación que sirva como capa intermedia para facilitar la comunicación, especificación, implementación y validación de un modelo. También se han propuesto unas pautas a la hora de aplicar este tipo de lenguaje a cualquier modelo basado en agentes. Estas usan el soporte de una herramienta de modelado para el lenguaje de metamodelado que se ha elegido.

Las ventajas que proporcionan el uso de metamodelos las hemos comentado ampliamente a lo largo de esta memoria. Sin embargo existen algunas limitaciones. Los metamodelos requieren un esfuerzo extra que puede que no fuera necesario en ciertos casos. Podría ser el caso de algunos modelos pequeños con un solo científico ejecutando los cuatro roles. En ese caso, la mayor parte de las ventajas de este enfoque se ven minimizadas, y el modelador puede que no las encuentre suficientemente útiles.

Además incluso en los casos donde el vocabulario del dominio específico se ha usado, algunas de las estructuras de diagramas formales, especialmente las complejas, puede que no sean fáciles de entender para el experto de la temática. Aunque muchos expertos en modelado e ingenieros del software están acostumbrados a utilizar diagramas estructurados como ontologías, puede no ser tan trivial para expertos del dominio que las pueden encontrar muy difíciles de leer.

También ha sido necesario adaptar algunos de los conceptos principales del modelado basado en objetos para la ingeniería de agentes ya que ésta varía ligeramente el enfoque general de los sistemas multiagente. Sin embargo esta adaptación es factible ya que tanto el lenguaje como la herramienta permiten fácilmente la posibilidad de introducir extensiones como nuevos conceptos, relaciones e iconos gráficos.

No obstante consideramos que el enfoque de este trabajo supone un paso importante en la búsqueda de modelos basados en agentes más fiables y transparentes.

Aunque el modelado informático es un mero instrumento al servicio del sociólogo, en nuestra opinión y experiencia, éste ayuda enormemente a concebir y formalizar el modelado sociológico, ya que es necesario alcanzar una visión global de un proceso social en su contexto dinámico e interactivo con otros hechos y procesos sociales. También porque al requerir una mayor precisión en el modelo formal que en el modelo descrito en lenguaje natural utilizado por el sociólogo, se fuerza a un desarrollo de modelos explicativos más precisos y formales que los que habitualmente se realizan, Además, esto ayuda a combinar diversas explicaciones, teorías y datos, y a formular preguntas que quizás anteriormente no se habían planteado.

6.2. Trabajo Futuro

Una vez hemos finalizado el trabajo, una de las conclusiones que hemos comprendido es lo difícil que puede resultar la aplicación de la simulación social a la sociología. Esto no sólo es debido a la parte en la que pueda influir la vertiente informática sino también a la parte sociológica, es decir, aún existen un gran desconocimiento de fenómenos sociales básicos, tanto por carencia de evidencias empíricas, como por limitaciones de las teorías sociológicas. De ahí que creamos que la simulación social puede ser un buen instrumento para tomar conciencia de estas carencias y tratar de superarlas. Esto significa que aparte de los nuevos descubrimientos que se hagan de su estudio, al poder proporcionar nuevas herramientas para la experimentación en las ciencias sociales, se deberían formular modelos explicativos mucho más formales, elaborados, que los sociólogos acostumbran.

6.2.1. Futuras Ampliaciones

Entre las futuras ampliaciones que podría hacerse a la aplicación resultante del proyecto se encuentran la de adaptar la visualización a dominios concretos, haciendo uso de distintas herramientas gráficas – como el mencionado motor 3D jMonkey – y utilizando ficheros de visualización adaptados a cada caso. De esta manera, se podría realizar una biblioteca de archivos de visualización adaptados a dominios más específicos y que estarían disponibles junto con la aplicación. Además, al haber diseñado la herramienta de modo que los posibles usuarios puedan participar activamente en ella, cambiando la visualización o el modo en que ésta se utiliza, la aplicación se podría adaptar a los usos concretos que cada uno quiera realizar.

Otras ampliaciones interesantes podrían encontrarse en otorgar una mayor interactividad al usuario a la hora de crear modelos, pues la forma en que ahora se especifica – mediante el editor de árbol de Eclipse – puede resultar poco intuitiva para usuarios poco expertos o que no tengan una gran relación con las herramientas informáticas. Una extensión práctica sería la de otorgar un editor de modelos de UML-AT gráfico junto con la aplicación de simulación que hemos presentado en este proyecto, de manera que se especifique el modelo por medio de figuras, tales como *cajas*. Este editor se encargaría de generar el fichero XML para cargar en la aplicación de visualización, concediendo así una participación más sencilla, amena e intuitiva al usuario a la hora de construir especificaciones.

En cuanto a la definición de los modelos, creemos que sería mucho más correcto separar la especificación del modelo de la especificación de las instancias, ya que ahora se realiza de manera conjunta. En nuestra opinión, tratar dentro de una misma especificación ambas cosas trae consigo una mayor confusión y no contribuye a que el modelo sea una pieza totalmente modular. Sería más favorable que se indicase la especificación del modelo por una parte y por otra (que podría ser dentro de la propia aplicación que presentamos) se creasen las instancias que van a participar en la simulación. Por tanto, una nueva ampliación de la herramienta incluiría la creación de instancias de manera interactiva a través de ella con la posibilidad de guardar la especificación de instancias, para poder hacer uso de ella más adelante. Esto ofrece ventajas evidentes a los usuarios, ya que pueden usar una única especificación de

modelo, pero distintas especificaciones de instancias, de manera que se contribuiría a crear una biblioteca de modelos, al igual que la de visualizaciones que se ha indicado anteriormente.

Finalmente, se podría extender la funcionalidad menos básica para ofrecer algunos servicios tales como guardar una simulación en mitad de su ejecución, de manera que en simulaciones arbitrariamente grandes o muy largas no haya que ejecutar desde el principio en cada uso de la herramienta; ofrecer algún tipo de estadística que refleje resultados concretos, etc.

7. Glosario

Agente: es una entidad capaz de percibir su entorno, procesar tales percepciones y responder o actuar en su entorno de manera racional, es decir, de manera correcta y tendiendo a maximizar un resultado esperado.

ECore: es la base de Eclipse Modeling Framework. Es una herramienta para ayudar en el diseño de lenguajes de metamodelado, ofrece pautas y directrices para su desarrollo.

EMF(Eclipse Modeling Framework): permite la construcción de editores y otras aplicaciones basadas en modelos de datos.

Ejecutar: simula el modelo ejecutando un paso cada cierto tiempo. Dicho tiempo será medido en segundos y será configurable.

Ejecutar paso a paso: es la forma de simular el modelo de manera que el usuario decida cuándo se ejecuta un paso de simulación.

Especificación del modelo: se entiende por esto la especificación de un modelo abstracto utilizando el lenguaje de UML-AT, del cual deberá haber instancias de las entidades para que puedan interactuar según las 'reglas' que proporciona dicho modelo.

Estado del mundo: instancias que existen tras un paso de ejecución.

Fichero de especificación del modelo: es el fichero de extensión .specification que contiene la especificación del modelo.

Fichero de visualización: es el fichero de extensión .visualization que contiene la manera en que se visualizará el modelo especificado.

Gestor de mundo: parte de la aplicación encargada de dirigir la simulación y que comunica los diferentes módulos.

Instancia: individuo concreto de una entidad determinada.

Interacción: actividad social entre varios agentes.

Metamodelado: Es un recurso para definir los lenguajes específicos de dominio de tipo grafo. Introduce nodos, arcos y atributos, que corresponden a los conceptos del dominio y satisfacen sus reglas y restricciones.

Motor: parte de la aplicación que se encarga de procesar el modelo.

Sistema Multiagente: es un sistema distribuido en el cual los nodos o elementos son sistemas de inteligencia artificial, o bien un sistema distribuido donde la conducta combinada de dichos elementos produce un resultado en conjunto inteligente.

UML-AT: lenguaje para especificar sistemas sociales a través de grafos.

Visualización del modelo: es la especificación de la manera en que se visualizarán las distintas partes que conformen una especificación, desde sus entidades a sus relaciones o posibles metadatos que el posible desarrollador pueda incluir y que tengan sentido para la visualización que quiera implementar.

Wrapper: parte de la aplicación encargada de que exista transparencia en cuanto a lo que la visualización se refiere. Es el punto de conexión entre la aplicación y las distintas librerías de visualización.

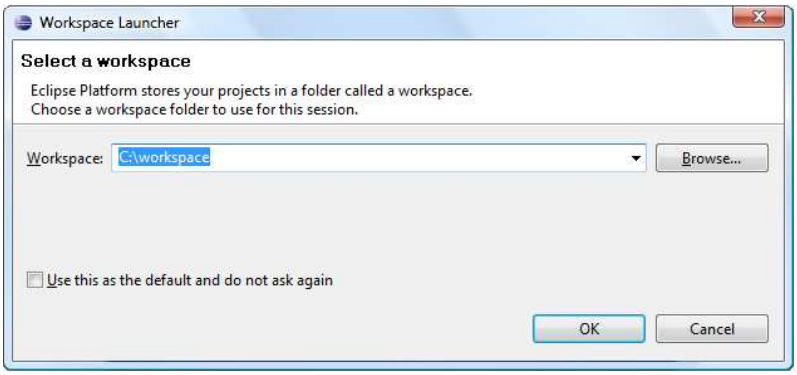
8. APENDICE I: Manual de Usuario

8.1. Desarrollo de un nuevo fichero de especificación de modelo

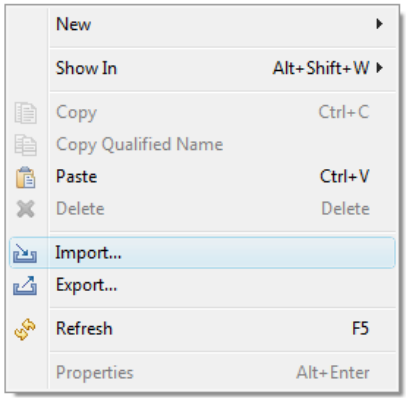
A lo largo de este punto introduciremos a la especificación de un nuevo modelo que puede ser visualizado a través de la herramienta desarrollada. En primer lugar hay que tener instalado el IDE Eclipse. Los pasos a seguir son los siguientes:

1. Ejecutar Eclipse

2. Seleccionar la carpeta en la que queremos que se guarde el Workspace.

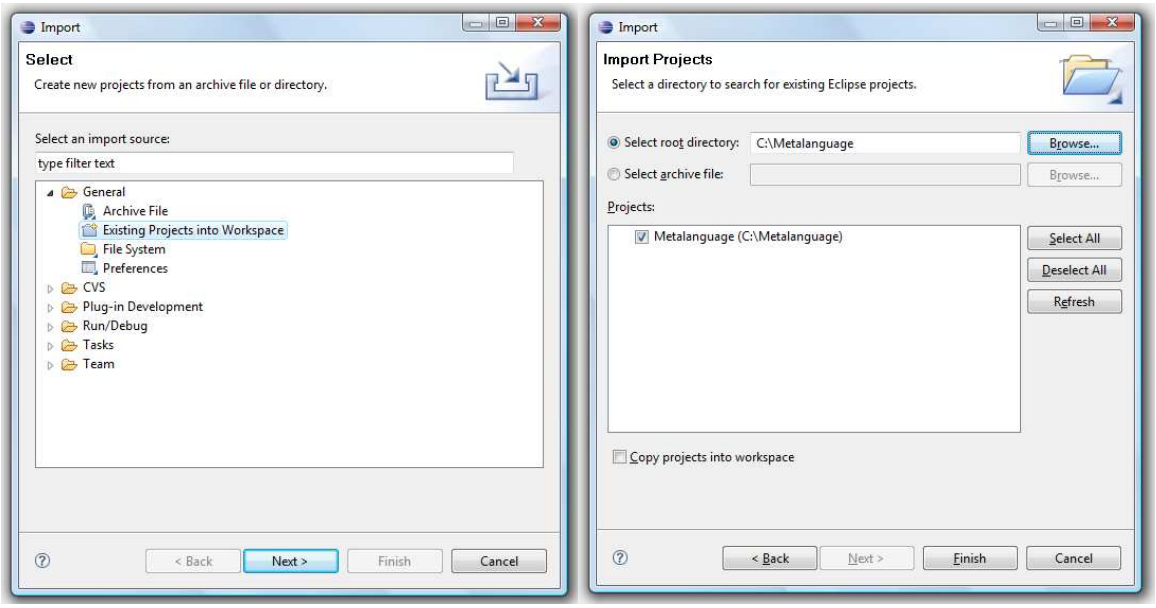


3. Una vez abierto Eclipse, pulsar con botón derecho en el panel de la izquierda, 'Package Explorer', y seleccionar 'Import…'.



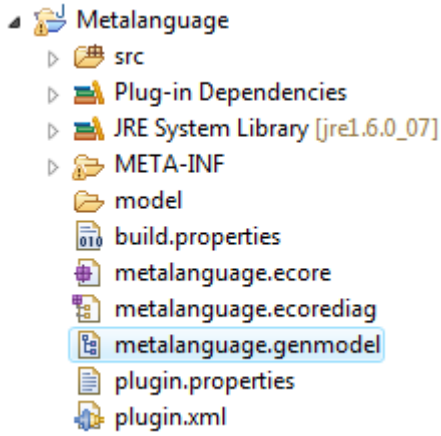
4. En el cuadro de diálogo que aparece, seleccionar
General >> Existing Projects into Workspace y pulsar 'Next'.

5. En el diálogo que aparece, seleccionar el directorio en el que se encuentra el proyecto 'Metalanguage' mediante el botón 'Browse'. Una vez seleccionado el directorio, aparecerá el proyecto en la lista que hay más abajo llamada 'Projects'. Comprobar que el proyecto aparece marcado y pulsar el botón 'Finish'.

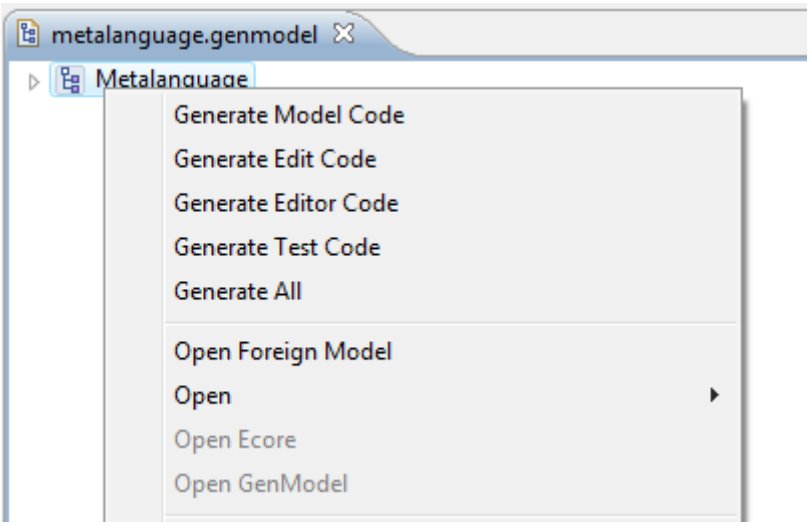


6. Mientras se abre el proyecto, es posible que Eclipse intente compilarlo si la opción 'Build Automatically' del menú 'Project' está seleccionada. Mientras dura este proceso, es muy probable que Eclipse reporte errores, los cuales desaparecerán al terminar la compilación.

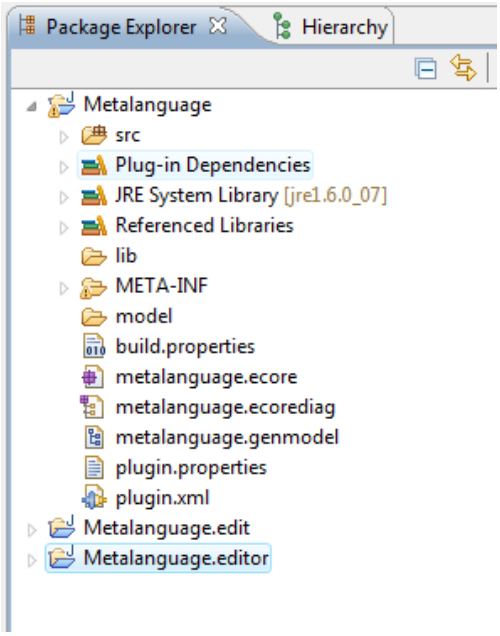
7. Desplegar el proyecto pulsando en el icono de proyecto que ha aparecido dentro de la pestaña 'Package Explorer' y hacer doble clic sobre el fichero 'metalanguage.genmodel'



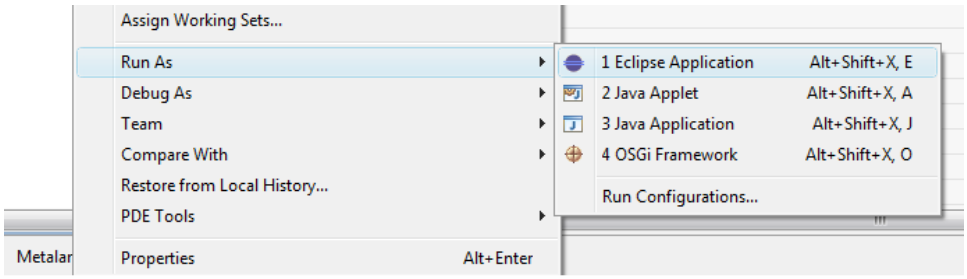
8. Al haber hecho doble clic, se abre el fichero en la parte central del IDE y aparece un icono que se puede desplegar. Sin desplegar, hacer clic derecho sobre el icono, donde aparecerá el siguiente menú contextual:



9. Por orden, pulsar primero sobre la opción 'Generate Edit Code' y después repetir clic derecho sobre el nodo 'Metalanguage' y elegir 'Generate Editor Code'. Esto generará, en orden sucesivo, dos proyectos que aparecerán en el 'Package Explorer'. Si aparece algún error en alguno de los proyectos, probar a hacer un 'Clean' de ese en concreto a través del menú 'Project' de Eclipse.

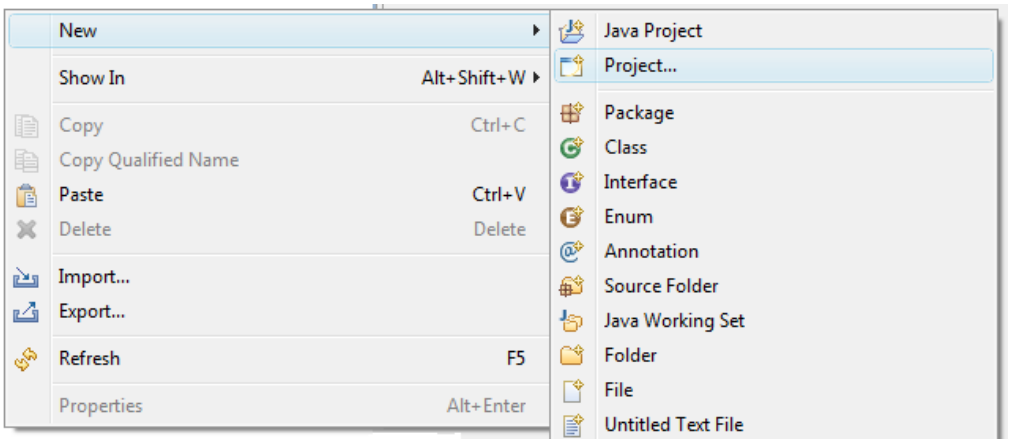


10. Hacer clic derecho sobre el proyecto 'Metalanguage.editor' y seleccionar **Run As >> Eclipse Application**



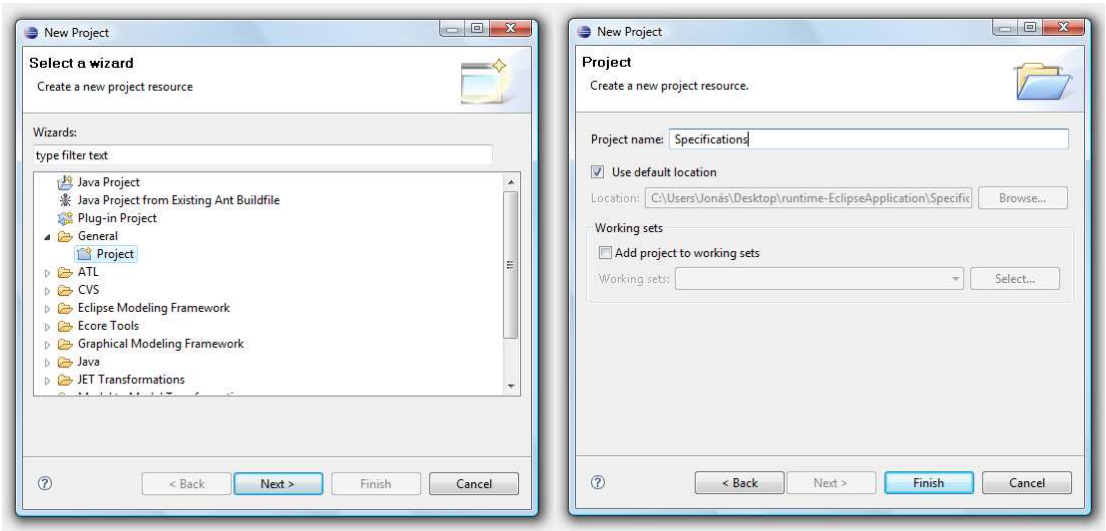
11. Se abrirá otra ventana del IDE que aparecerá vacía por iniciarse por primera vez.

12. Hacer clic derecho en una región vacía de la pestaña de la izquierda ‘Package Explorer’ y seleccionar la opción **New >> Project…**

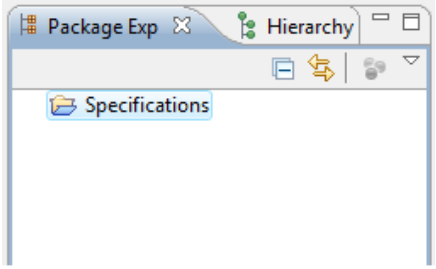


13. En el diálogo que aparece, seleccionar **General >> Project** y pulsar ‘Next’.

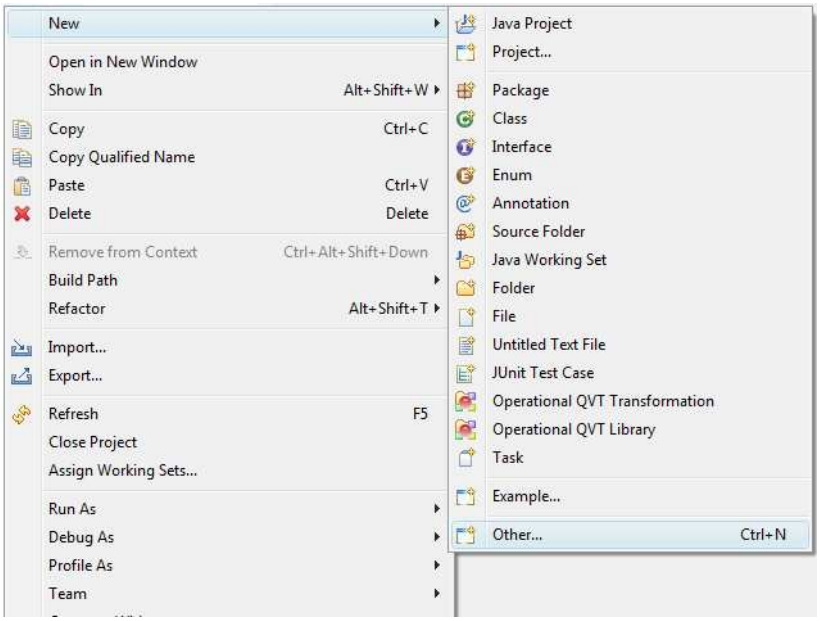
14. En el siguiente cuadro de diálogo, escribir el nombre del proyecto y pulsar ‘Finish’. Observar que el proyecto será guardado en la ruta por defecto a menos que se seleccione otra distinta desmarcando la opción ‘Use default location’ y seleccionándola con el botón ‘Browse’. La ruta por defecto para este ejemplo se puede ver en la imagen de la derecha.



15. Al hacer esto, se creará el proyecto en el ‘Package Explorer’.

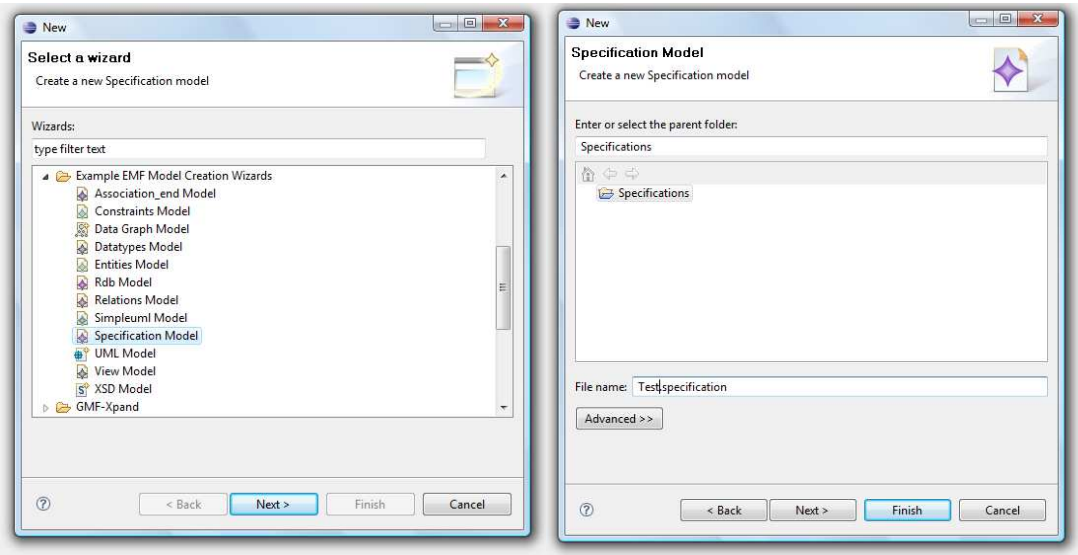


16. Hacer clic derecho sobre el icono del nuevo proyecto y elegir la opción **New >> Other…**

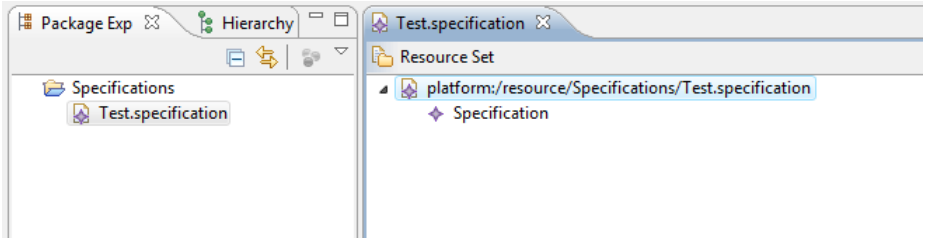


17. En el cuadro de diálogo que aparecerá, seleccionar **Example EMF Model Creation Wizards >> Specification Model** y pulsar 'Next'.

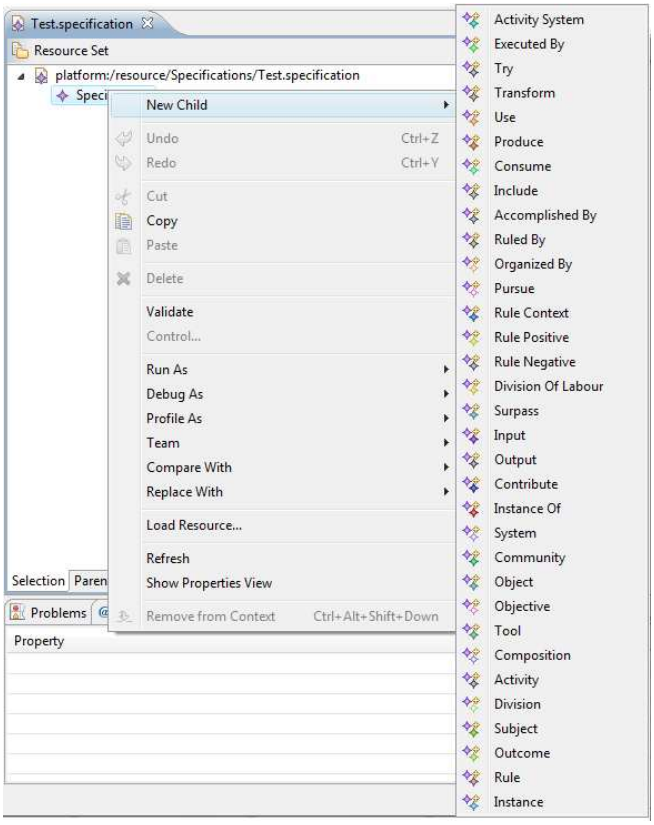
18. Escribir el nombre del fichero .specification y pulsar 'Finish'.



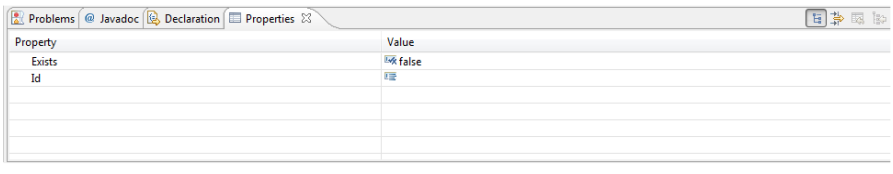
19. Dentro del proyecto, en la pestaña ‘Package Explorer’ a la izquierda, aparecerá el fichero. En el centro veremos el contenido del fichero en forma de árbol. Para empezar a editar hay que desplegar el primer nodo para acceder al nodo ‘Specification’.



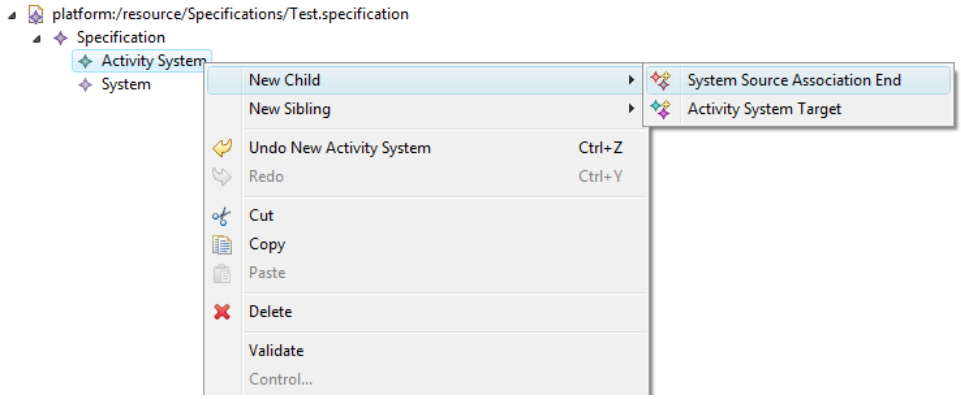
20. Haciendo clic derecho sobre el nodo ‘Specification’, podemos añadir entidades y relaciones a través del menú contextual, en su opción ‘New Child’.



20. Al añadir una nueva entidad o relación, aparecerá abajo la pestaña ‘Properties’ con sus propiedades y valores. Si no se viera dicha pestaña, hacer clic derecho en el nodo de la entidad y pulsar en el menú contextual que aparecerá sobre la opción ‘Show properties view’. El siguiente ejemplo son las propiedades para una entidad.



21. Para añadir ‘association ends’, se debe añadir primero la relación. Una vez tengamos el nodo de la relación en el árbol, pulsar con clic derecho sobre él y añadir los ‘associations’ a través de la opción ‘New Child’. Las propiedades para los ‘association ends’ aparecen también en la pestaña de abajo.



La opción ‘New Sibling’, que se puede observar en la imagen anterior, sirve para introducir un nodo *hermano* del nodo en el cual se ha hecho clic derecho. Al hacerlo, se introduciría un nodo al mismo nivel que el seleccionado, es decir, como hijo del nodo ‘Specification’.

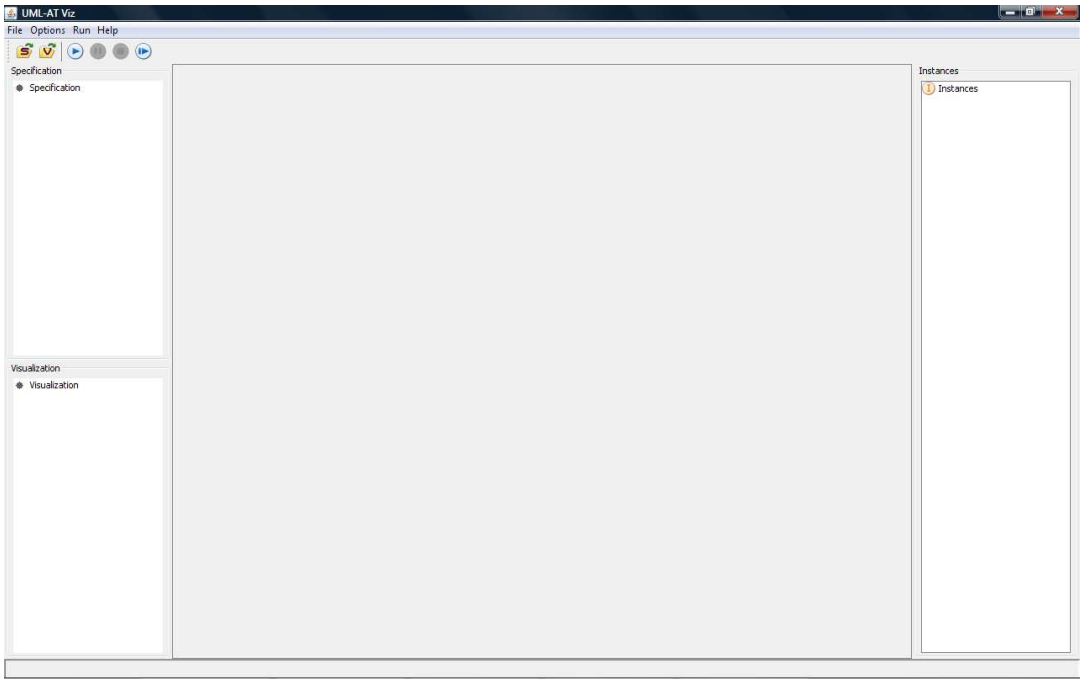
Una vez que hayamos construido el modelo, deberemos salvarlo para poder cargarlo después en la aplicación. Es posible acceder al contenido XML del fichero de la especificación abriendo el fichero de manera externa a Eclipse con cualquier editor de texto como el Bloc de Notas.

8.2. Desarrollo de un nuevo fichero de visualización

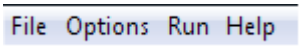
Los pasos a seguir son similares a los descritos para el fichero de especificación, salvo que se parte del proyecto ‘Visualization’ en lugar del proyecto ‘Metalanguage’. El resto de pasos son análogos.

8.3. Manejo básico de la aplicación

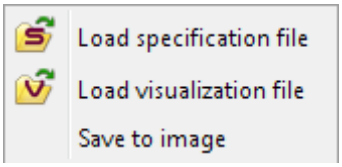
En este punto abordaremos las partes de la aplicación y su manejo. En primer lugar, cuando se arranca el programa, la interfaz gráfica presentada al usuario tiene el siguiente aspecto:



La parte central es la reservada a la visualización y es lo que identificaremos como *Canvas*. En la parte superior de la ventana observamos la barra de menús

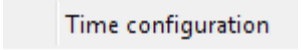


- El menú **File** contiene las siguientes opciones:



- Load Specification File: hace aparecer un cuadro de diálogo para seleccionar el fichero .specification que se pretende ejecutar en la simulación.
- Load Visualization File: hace aparecer un cuadro de diálogo para seleccionar el fichero .visualization que contiene información de visualización.
- Save to image: salva en un fichero de imagen jpg, png o gif el contenido del Canvas.

- El menú **Options** tiene la siguiente opción:



- Time configuration: permite ajustar el número de segundos entre cada paso de simulación cuando se ejecuta con el botón 'Play'.

- El menú **Run** tiene las siguientes opciones:

  - Step by step: ejecuta la simulación paso a paso.
  - Straight: ejecuta la simulación cada cierto tiempo, expresado siempre en segundos, y que se puede configurar desde el menú 'Options'. Por defecto se ejecuta un paso de simulación cada 4 segundos.
  - Pause: pausa la ejecución. Al reanudar con cualquiera de las opciones de 'Play' se parte desde el mismo punto en que se pausó la simulación.
  - Stop: para la ejecución. Al reanudar con cualquiera de las opciones de 'Play' se parte desde el inicio de la simulación.

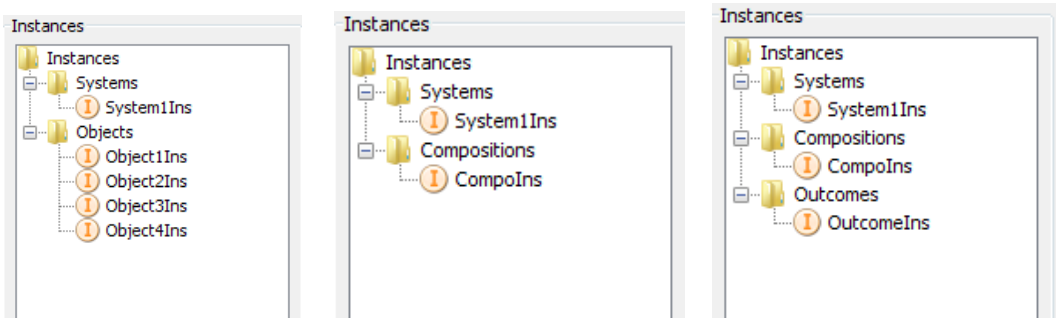
- Por último, el menú **Help** contiene solamente el 'About' (Acerca de…) de la aplicación.

Inmediatamente debajo de la barra de menús encontramos la barra de herramientas. Dicha barra puede 'desprenderse' de su posición original pinchando sobre la zona de la izquierda y arrastrando hacia otro lugar. Los botones de la barra son los siguientes:

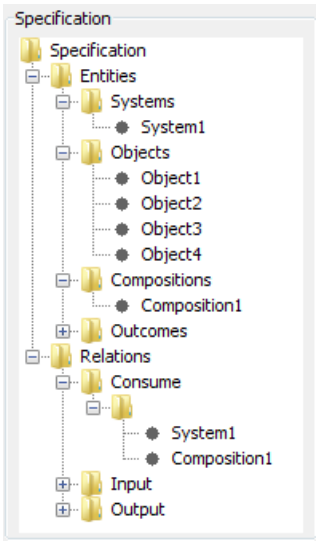
Al pasar el ratón por encima de cada uno de ellos y mantenerlo durante unos instantes, aparece información sobre la utilidad de cada botón. Todos los botones tienen su correspondiente opción en el menú de la aplicación, por lo que vamos a mencionar para qué se usa cada botón, sin explicar a fondo. De izquierda a derecha tenemos:

- Cargar un fichero de especificación
- Cargar un fichero de visualización
- Ejecutar simulación
- Pausar simulación
- Parar simulación
- Ejecutar paso a paso

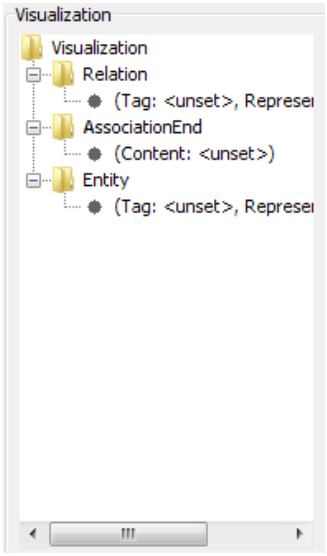
En la parte derecha de la aplicación hay un panel en el que se van mostrando las instancias existentes en cada momento (el estado del mundo) en sucesivos pasos de ejecución. En las imágenes de abajo vemos el estado del mundo en tres momentos distintos de una ejecución:



En la parte de la izquierda de la aplicación hay dos paneles. El panel de arriba corresponde a la especificación del fichero que se ha cargado, de manera que el usuario siempre la tenga a mano. En dicho panel se representan jerárquicamente, en forma de árbol, las entidades y las relaciones del modelo.



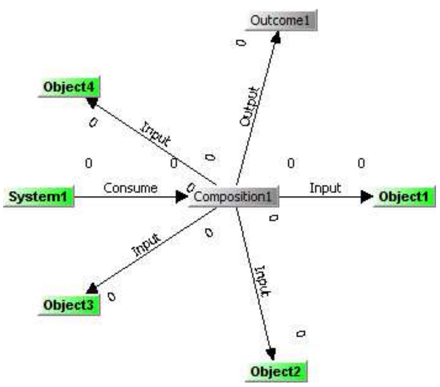
En la parte de abajo se encuentra el panel de la visualización, a través del cual se tiene acceso a la visualización del fichero.

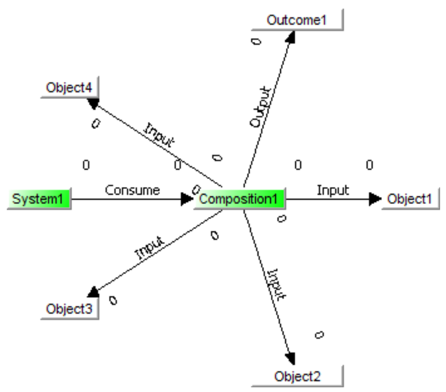


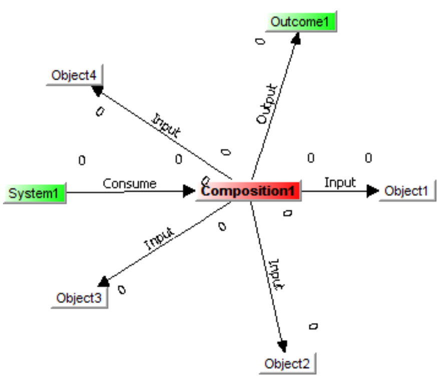
Por último, en el Canvas observamos la visualización concreta. La visualización básica que se incluye con la aplicación está basada en los diagramas UML-AT, donde las entidades se representan mediante *cajas* unidas por flechas correspondientes a las relaciones que las conectan. Hemos introducido un código de colores asociados a la simulación:

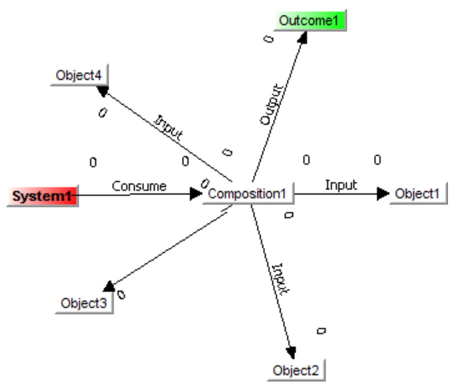
- Antes de ejecutar:
    - <u>VERDE:</u> entidades que existen en el inicio
    - <u>GRIS:</u> entidades que no existen en el inicio
- Durante la ejecución:
    - <u>VERDE:</u> entidades cuyas instancias existen en ese paso de ejecución
    - <u>BLANCO:</u> entidades cuyas instancias no existen en ese momento
    - <u>ROJO:</u> entidades que se han ejecutado (recordemos que sólo pueden ser del tipo 'System' o 'Composition')

En las siguientes imágenes se muestra un proceso de ejecución:









9. Referencias

[1] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vaderheyden, 2004, “Eclipse Development using the GEF and the EMF”, IBM RedBooks.
[2] Iván García-Magariño, Rubén Fuentes Fernández, Jorge J. Gómez-Sanz, 2008, “Guideline for the definition of EMF metamodels using an Entity-Relationship approach”.
[3] Rubén Fuentes, Jorge J. Gómez-Sanz, Juan Pavón, 2008, “Captura del Entorno Social de Sistemas Multiagente”.
[4] Rubén Fuentes, Jorge J. Gómez-Sanz, Juan Pavón, 2004, “Activity Theory for the Análisis and Design of Multi-Agent Systems”.
[5] Gilbert, N. Agent based models. Sage
[6] Edmonds, B: The Use of Models - making. Multi-Agent-Based Simulation, Lecture Notes in Artificial Intelligence
[7] Drogoul et al.: Multi-Agent Based Simulation: Where are the Agents?
[8] http://repast.sourceforge.net/index.html
[9] http://www.swarm.org/index.php/Swarm_main_page
[10] W3C.XML Schema. http://www.w3c.org
[11] Eclipse. Eclipse Modeling Framework (EMF)
http://www.eclipse.org/modeling/emf/
[12] www.jgraph.com
[13] www.jmonkeyengine.com
[14] http://grasia.fdi.ucm.es

AUTORIZACIÓN

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Jonás Fernández Reviejo

Fco. Javier Nieto Espinal

Lara Quijano Sánchez