Un Framework flexible para resolver problemas de optimización mediante sistemas multi-agente

Trabajo de Fin de Máster

David Fernández Jiménez

Máster Universitario en Investigación e Innovación en TIC

Universidad Autónoma de Madrid

Director: Juan de Lara Jaramillo

Curso: 2012/2013

Índice

1 Introducción y motivación 5
2 Antecedentes 8
2.1 Problemas en grafos 8
2.1.1 Problema del vendedor viajero 8
2.1.2 Problema del vendedor viajero generalizado 9
2.1.3 Problema del cartero chino 10
2.1.4 Conversión de problemas en grafos 11
2.2 Agentes y Sistemas Multi-Agente 13
2.2.1 Agente 13
2.2.2 Sistemas Multi-Agente 14
2.3 Inteligencia de Enjambre 17
2.3.1 Introducción 17
2.3.2 Algoritmos basados en hormigas 17
2.3.3 Algoritmos basados en abejas 20
2.4 Ingeniería Dirigida por Modelos 24
2.4.1 Introducción 24
2.4.2 Conceptos y definiciones 24
2.4.3 Meta-modelado en varios niveles 26
2.5 Herramientas y Plataformas 30
2.5.1 Plataformas de simulación de Sistemas Multi-Agente 30
2.5.2 Plataformas para MDE 33
3 Arquitectura 35
3.1 Solución teórica 35
3.2 Solución adoptada 38
3.3 Diseño de la aplicación resultado 41
3.4 Esquema global del sistema 45
3.4.1 Generación del meta-modelo 46
3.4.2 Generación del editor textual para los modelos 47
3.4.3 Generación del código 48
4 Experimentos 50
4.1 Resolución del TSP 50

4.2 Resolución del GTSP ... 56
4.3 Resolución del CPP ... 58
5 Trabajos relacionados ... 60
6 Conclusiones y trabajo futuro ... 64

TABLA DE ILUSTRACIONES

Ilustración 1 – Grafo no dirigido de 5 vértices y 6 aristas 8
Ilustración 2 – Grafo dirigido de 4 vértices y 6 aristas 8
Ilustración 3 – Grafo ponderado 8
Ilustración 4 – Ejemplo de grafo dividido en conjuntos de nodos 10
Ilustración 5 – Esquema de un agente en su entorno (tomada de [8]) 13
Ilustración 6 – Arquitectura típica de un de recuperación de información muti-agente (tomada de [8]) 16
Ilustración 7 – Ejemplo de DSL para una interfaz gráfica de usuario 25
Ilustración 8 – Meta-modelo para el DSL de GUIs 25
Ilustración 9 - Esquema de generación de código con MDE 26
Ilustración 10 - Modelado en dos niveles (tomada de [19]) 27
Ilustración 11 - Metamodelado en tres niveles (adaptada de [19]) 27
Ilustración 12 – Ejemplo con potenciación (tomada de [19]) 28
Ilustración 13 – FlowChart de Repast 31
Ilustración 14 – Arquitectura de un sistema en MASON 32
Ilustración 15 - Solución teórica en 3 niveles 36
Ilustración 16 – Meta-modelo 39
Ilustración 17 – Diagrama de clases de análisis de la aplicación generada 41
Ilustración 18 – Diagrama de clases detallado de la lógica de la simulación 42
Ilustración 19 – Diagrama de clases detallado de los conversores y grafos 43
Ilustración 20 – Estructura de almacenamiento de los conversores 44
Ilustración 21 – Arquitectura de alto nivel de la aplicación 46
Ilustración 22 – Menú contextual para generar el código del modelo 50
Ilustración 23 – Consola de configuración para resolver TSP con ACO (pestaña "About") 51
Ilustración 24 - Consola de configuración para resolver TSP con ACO (pestaña "Model") 51
Ilustración 25 – Grafo de distancias para el TSP 52
Ilustración 26 - Consola de configuración (pestaña "Inspectors") 53
Ilustración 27 – Evolución del coste de la solución encontrada por ACO 54
Ilustración 28 – Consola de configuración para resolver TSP con BSO (pestaña "Model") 55
Ilustración 29 – Evolución del coste de la solución encontrada por BSO 55
Ilustración 30 - Instante de la resolución del GTSP por BSO 56
Ilustración 31 - Grafo de distancias para el CPP 58
Ilustración 32 Tipos de diagramas y sus relaciones en MASDK (tomada de [43]) 61
Ilustración 33 – Captura de pantalla de TSPAntSim (tomada de [45]) 61
Ilustración 34 - Meta-modelo de las soluciones 65

1 INTRODUCCIÓN Y MOTIVACIÓN

En la actualidad el software se ha convertido en pieza clave de la sociedad, ya que está presente en prácticamente todos los ámbitos. Esta creciente necesidad de software hace que las técnicas clásicas de desarrollo a veces no funcionen bien ya que ahora se requiere una mayor celeridad en el desarrollo, manteniendo la calidad. Las técnicas de desarrollo dirigido por modelos (o MDE, del inglés Model Driven Engineering) [1] se basan en generar y explotar modelos del dominio de la aplicación permitiendo una mayor reutilización y un incremento en la velocidad de desarrollo. En este trabajo se ha estudiado la forma de aplicar estas técnicas de MDE sobre problemas de optimización. El objetivo es aprovechar el mayor nivel de abstracción que nos proporciona el enfoque dirigido por modelos para construir un entorno donde poder resolver problemas de optimización de forma rápida y flexible.

De entre la gran variedad de problemas de optimización se ha centrado el interés en aquellos representables mediante el uso de grafos [2]. Se ha elegido este tipo de problemas ya que mediante grafos se pueden representar problemas en un gran número de ámbitos distintos (distancias entre ciudades, flujos eléctricos y de agua, redes de ordenadores, sociogramas, etc.).

Esta versatilidad de los grafos hace que el número de problemas de optimización que se pueden representar con ellos sea muy amplio. Como el intentar implementar cada uno de los algoritmos que soluciona cada problema tendría un coste inmenso, se investigaron otras formas de poder darles solución. Si se consigue transformar un problema del cual no se conoce (o no se ha implementado) la solución, a otro cuya solución sí sea conocida, seríamos capaces de obtener una solución a nuestro problema original interpretando la solución obtenida en el problema conocido, para obtener la solución del problema original.

Aplicando técnicas de MDE a esta idea, se podría conseguir un sistema en el cual un usuario pueda modelar la solución de su problema especificando la manera de realizar las transformaciones a otro problema para el cual ya se conoce la forma de resolverlo. El poder resolver problemas mediante transformaciones resultaría mucho más rápido y cómodo que implementar el algoritmo concreto.

Para que las transformaciones puedan realizarse, debe existir un conjunto de problemas de optimización que sí tengan implementada una solución. A la hora de implementar una solución hay que tener en cuenta que muchos de los problemas de optimización en grafos son muy costosos de resolver, normalmente de complejidad NP-completa [3]. Esto implica que para grafos relativamente grandes, tratar de resolver uno de estos problemas de forma exacta sea inabarcable. Para paliar este problema surgen las técnicas heurísticas, las cuales permiten encontrar soluciones de forma más rápida, a cambio de que la solución no sea la óptima, sino que únicamente esté cerca de ella.

De entre las heurísticas existentes se centró el interés en las que se basan en la inteligencia de enjambre (o en inglés, *swarm intelligence*) [4]. Estos métodos de inteligencia de enjambre se basan en el comportamiento colectivo de ciertos sistemas descentralizados y auto-organizados, los cuales están formados por un conjunto de agentes que a partir de un comportamiento basado en reglas simples consiguen, interactuando con el resto de agentes y con su entorno, que emerja un comportamiento global complejo. Estos sistemas tienen su origen en la observación del comportamiento de algunos animales, como por ejemplo, la forma de buscar alimento de las hormigas y las abejas o la manera sincronizada con la que se mueven los bancos de peces. En base a estos comportamientos se ha desarrollado algoritmos que se emplean para resolver un gran número de problemas como por ejemplo, la búsqueda de caminos mínimos, el entrenamiento de redes neuronales, y problemas de maximización, entre otros [4].

En los algoritmos de inteligencia de enjambre se sitúan un cierto número de agentes en un determinado espacio y a través de las interacciones entre ellos y con su entorno se consigue obtener una solución. Dado el enfoque descentralizado de la solución y la autonomía de sus partes, la implementación más adecuada para esta clase de algoritmos es a través de un sistema multi-agente (o MAS, del inglés Muti-Agent System). Un sistema multi-agente es una red débilmente acoplada de agentes, entidades autónomas con objetivos propios, que interactúan consiguiendo realizar alguna tarea que escapa a sus capacidades individuales.

En este punto, ya se puede describir el planteamiento global del sistema: a partir de técnicas de MDE se implementará un entorno capaz de solucionar problemas de optimización, por medio de técnicas de swarm intelligence implementadas mediante un sistema multi-agente. Definido el enfoque del sistema, se estudiaron las distintas alternativas para su implementación.

Para el desarrollo dirigido por modelos se decidió usar Eclipse Modelling Framework (o EMF), ya que proporciona un entorno integrado que a través de sus plugins y herramientas permite definir meta-modelos, instanciar modelos de forma gráfica y textual y crear generadores de código. Para el desarrollo del MAS, se decidió usar la librería MASON. Esta librería de propósito general está desarrollada en Java y facilita la creación de MAS, proporcionando además facilidades para la visualización de los sistemas generados.

Se plantea conseguir un entorno donde, a partir de un lenguaje de dominio específico [1] se permita definir qué tipo de problema se quiere solucionar, el grafo sobre el cual se plantea el problema y el método a utilizar para resolverlo. Una vez definido el problema, el sistema deberá ser capaz de generar el código necesario para implementar un MAS que, con métodos de swarm intelligence, de solución a dicho problema.

Los principales retos que el sistema planteado presenta son:

- Desarrollar un sistema multi-agente que consiga resolver problemas de optimización mediante algoritmos de swarm intelligence.
- Conseguir resolver problemas de optimización mediante transformaciones a otros problemas.
- Aplicar técnicas de MDE para conseguir que el sistema se adapte en función de los detalles concretos que instancie el usuario.

La estructura del resto del trabajo es la que se muestra a continuación. En la sección 2 se explican los fundamentos teóricos en los que se apoya el sistema. A continuación, en la sección 3, se presenta la arquitectura de la aplicación donde se incluye el meta-modelo de nuestro dominio, así como un esquema global de funcionamiento y el diseño de cada una de las partes. En la sección 4 se encuentran los experimentos realizados con los distintos problemas de optimización. Para finalizar, en la sección 5 se presentan algunos trabajos relacionados y por último en la sección 6 se muestran las conclusiones y el trabajo futuro.

2 ANTECEDENTES

En esta sección se explicarán las bases teóricas necesarias para la comprensión del sistema desarrollado. En primer lugar se explicará la forma de representar problemas mediante grafos y las posibles conversiones entre problemas. Después se explica qué es un agente y un sistema multi-agente, para posteriormente definir la inteligencia de enjambre (o swarm intelligence) y explicar algunos de sus algoritmos. Seguidamente se expondrá en qué consiste la ingeniería dirigida por modelos (o Model-Driven Engineering) para concluir con un estudio de las herramientas y plataformas identificadas para el desarrollo de sistemas multi-agentes y de ingeniería dirigida por modelos.

2.1 Problemas en grafos

Podemos definir un grafo como un conjunto de vértices unidos entre sí a través de aristas. De manera más formal diremos que un grafo G es un par ordenado $G=(V,E)$, donde V es un conjunto de vértices y E es un conjunto de aristas que relacionan dichos vértices. Dependiendo de si las aristas de un grafo tienen o no una dirección definida hablaremos de grafos dirigidos o no dirigidos.

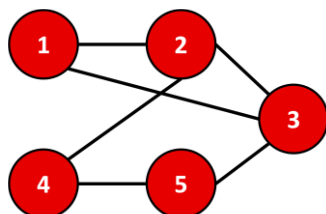

Ilustración 1 – Grafo no dirigido de 5 vértices y 6 aristas

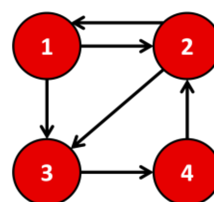

Ilustración 2 – Grafo dirigido de 4 vértices y 6 aristas

Si las aristas (dirigidas o no) del grafo llevan asociadas un peso o coste, se dice que es un grafo ponderado.

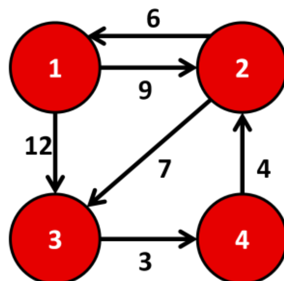

Ilustración 3 – Grafo ponderado

2.1.1 Problema del vendedor viajero

El problema del vendedor viajero (o **TSP**, del inglés Travelling Salesman Problem) se ha convertido en uno de los más famosos en ciencias de la computación. Tiene su origen en

1930 cuando el australiano Karl Menger invitó a la comunidad de investigadores a considerar, desde un punto de vista matemático, el siguiente problema cotidiano. Asumiendo que en cierto territorio existen N ciudades, el objetivo es encontrar una ruta que cumpla las siguientes restricciones:

- Debe empezar y terminar en la misma ciudad.
- Debe pasar exactamente una vez por todas las ciudades.
- La distancia del camino debe ser la mínima posible.

La representación de este problema utilizando un grafo ponderado es realmente sencilla: cada ciudad estará representada por un nodo y las aristas representan los caminos entre las ciudades, asignando el peso de la arista en función de la distancia entre ellas. El problema ahora queda traducido a encontrar un camino del grafo que pase una sola vez por cada vértice y tenga el coste mínimo. Existen versiones del TSP para grafos dirigidos (TSP asimétrico) y para grafos no dirigidos (TSP simétrico).

A pesar de la sencillez del planteamiento, la resolución del problema tiene una complejidad este problema es NP-completo, lo cual implica [3]:

- Es un problema complejo que no puede ser resuelto en tiempo polinomial en una máquina de Turing determinista.
- Cualquier otro problema NP-completo se puede transforma a un TSP en un tiempo polinomial.

2.1.2 Problema del vendedor viajero generalizado

El problema del viajante generalizado (o **GTSP**, del inglés Generalized TSP) se define sobre un grafo especial en el cual los nodos están agrupados en m conjuntos de nodos mutuamente excluyentes, es decir, siendo N el conjunto de todos los nodos y S un conjunto de nodos tenemos que $N = S_1 \cup S_2 \cup ... \cup S_m$ con $S_I \cap S_J = 0$ para todo $I, J, I \neq J$. Además, en este tipo de grafo los arcos sólo pueden estar definidos entre nodos que pertenecen a distintos conjuntos, es decir, no puede existir una arista entre dos nodos del mismo conjunto [5]. En la Ilustración 4 se muestra un ejemplo de este tipo de grafo.

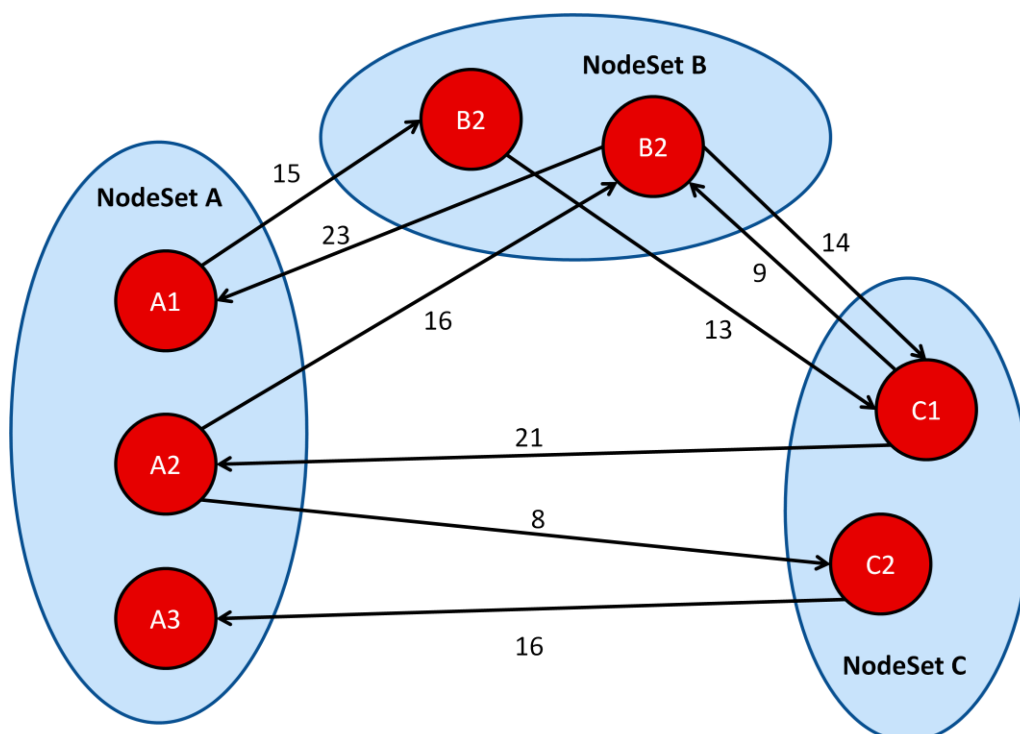

Ilustración 4 – Ejemplo de grafo dividido en conjuntos de nodos

Una vez definido el grafo, se enuncia el GTSP como el problema de encontrar un camino mínimo que incluya exactamente un nodo de cada conjunto de nodos. Es sencillo de observar que el TSP es un caso particular del GTSP en cual únicamente hay un nodo en cada conjunto.

Al ser el GTSP un caso más general del TSP, es intuitivo de ver que su complejidad será también NP-completa.

2.1.3 Problema del cartero chino

La formulación del problema del cartero chino (o **CPP**, del inglés Chinese Postman Problem) es la siguiente: Un cartero debe repartir la correspondencia a cada una de las casas de su distrito teniendo las siguientes restricciones:

- Tiene el mismo punto de origen que de destino (la oficina de correos).
- Para repartir a todas las casas, debe recorrer todas las calles al menos una vez.
- El coste del camino debe ser mínimo.

Para representar el problema con un grafo:

- Se generan tantos nodos como intersecciones entre calles.
- Se genera una arista por cada calle del barrio.
- A cada arista se le asigna un peso en función de la longitud de la calle.

Si el CPP se representa sobre un grafo dirigido, el problema podría verse como que el cartero debe recorrer, en lugar de todas las calles, todas las aceras de todas las calles (pudiendo ser una acera más larga que la otra).

2.1.4 Conversión de problemas en grafos

Ya que la representación de problemas con grafos se puede aplicar a muchos contextos (rutas entre ciudades, redes de ordenadores, redes sociales, etc.), existen una gran cantidad de problemas distintos representables usando esta técnica.

Diseñar una solución para cada problema individual puede ser costoso, por lo que a veces es preferible transformar el problema a otro más general, usualmente al TSP. En [6] se explica cómo transformar problemas de enrutado de arcos (en inglés Arc Routing Problems o ARP) en una instancia del TSP dirigido. Esta conversión consigue resolver el CPP en grafos dirigidos, no dirigidos y mixtos (con aristas dirigidas y no dirigidas).

La **conversión de un CPP a un TSP** explicada en [6] se basa en 3 pasos. Si partimos de un grafo $G = (V, A)$, debemos:

1) Sustituir las aristas no dirigidas de A por dos aristas dirigidas, una con cada sentido. El grafo (ahora ya dirigido) resultante lo llamaremos $\bar{G} = (V, \bar{A})$.

2) El segundo paso es convertir $\bar{G}$ en una instancia del GTSP. Para ello
   a. Se genera un nodo del GTSP por cada arista perteneciente a $\bar{A}$.
   b. Cada nodo generado estará en su propio conjunto de nodos, salvo aquellas parejas de nodos que provengan de la misma arista no dirigida de A. En este caso, esos dos nodos del GTSP estarán en el mismo conjunto.
   c. Para cada nodo de cada conjunto se crean aristas no dirigidas a todos los nodos del resto de conjuntos. El peso asignado a cada arista será igual a la distancia mínima entre los nodos de G a partir de los que fueron generados.

3) Por último, se convierte la instancia del GTSP obtenido en 2) en un TSP.

La explicación de cómo **convertir el GTSP a un TSP** se explica en [5]. Esta conversión consta de tres pasos:

1) Se asigna un orden arbitrario a los nodos de cada conjunto, es decir, si C_i es un conjunto de nodos e $i \epsilon C_i$, se ordenan obteniendo $i_1, i_2, i_3 .. i_n$. Ahora se crea un ciclo dirigido de coste 0 en función de dicho orden. Esto da lugar a las aristas $(i_1, i_2), (i_2, i_3) \ldots (i_n, i_1)$, todas ellas de coste 0.

2) A cada nodo i_j del conjunto C_i se le asignan las aristas de su sucesor en el orden generado, i_{j+1}, salvo la que forma el ciclo de coste 0 con el resto de nodos del conjunto.

3) Se ajusta el coste de las aristas del grafo. Si A es el conjunto formado por todas las aristas del grafo y cada arista tiene un coste c_{ij}, su nuevo coste será

$$c'_{ij} = \begin{cases} c_{ij} & \text{si } i,j \text{ pertenecen al mismo conjunto} \\ c_{ij} + \beta & \text{si } i,j \text{ pertenecen a conjuntos distintos} \end{cases}$$

donde $+\infty > \beta > \sum_{(i,j)\epsilon A} c_{ij}$

El generar ciclos de coste 0 dentro de los conjuntos consigue que a la hora de resolver el TSP, cuando se visite un nodo, seguidamente se visiten todos los de su antiguo conjunto antes de visitar un nodo de otro conjunto. Además, como se cambian las aristas de todos los nodos por las de su antecesor en el orden, cuando se finalice el ciclo de coste cero, se dispondrá de las aristas del nodo que ha sido visitado del conjunto.

Estas conversiones nos permiten convertir las soluciones del TSP de los grafos resultados de la transformación en soluciones de los problemas originales. En el caso del GTSP, la forma de obtener la secuencia de nodos que le dan solución a partir de la secuencia de nodos que solucionan el TSP es la siguiente:

1) El primer nodo de la solución al TSP será también el primer nodo de la secuencia que soluciona el GTSP.
2) Se selecciona el siguiente nodo de la secuencia que soluciona el TSP y:
    - Si pertenece a un conjunto distinto que el último añadido a la secuencia con la solución del GTSP, se añade a la secuencia.
    - Si no, se descarta.
3) Volver al paso 2) hasta que se hayan recorrido todos los nodos que solucionan el GTSP.

En el caso del CPP, la secuencia de aristas que lo resuelve se obtendrá de la solución obtenida para el GTSP. Ya que cada nodo del GTSP corresponde a una arista del CPP, para construir la solución del CPP se irán recorriendo los nodos que solucionan el GTSP añadiendo la arista correspondiente. Si llamamos N a la secuencia de nodos que resuelven el GTSP, si el nodo $n_i \in N$ genera la arista (a,b) y el nodo $n_{i+1} \in N$ genera la arista (c,d):

- Si $b \neq c$, antes de añadir la arista (c,d) a la solución del CPP se deberán añadir todas aquellas que enlacen los nodos b y c por el camino mínimo.
- Si $b = c$, no se añade ninguna arista entre arista (a,b) y (c,d).

2.2 Agentes y Sistemas Multi-Agente

2.2.1 Agente

Del mismo modo que sucede con la inteligencia artificial, no existe una definición universalmente aceptada para el concepto de agente, lo cual genera diversos debates y controversia al respecto. En nuestro caso, diremos que un agente es un sistema computacional situado en cierto entorno con el que interactúa de forma autónoma y flexible, con acciones que le permiten conseguir sus objetivos. Esta definición incluye tres conceptos clave [7]:

- **Situación** dentro de un entorno: Implica que el agente tiene ciertos sensores que le permiten obtener información del entorno y que sus acciones también afectarán al entorno.
- **Autonomía**: Que los agentes sean autónomos implica que son capaces de tomar decisiones propias, sin necesidad de la intervención directa de personas o de otros agentes.
- Acciones **flexibles**: Implica que las acciones pueden catalogarse en
    - reactivas: Respuestas rápidas a cambios percibidos en el entorno.
    - proactivas: Acciones iniciadas por el agente que muestran cierto oportunismo u orientación hacia sus objetivos
    - sociales: Interacción con otros agentes o incluso con personas para conseguir sus objetivos y ayudar a otros a que los consigan.

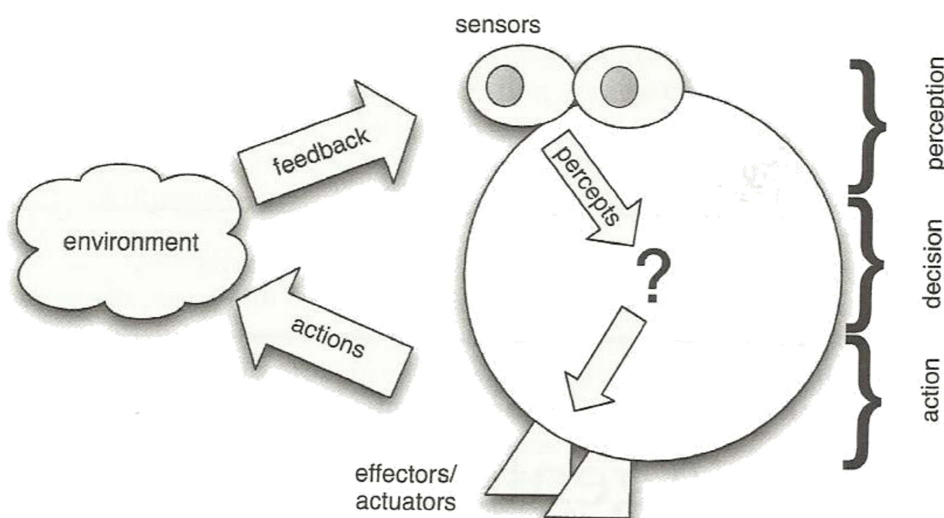

Ilustración 5 – Esquema de un agente en su entorno (tomada de [8])

En otras definiciones se les atribuyen a los agentes propiedades como la de movilidad y adaptabilidad [7] y ciertos investigadores de inteligencia artificial hablan de un concepto

más complejo de agente al cual le otorgan conocimientos, creencias, deseos, obligaciones e incluso emociones [9].

Las personas con nociones de programación orientada a objetos pueden encontrar semejanzas al compararlo con un enfoque basado en agentes, sin embargo existen grandes diferencias entre ambos [8]. Pese a que el enfoque orientado a objetos se basa en la encapsulación y en que un objeto tiene el control de su propio estado interno, la autonomía de los agentes hace que no sólo tengan control sobre su estado sino que también decidan cuándo cambiarlo. Por supuesto, un agente puede cambiar su estado por una petición de otro agente (al igual que un objeto lo cambiaría cuando invocasen a uno de sus métodos), pero este lo hará por su propia voluntad o interés ("Objects do it for free; agents do it because they want to"). Otra importante diferencia es que el grado de autonomía de una agente hace necesario que cada uno tenga su propio hilo de ejecución, situación que no sucede con los objetos [8].

2.2.2 Sistemas Multi-Agente

Podemos definir un sistema multi-agente (o **MAS**, del inglés Multi-Agent System) como una red débilmente acoplada de agentes que, a pesar de que cada uno tiene objetivos propios, trabajan juntos para resolver algún problema que está fuera de su capacidad individual de resolución o su conocimiento.

Los sistemas multi-agentes pueden estar compuestos por agentes de un solo tipo (sistema homogéneo) en el cual todos los agentes tienen las mismas características o bien por varios tipos de agente (sistema heterogéneo) en el cual cada tipo tendrá sus propias capacidades y limitaciones.

Las características principales de un MAS son las siguientes [7]:

- Ningún agente dispone de la información o capacidad suficiente para resolver por sí mismo el problema global.
- No existe un sistema de control global.
- Los datos están descentralizados.
- La computación es asíncrona.

A simple vista podría parecer que un sistema multi-agente no es más que un sistema distribuido. Un MAS puede verse como un caso particular de caso sistema distribuido en el cual las propiedades de los agentes tienen las siguientes consecuencias [8]:

- Ya que cada agente es autónomo y por tanto capaz de tomar sus propias decisiones, la sincronización (si existe) queda relegada a un segundo plano. Esta

situación dista bastante de los sistemas distribuidos en los cuales la sincronización es un aspecto clave tenido en cuenta desde la fase de diseño.
- La interacción que se producen entre dos agentes es interesada, es decir, se produce fruto del interés de cada agente en satisfacer sus propios objetivos. En un sistema distribuido todas las partes comparten un objetivo común y se comunican para conseguirlo.

Los sistemas multi-agente han encontrado aplicaciones en un gran número de dominios distintos. En [8] se explican varios dominios donde los MAS son adecuados. Por ejemplo, en el desarrollo de sistemas de sensores distribuidos. En este tipo de sistemas cada sensor obtiene cierta información parcial, la cual puede entrar en conflicto con la obtenida por otros sensores. Si este sistema se implementa con un MAS en el cual los agentes cooperen intercambiándose sus predicciones y medias, se obtendrá una mejor solución.

Otra de los dominios donde los MAS han tenido ya cabida es en el de la recuperación de información (en inglés Information Retrieval o **IR**). En el ámbito de la IR, nos encontramos que las fuentes de información son heterogéneas y están repartidas en distintas bases de datos y sitios Web. Si el sistema de IR se organiza como se muestra en la Ilustración 6, se puede obtener un sistema más robusto y de buen rendimiento.

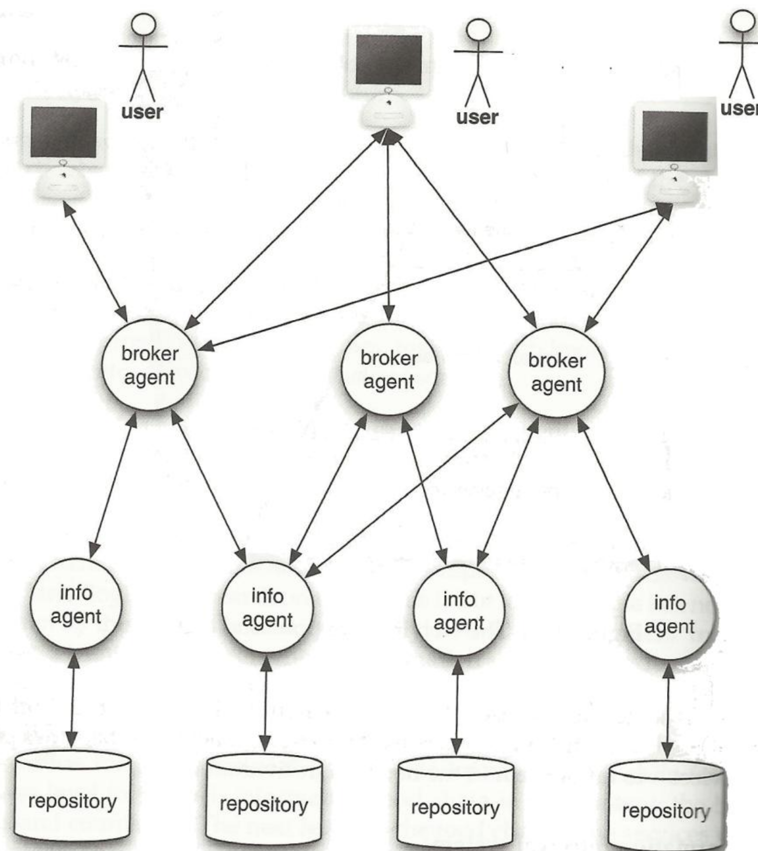

Ilustración 6 – Arquitectura típica de un de recuperación de información muti-agente (tomada de [8])

Existen muchos otros ámbitos donde se aplican los MAS. En este trabajo se han aplicado al ámbito de la simulación social [10]. Uno de los enfoques tras la tecnología de sistemas mutli-agente es la de usar a los agentes como una herramienta con la que experimentar en sociología. Un agente puede utilizarse para representar el comportamiento de un individuo o de una organización o de un conjunto de individuos que actúen como una entidad [8]. En nuestro caso se utilizará para simular los comportamientos sociales de ciertas clases de insectos. Estos insectos, pese a tener unas normas de comportamiento sencillas, mediante la interacción entre ellos y con su entorno, consiguen que el conjunto adquiera un comportamiento global complejo. Estos comportamientos se denominan como inteligencia de enjambre (o en inglés swarm intelligence). El objetivo que se persigue con esta simulación social es el de reproducir esta inteligencia de enjambre para aplicarla a la resolución de problemas de optimización en grafos.

2.3 Inteligencia de Enjambre

La inteligencia de enjambre (o **swarm intelligence**) es una rama de la inteligencia artificial que se basa en el comportamiento colectivo de sistemas descentralizados y auto-organizados. Estos sistemas pueden verse como sistemas multi-agente en los cuales los agentes siguen un comportamiento sencillo pero mediante las interacciones de unos con otros consiguen que emerja un comportamiento global complejo [11].

En los siguientes apartados se explicarán los orígenes de la inteligencia de enjambre y se detallarán algunos de los algoritmos más representativos.

2.3.1 Introducción

La inteligencia de enjambre tiene sus orígenes en la observación del comportamiento social que se produce en las colonias de ciertos insectos tales como hormigas, abejas, avispas y termitas. En estas colonias cada individuo parece tener su propia agenda y aun así las colmenas parecen estar perfectamente organizadas. La integración continua de todas las acciones individuales no parece requerir ninguna supervisión [4]. Esta auto-organización y la formación de inteligencia colectiva (o inteligencia de enjambre) es propiciada por la interacción entre los insectos [12], la cual varía enormemente dependiendo del caso que se estudie.

2.3.2 Algoritmos basados en hormigas

El caso de las hormigas es uno de los más estudiados dentro de la inteligencia de enjambre. Muchas especies de hormigas a la hora de buscar comida siguen un comportamiento basado en marcar el camino para que otras lo sigan. Cuando una hormiga encuentra alimento, carga con él y vuelve al nido para depositarlo. Desde que carga el alimento la hormiga va segregando una sustancia química denominada feromona. Cuando otra hormiga que busca comida encuentra la traza de feromona, sabrá que es un camino hacia la fuente de alimento. La feromona segregada se va evaporando a lo largo del tiempo, lo que implica que los caminos más largos hacia la fuente de comida tendrán menor intensidad de feromona, ya que tomarán más tiempo en ser recorridos. Si existen varios caminos, es más probable que siguiendo el camino con mayor nivel de feromona se llegue a la fuente de alimento por un camino más corto.

En [13] y [14] se explica un sistema de hormigas en dos dimensiones el cual está formado por un nido desde donde parten las hormigas, una fuente de alimento y, entre medias, varios obstáculos. En este sistema las hormigas se sirven de dos tipos de feromonas: una para poder encontrar la fuente de alimento y otra para saber la mejor ruta por la que regresar al nido. Pese a la sencillez del comportamiento de las hormigas, se muestra como estas consiguen encontrar el camino más corto, sorteando los obstáculos, entre el nido y el alimento.

Este comportamiento basado en las feromonas segregadas inspira los Ant Colony Optimization algorithms (**ACO** algorithms) [4]. Una de las aplicaciones más estudiada de estos algoritmos es la de la búsqueda de caminos mínimos. Dentro de este tipo de problemas, el más importante es el problema del viajante (o TSP), que se explicó en la sección 2.1.1 .

Sistema de Hormigas para resolver el TSP

En este sistema, explicado en [4], se generan m hormigas virtuales y cada una de ellas va a construir una solución (aproximada) del TSP, desplazándose de un vértice del grafo a otro hasta que se hayan recorrido todos. Para cada hormiga, el paso de un vértice (ciudad) i a otro vértice j , depende de:

1. Si la hormiga ya ha visitado la ciudad j, no volverá a visitarla (se crea una lista de ciudades tabú). Llamaremos J_i^k al conjunto de ciudades pendientes de visitar por la hormiga k , cuando se encuentra en la ciudad i.
2. La visibilidad de la ciudad, definida como la inversa de la distancia $\eta_{ij} = 1/d_{ij}$. La visibilidad es constante para todo el problema y representa *deseabilidad heurística* de dicha arista.
3. La cantidad de feromona, $\tau_{ij}(t)$, que se encuentre en la arista que une las ciudades. Esta cantidad de feromona se va actualizando e indica la *deseabilidad aprendida* de la ruta.

La probabilidad de que una hormiga k que se encuentra en una ciudad i vaya a la ciudad j tal que $j \epsilon J_i^k$ viene dada por la siguiente regla de transición:

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \epsilon J_i^k} [\tau_{il}(t)]^\alpha \cdot [\eta_{il}]^\beta}$$

En la ecuación, α y β son parámetros ajustables que regulan el peso relativo que se da a la visibilidad y a la traza de feromona. Si $j \notin J_i^k$ la probabilidad de ir a j será 0.

Una vez que una hormiga k ha finalizado el recorrido, depositará cierta cantidad de feromona dependiendo de lo buena que sea su ruta.

$$\Delta\tau_{ij}^k(t) = \begin{cases} Q/L^k(t) & \text{si } (i,j) \in T^k(t) \\ 0 & \text{si } (i,j) \notin T^k(t) \end{cases}$$

Donde

$T^k(t)$ es la ruta encontrada por la hormiga k en la iteración t.

$L^k(t)$ es la longitud de la ruta.
Q es un parámetro ajustable.

Una vez todas las hormigas han finalizado sus rutas, se producirá el decaimiento de la feromona, de la siguiente manera:

$$\tau_{ij}(t) \leftarrow (1-\rho)\cdot\tau_{ij}(t) + \Delta\tau_{ij}(t)$$

Donde $\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t)$. La cantidad inicial de feromona será una cantidad constante τ_0 para todas las aristas del grafo.

Sistema de Colonia de Hormigas para resolver el TSP

Este sistema fue introducido para mejorar el rendimiento del Sistema de Hormigas recién explicado. El Sistema de Colonia de Hormigas consiste en realizar cuatro modificaciones del Sistema de Hormigas [4].

La primera modificación es la variación de la regla de transición para permitir una exploración más explícita. Una hormiga k situada en una ciudad i elegirá la ciudad j en función de la siguiente regla:

$$j = \begin{cases} argmax_{u\in J_i^k} \ \{[\tau_{iu}(t)]\cdot[\eta_{iu}]^{\beta}\} & \text{si } q \leq q_0 \\ J & \text{si } q \leq q_0 \end{cases}$$

Donde

q es una variable uniformemente distribuida en $[0,1]$.
q_0 es un parámetro ajustable $(0 \leq q_0 \leq 1)$.
$J \in J_i^k$ es una ciudad elegida en función de la siguiente probabilidad:

$$p_{iJ}^{k}(t) = \frac{[\tau_{iJ}(t)]\cdot[\eta_{iJ}]^{\beta}}{\sum_{l\epsilon J_i^k} [\tau_{il}(t)]\cdot[\eta_{il}]^{\beta}}$$

La segunda modificación consiste en ajustar la forma en que la feromona se actualiza. En el Sistema de Colonia de Hormigas sólo la hormiga que haya encontrado la mejor solución en cada iteración incrementará la feromona de su ruta (a diferencia del Sistema de Hormigas, en el que todas lo actualizaban). Esto ayuda a obtener exploraciones más directas. La actualización seguirá la siguiente regla:

$$\tau_{ij}(t) \leftarrow (1-\rho)\cdot\tau_{ij}(t) + \Delta\tau_{ij}(t)$$

Donde

Los pares (i,j) son aristas pertenecientes a T^+.
T^+ es el mejor camino encontrado hasta el momento.

$\Delta\tau_{ij}(t) = 1/L^{+}$.
L^{+} es la longitud de T^{+}.

El tercer ajuste consiste en realizar actualizaciones locales del nivel de feromona. Cuando una hormiga k está en una ciudad i y se traslada a la ciudad $j\epsilon J_i^k$, la concentración de feromona del enlace que (i,j) es actualizada mediante la siguiente fórmula:

$$\tau_{ij}(t) \leftarrow (1-\rho)\cdot\tau_{ij}(t) + \rho\cdot\tau_0$$

El valor de τ_0 es el mismo que se asigna inicialmente a todos los arcos el cual tomará un el siguiente valor:

$$\tau_0 = (\mathrm{n}\cdot\mathrm{L_{nn}})^{-1}$$

Donde
n es el número de vértices (o de ciudades).
$\mathrm{L_{nn}}$ es la longitud del camino obtenido por Nearest Neighbor Heuristic [15].

Esta actualización local produce que cuando una hormiga elige una arista el nivel de feromona de la misma disminuye, fomentando que otras hormigas realicen búsquedas por aristas no visitadas aun.

El cuarto y último ajuste es la utilización de una lista de ciudades candidatas. Esta lista consiste en lo siguiente.

- Cada ciudad tendrá su lista de ciudades candidatas.
- Esta lista tendrá una longitud determinada por el parámetro cl .
- El listado de candidatas de una ciudad está formada por las ciudades más próximas a ella, estando la lista ordenada de forma ascendente en función de la distancia.

Cuando una hormiga se encuentre en una ciudad, visitará en primer lugar las ciudades que estén en su lista de candidatas, visitando el resto únicamente cuando todas las de la lista estén visitadas.

2.3.3 Algoritmos basados en abejas

Del mismo modo que pasaba con las hormigas, estos algoritmos se basan en la manera que las abejas tienen de buscar fuentes de alimento. Durante la búsqueda de alimento, una colmena envía varias abejas exploradoras las cuales tomaran diferentes direcciones. Una vez una abeja vuelve a la colmena y deposita en ella el alimento, realizará la denominada "danza de la abeja" (o "waggle dance" en inglés) con la cual consigue transmitir la siguiente información [16]:

- La dirección en la cual ha encontrado el alimento.
- La distancia que separa a la colmena del alimento.
- La calidad del mismo (en función de su riqueza en nutrientes, azúcares, etc.).

Este baile permite evaluar las distintas rutas que han encontrado cada una de las abejas exploradoras. Cuando una abeja que ya ha bailado vuelve a seguir la ruta que ha descubierto, le acompañarán un cierto número de abejas seguidoras que irá en función de la calidad de la ruta [16].

Este comportamiento origina las heurísticas Bee Swarm Optimization (**BSO**), las cuales se pueden aplicar a un gran número de problemas de optimización. Por ejemplo, en [17] se presenta cómo mediante BSO se puede solucionar el problema de máxima satisfactibilidad ponderada o MAX-W-SAT.

Resolución del TSP con BSO

En este algoritmo, explicado en [18]**,** las abejas virtuales intentan conseguir todo el néctar que puedan, por lo que se asume que la cantidad de néctar es inversamente proporcional a la distancia a la que éste se encuentre. Las abejas buscarán el néctar durante varias iteraciones. Cada iteración está compuesta por varias etapas. Durante una etapa cada abeja recorre s nodos, creando una ruta parcial del TSP. Después de visitar los s nodos de la etapa, las abejas vuelven al nido. Una vez todas las abejas han finalizado una etapa, en función de lo buena que sea su ruta parcial decidirán:

- Continuar con su ruta y reclutar más abejas para que la sigan.
- Continuar su ruta, pero no reclutar a más abejas.
- Abandonar su ruta y seguir a otra abeja.

Durante cada etapa, las abejas eligen el camino a seguir en función de las probabilidades que se asignan, siguiendo el modelo de regresión logística, a cada arco. La probabilidad de que una cierta abeja k, estando en el nodo i se traslade al nodo j durante la iteración z y la etapa $u+1$ será:

- Si $i = g_k(u,z), j \in N_k(u,z), \forall k, u, z$:

$$p_{ij}^k(u+1,z) = \frac{e^{-ad_{ij} \frac{z}{\sum_{r=\max(z-b,1)}^{z-1} n_{ij}(r)}}}{\sum_{l \in N_k(u,z)} e^{-ad_{il} \frac{z}{\sum_{r=\max(z-b,1)}^{z-1} n_{il}(r)}}}$$

- en otro caso:

$$p_{ij}^{k}(u+1,z) = 0$$

Donde

d_{ij} es la longitud del enlace entre los nodos (i,j).
B es el número total de abejas en el nido.
$n_{il}(r)$ es el número de abejas que han visitado el enlace (i,j) en la iteración r.
b es la longitud de la memoria.
$g_k(u,z)$ es el último nodo que la abeja k visitó al final de la etapa u en la iteración z.
$N_k(u,z)$ es el conjunto de los nodos no visitados por la abeja k en la etapa u de la iteración z.
a es un parámetro de entrada ajustable por el analista.

A continuación se explicará en más detalle la ecuación de probabilidad:

- La distancia d_{ij} multiplica a una exponencial negativa, por lo que a mayor distancia del enlace (i,j), menor probabilidad de escogerlo.
- A mayor número de iteraciones (cuando mayor sea z) será mayor la influencia de la distancia frente a la del número de abejas que pasan por un enlace. Es decir, a medida que van pasando iteraciones las abejas van teniendo menor libertad de vuelo.
- El parámetro b indica las iteraciones hacia atrás que una abeja puede recordar el número de visitas que han tenido los enlaces.

Una vez que se finalice la etapa, cada abeja volverá al nido y decidirá si continuar o no con su ruta parcial. La probabilidad de que al principio de la etapa $\mathrm{u}+1$ la abeja k continúe con su ruta parcial viene dada por:

$$p_k(u+1,z) = e^{-\frac{L_k(u,z) - min_{r\in(u,z)}(L_r(u,z))}{u\cdot z}}$$

Donde $L_k(u,z)$ es la longitud de la ruta parcial descubierta por k en la etapa u de la iteración z. Esta ecuación garantiza que la abeja que ha descubierto la ruta parcial mínima de cada etapa continuará con ella.

Si una abeja decide seguir su ruta, debe decidir si va a reclutar o no abejas para que también sigan la ruta que ha descubierto. La probabilidad de que una abeja sigua su ruta sin reclutar a más abejas se nombrará como p^* y dado que las abejas son insectos sociales se tenderá a tomar un valor $p^* \ll 1$.

Las abejas que deciden abandonar su ruta parcial deberán seguir a otra abeja que haya decidido continuar y reclutar. La probabilidad de elegir cada uno de los caminos parciales ofrecidos por las abejas reclutadoras viene dado por:

$$p_{\xi}(u,z) = \frac{e^{\rho\beta_{\xi}(u,z)\,-\,\theta\,\alpha_{\xi}(u,z)}}{\sum_{\tau\in Y(u,z)} e^{\rho\beta_{\tau}(u,z)-\theta\alpha_{\tau}(u,z)}} \quad \xi \in Y(u,z), \forall u,z$$

Donde

ρ, θ son parámetros dados por el analista.

$\alpha_{\xi}(u,z)$ es la longitud normalizada de las rutas parciales.

$\beta_{\xi}(u,z)$ es el número de abejas normalizado que está anunciando su ruta.

$Y(u,z)$ es el conjunto de rutas parciales que fueron visitadas por al menos 1 abeja.

2.4 Ingeniería Dirigida por Modelos

2.4.1 Introducción

En la actualidad, la industria del software se ha convertido en una de las mayores del mundo debido a que éste es pieza clave tanto de la vida cotidiana como en la industria [1]. Estas circunstancias crean la necesidad de que el desarrollo de software deba ser cada vez más rápido y de mayor calidad.

El desarrollo de software dirigido por modelos (en inglés Model Driven Software Development o **MDSM**) tiene como objetivo conseguir [1]:

- Un aumento en la velocidad en el desarrollo del software, propiciado por la generación automática de código a partir de los modelos.
- Un incremento en la calidad del software, debido al uso de transformaciones automáticas y al uso de lenguajes de modelado formales.
- Una mayor capacidad de reutilización, ya que una vez definida la arquitectura y las transformaciones se obtiene una línea de producción de software.

2.4.2 Conceptos y definiciones

La ingeniería dirigida por modelos (en inglés Model Driven Engineering o **MDE**) es una metodología de desarrollo software basada en generar y explotar modelos propios del dominio de la aplicación, en lugar de utilizar modelos informáticos. Esto supone elevar el nivel de abstracción, centrando el foco en el dominio del problema. Una vez se han definido los modelos propios del dominio de la aplicación se pasa al dominio de la solución a través del uso de transformaciones automáticas y generadores de código.

El MDE se basa en la definición de **lenguajes de dominio específico** o **DSL** (de inglés, Domain Specific Languaje). Los DSLs recogen todos los conceptos del dominio de la aplicación. Por ejemplo, si estamos trabajando en el dominio de las interfaces gráficas de usuario (en inglés Graphical User Interface o GUI), un lenguaje específico para nuestro dominio podría contener los conceptos que se muestran en la Ilustración 7.

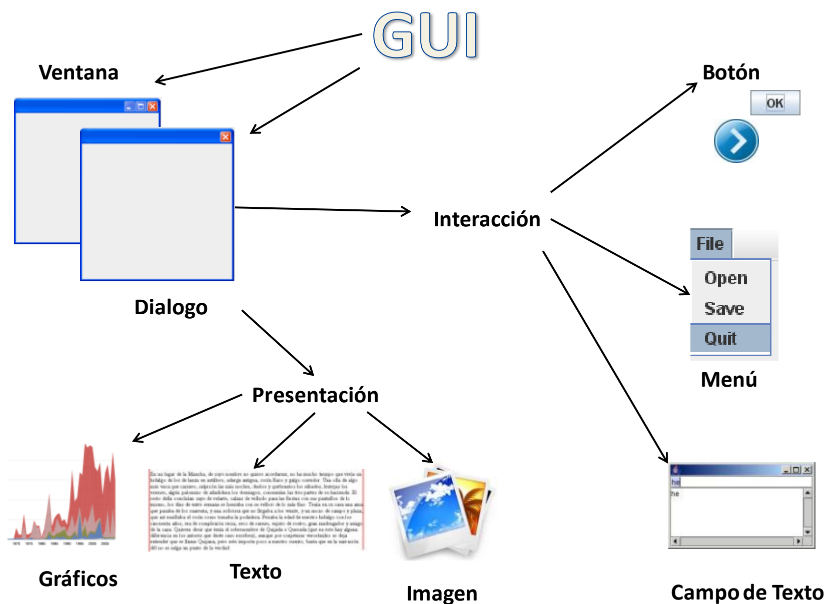

Ilustración 7 – Ejemplo de DSL para una interfaz gráfica de usuario

En el ámbito de MDE decimos que un **meta-modelo** es una descripción de cómo son los modelos y cómo se pueden estructurar. Es decir, un meta-modelo define los conceptos, las relaciones y los atributos propios del dominio de la aplicación. De manera más formal diremos que un meta-modelo define la sintaxis abstracta [1] del DSL. La forma más común de representar un meta-modelo es mediante un diagrama de (meta) clases. En la Ilustración 8 se muestra el meta-modelo correspondiente al dominio de las GUI.

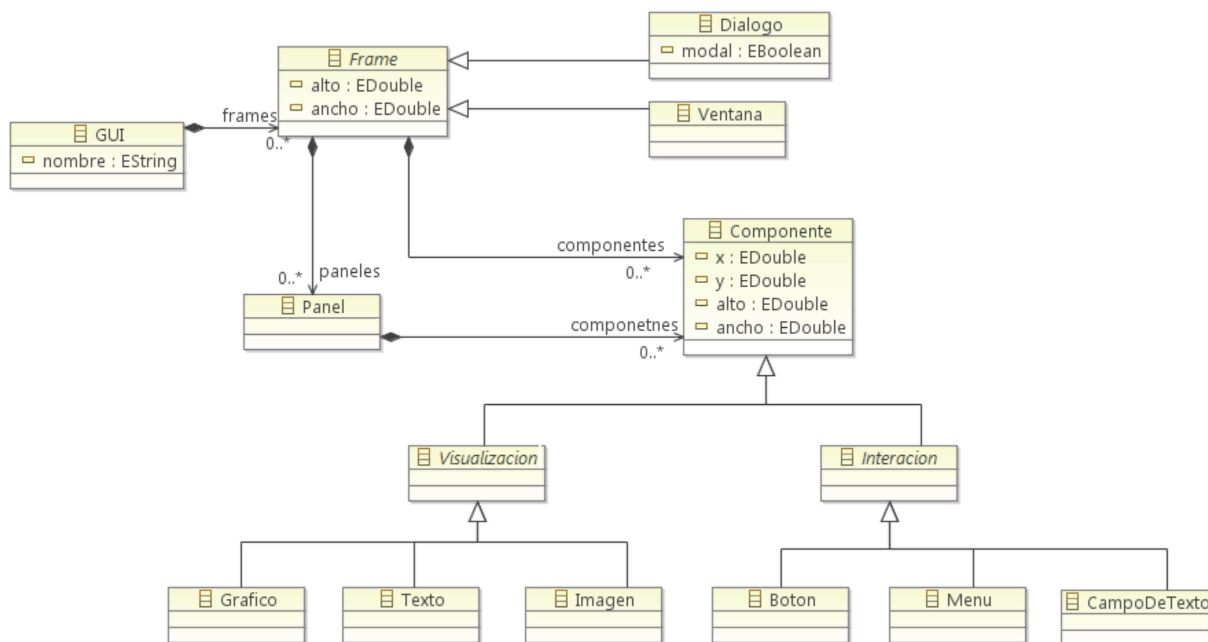

Ilustración 8 – Meta-modelo para el DSL de GUIs

Vemos como un meta-modelo define todas las configuraciones válidas que se pueden dar en nuestro domino. En el ejemplo de las interfaces gráficas de usuario, el meta-modelo definirá todas las posibles interfaces que se puedan crear. Definidas todas las configuraciones posibles, para llevarlo ahora a un problema concreto de nuestro dominio se debe crear una instancia del meta-modelo, es decir, se debe definir **un modelo**. Esta instanciación es similar a la que hay entre un diagrama de clases y uno de objetos.

Una vez se haya definido un modelo, se utilizarán herramientas de MDE para que mediante el uso de generadores de código adaptados a nuestro meta-modelo se genere parte (o todo) el código de la aplicación. En la Ilustración 9 se muestra un esquema de cómo se realiza la generación de código con MDE.

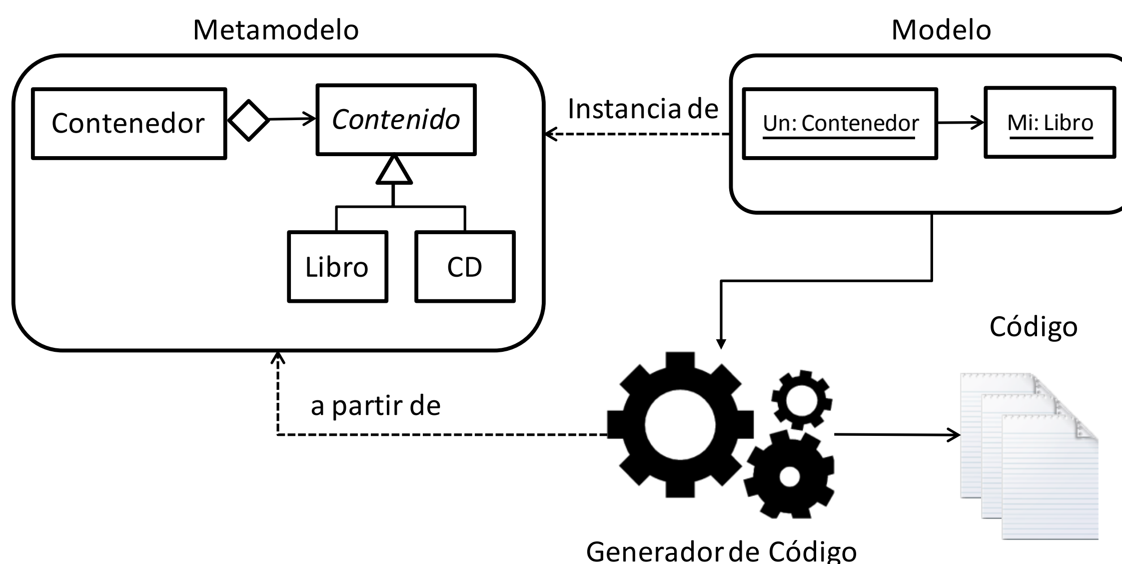

Ilustración 9 - Esquema de generación de código con MDE

2.4.3 Meta-modelado en varios niveles

En el ámbito de MDE, algunos autores han señalado las limitaciones de considerar sólo dos meta-niveles de modelado al mismo tiempo (esto es, meta-modelos y modelos), tanto para la ingeniería del lenguaje como para el dominio del modelado. Un ejemplo muy común para ilustrar esta limitación es el meta-modelado de productos y tipos de productos [19], mostrado en la Ilustración 10.

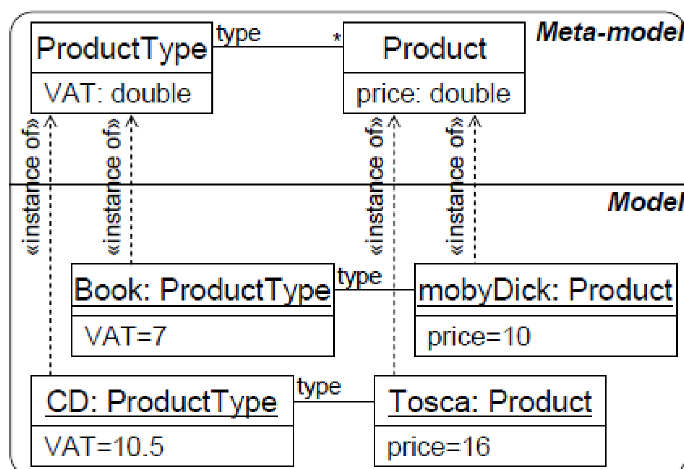

Ilustración 10 - Modelado en dos niveles (tomada de [19])

Esta solución, pese a ser válida, requiere que al hacer el modelo se especifique de forma manual las relaciones de tipado (relación *type* en la ilustración), pudiendo dar lugar a errores ya que no hay forma de chequear si estas son correctas [19]. Si pasamos ahora a un modelado en tres niveles, obtendríamos la solución mostrada en la Ilustración 11.

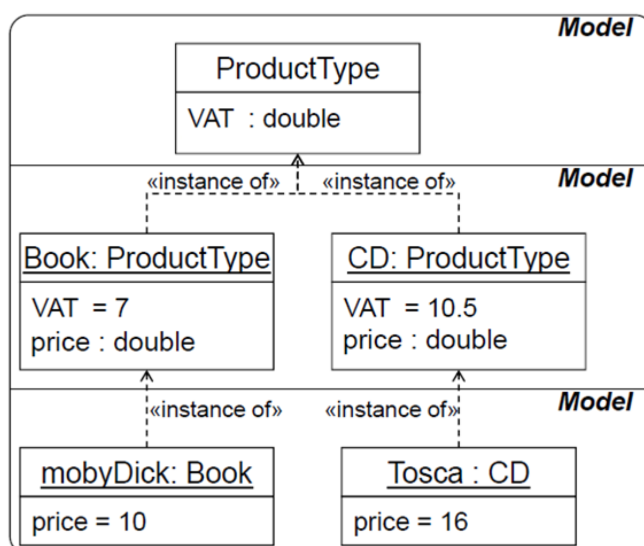

Ilustración 11 - Metamodelado en tres niveles (adaptada de [19])

En este caso, el *ProductType* se ha declarado en el nivel superior y se instancia en dos niveles. En el siguiente nivel se instancia *ProductType* mediante dos tipos de producto (en nuestro caso *Book* y *CD*) asignándoles a cada uno un impuesto distinto (atributo *VAT*). Por último, en el nivel inferior, se instancian los tipos de productos de la clase inmediatamente superior y se le asigna valor al atributo *price* declarado en el nivel superior. Se observa como mediante este modelado en tres niveles se consigue reducir la complejidad accidental del modelado (ahora la clase *Product* ya no es necesaria) [20] consiguiendo manejar de forma más intuitiva las relaciones de tipado.

No obstante, en la Ilustración 11 se observa como el atributo *price* ha sido declarado por todas las instancias de *ProductType,* ya que todas lo necesitaban. Para evitar esta situación, *price* debería estar declarado en el nivel superior de tal forma que se indique que no tomará valor hasta dentro de dos meta-niveles. Para explicar cómo se consigue declarar atributos de esa manera es necesario introducir los siguientes conceptos:

- **Clabjects**: Son elementos que actúan al mismo tiempo como tipo (clase) para el nivel inferior e instancia (objeto) para el nivel superior [20].
- **Potencia**: Es la forma de expresar cuantas veces una propiedad necesita ser definida en los siguientes meta-niveles hasta convertirse en una instancia plana con un valor asignado. La potencia puede asignarse tanto a clases como a atributos y asociaciones [19].

Si ahora, manteniendo el planteamiento en tres meta-niveles, hacemos uso de potencias, obtendremos el meta-modelo de la Ilustración 12.

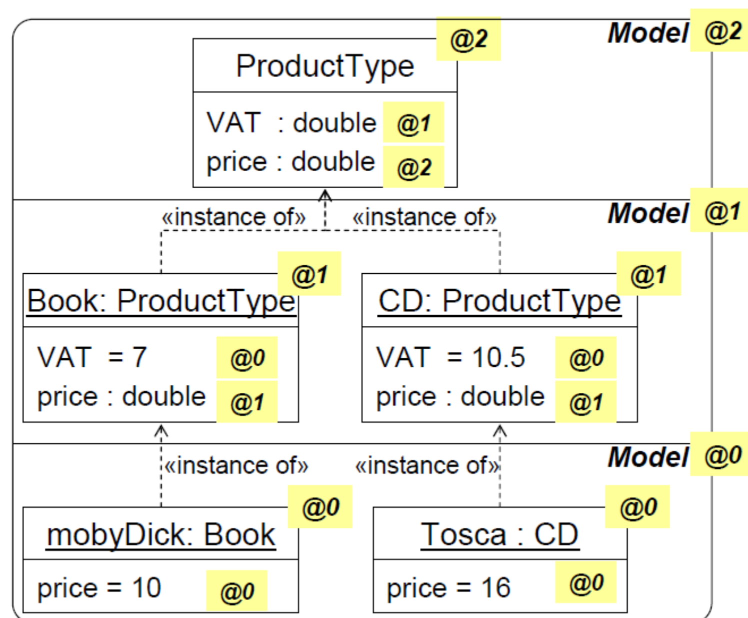

Ilustración 12 – Ejemplo con potenciación (tomada de [19])

Si nos fijamos en el nivel superior de la Ilustración 12, observamos como la clase *ProductType* tiene potencia 2, teniendo sus atributos *VAT* y *price* potencias 1 y 2 respectivamente. Es decir, tendremos que bajar dos niveles para construir una instancia de *ProductType*, pero el atributo *VAT* se ha de terminar de instanciar en el siguiente nivel.

Si bajamos un nivel encontramos los denominados Clabjects (*Book* y *CD*), los cuales actúan como instancia de *ProductType* y a la vez como tipo para el nivel inferior. Se puede observar que la potencia de todos los elementos ha disminuido en uno y que aquellos

elementos que ahora son de potencia 0 (el atributo *VAT* en nuestro caso) ya tienen asignado su valor.

Si nos fijamos en el meta-nivel inferior vemos como *ProductType* ya ha sido instanciado, ya que todos sus atributos tienen asignado su valor.

2.5 Herramientas y Plataformas

El sistema a implementar deberá conseguir que mediante técnicas de MDE se genere un MAS que solucione ciertos problemas de optimización (ya sea de forma directa o transformándolos a otros problemas que sí se sepan resolver). Antes de desarrollar el sistema se valoraron que plataformas utilizar.

2.5.1 Plataformas de simulación de Sistemas Multi-Agente

Existen una gran cantidad de plataformas para la programación de MAS. Con el objetivo de seleccionar una para este trabajo, se realiza un repaso de las plataformas presentadas en [21], atendiendo a:

- Ya que quiere desarrollar un sistema, se descartan las plataformas propietarias centrándose únicamente en las de software libre.
- Se valora que las plataformas sean actuales y/o se actualicen periódicamente.
- Aunque en un grado menor, se valorara positivamente que las plataformas puedan correr sobre varios sistemas operativos.

A partir de estos criterios, se investigó sobre varias de las plataformas encontradas en [21], las más destacadas se introducen a continuación.

La plataforma **Swarm** [22] fue la primera de propósito general y reutilizable orientada a simulación multi-agente. Hasta su aparición, las plataformas existentes eran específicas para un modelo. Swarm fue desarrollada en lenguaje Objetive-C y posteriormente se ha reescrito en Java debido a la complejidad de su aprendizaje. A pesar de que es una plataforma algo antigua, sigue siendo considerada como una de las más potentes.

El lenguaje de modelado de agentes **MAML (Multi-Agent Modelling Language)** [23] fue pensado para proporcionar un nivel de abstracción más que el que proporciona Swarm, para permitir acercar la plataforma a usuarios sin conocimientos profundos de programación. El resultado que se obtiene con el uso de esta plataforma será código ejecutable por Swarm. A pesar de resultar interesante a priori, se descartó el profundizar en esta plataforma ya que no es mantenida desde el año 1999.

La plataforma **AndroMeta** [24], programada en C++, está disponible para distribuciones de Linux y MAC y únicamente es gratis para usos no comerciales o académicos. Una de las propiedades más interesante de esta plataforma es que el usuario no programa directamente el código C++ que se ejecuta, sino que los modelos se implementan en un metalenguaje y la plataforma se encarga de generar el código C++. Los agentes se definen como si fueran objetos para los cuales se deben de especificar ciertos métodos.

Esta plataforma se encontró muy interesante y se decidió experimentar con ella, obteniendo las siguientes conclusiones propias:

- No es sencilla de instalar y poner en funcionamiento.
- Existe poco soporte (contactos de ayuda, foros, etc.) para solventar dudas o problemas.
- Los ejemplos y la documentación son suficientes para implementar modelos sencillos de prueba.
- La versión descargada no genera bien los makefiles necesarios para compilar el código C++ que genera, obligando a una compilación manual por parte del usuario.

La herramienta **Repast (Recursive Porous Agent Simulation Toolkit)** [25] tiene un enfoque similar a Swarm, ya que en sus orígenes fue pensada como una reescritura de Swarm en código Java. Repast ofrece la posibilidad de generar código automáticamente a partir del diseño del comportamiento de un agente mediante un *FlowChart*, tal como se muestra en la Ilustración 13.

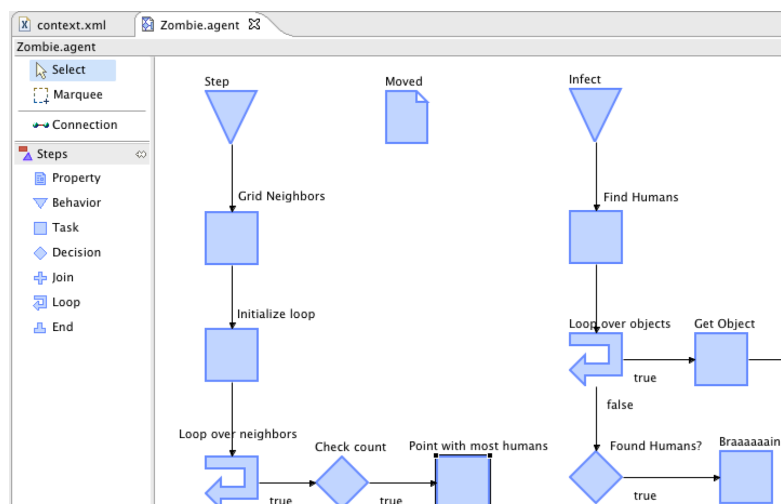

Ilustración 13 – FlowChart de Repast

Esta plataforma tiene bastante acogida y actividad. En marzo del 2012 publicaron dos nuevas versiones de la plataforma:

- Repast Simphony 2.0. Nueva versión, la cual permite implementar los modelos en ReLogo (dialecto de Logo), Groovy y Java.
- Repast for High Performance Computing 1.0.1. Versión desarrollada en C++, que está enfocada para el uso en supercomputadores y clusters de gran escala.

La plataforma **MASON (Multi-Agent Simulation of Neighbourhoods)** [26] consiste en un conjunto de clases Java a partir de las cuales los usuarios implementan sus modelos a través de la especialización o implementación de las clases y/o interfaces.

Esta plataforma es bastante popular y se ha escogido para someterla a un estudio más exhaustivo por contar, entre otras, con las siguientes características:

- Pese a estar desarrollada en Java es bastante rápida.
- Fácil de aprender para usuarios con conocimientos en Java y/o programación orientada a objetos.
- Es una plataforma muy usada y la documentación es adecuada.
- Los modelos son independientes de la visualización (tal como pasaba en la plataforma Swarm) y ofrece posibilidad de visualización en 2D y 3D. En la Ilustración 14 se muestra las principales clases de MASON responsables del modelo (sistema) y a la visualización. La separación entre ellas y el hecho de que las relaciones sean desde las clases de la visualización a las del modelo aseguran la independencia entre modelo y visualización.

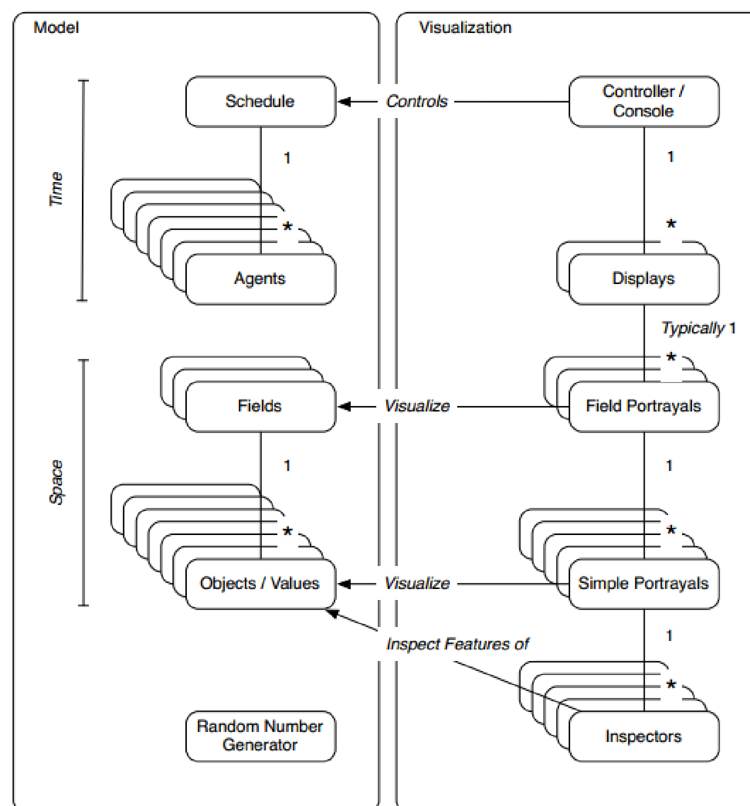

Ilustración 14 – Arquitectura de un sistema en MASON

Tras la instalación y puesta a prueba de la aplicación, se han obtenido las siguientes conclusiones:

- La instalación es muy sencilla ya que basta con incluir la librería (fichero .jar) en el proyecto Java en el cual se quiera utilizar.
- El coste de aprendizaje es bajo y se consiguen hacer modelos relativamente complejos en poco tiempo.
- Gracias a la herencia de Java, los modelos son fácilmente extensibles.
- La interfaz gráfica es muy sencilla de incluir en los modelos y ofrece posibilidades interesantes tales como grafos, histogramas y series temporales.

Estas conclusiones fueron las que llevaron a tomar la decisión de utilizar esta plataforma para generar el sistema multi-agente.

2.5.2 Plataformas para MDE

Para el desarrollo de MDE es muy común la utilización de **Eclipse Modeling Framework (EMF)** [27]. EMF proporciona un entorno que recoge todo el proceso desde el diseño del meta-modelo hasta la generación de código.

En EMF los meta-modelos se definen a partir de ficheros Ecore los cuales pueden ser generados de forma manual o mediante el plugin **Emfactic** [28], el cual lo genera a partir de una sintaxis sencilla.

Para instanciar los meta-modelos, EMF ofrece varias posibilidades:

- De forma gráfica, utilizado el **Graphical Modeling Framework** [29] o **EuGENia** (incluida en el Proyecto Epsilon [30]).
- De forma textual, con la herramienta **Xtext** [31] la cual permite definir lenguajes de manera sencilla.

La generación de código en EMF se realiza mediante lenguajes como **Acceleo** [32] o Epsilon Generation Language (**EGL**) [33]. Ambos lenguajes están basados en plantillas y generan código a partir de un modelo (instancia de un meta-modelo), con la diferencia de que Acceleo implementa el estándar MOFM2T de la OMG [34], mientras que EGL es fácilmente integrable con otros lenguajes Epsilon para la gestión de modelos [30].

Existen otros entornos similares al EMF donde desarrollar MDE, como por ejemplo las **Microsoft DSL Tools** para Visual Studio. Este entorno no se ha considerado más en profundidad por ser menos utilizado y de software propietario.

El framework **metaDepth** [35] es un entorno que se está desarrollando en torno al proyecto METEORIC, apoyado por el Ministerio de Ciencia e Innovación de España. Este

framework está especialmente desarrollado para meta-modelos multi-nivel ya que soporta un número arbitrario de meta-niveles.

MetaDepth está integrado con la familia de lenguajes Epsilon lo que permite manipular modelos con EOL [36], transformaciones modelo a modelo con ETL [37] y generar código con EGL.

Pese a que MetaDepth permite modelar utilizando un número arbitrario de meta-niveles, se decidió desarrollar el sistema con EMF ya que, al estar integrado en Eclipse permite, desde un mismo entorno (mediante sus distintas herramientas y plugins de modelado), instanciar modelos de forma gráfica y textual, realizar transformaciones modelo-a-modelo, generar código, etc.

Para instanciar el meta-modelo se decidió usar un DSL textual generado con Xtext. Las razones que han llevado a elegir Xtext en lugar de un editor gráfico son las siguientes:

- Exceptuando la parte de grafos, el visualizar de forma gráfica los conceptos que maneja el sistema no aporta un valor añadido.
- Pese a que un grafo sí sería más intuitivo definirlo de forma gráfica, también es más costoso y propenso a errores que si se hace de forma textual (por ejemplo, de manera similar a una matriz de adyacencia).
- Xtext cuenta con funciones de autocompletado y detección de errores sintácticos, lo cual hace muy cómoda la definición de modelos.
- Con Xtext se puede conseguir un lenguaje cercano al lenguaje natural que permita entender perfectamente el modelo definido de un solo vistazo.

El generador de código de la aplicación se implementará con plantillas EGL. A parte de contar con mayor experiencia desarrollando con EGL que con Acceleo, se eligió EGL ya que se podría combinar más fácilmente con otros lenguajes Epsilon, si fuese necesario.

La integración de Xtext y EGL con EMF permiten obtener un único entorno dónde:

- Se puedan instanciar los modelos de forma cómoda gracias a la detección de errores sintácticos on-line y al autocompletado que ofrece Xtext.
- Una vez definido el modelo, se invoquen a las plantillas EGL que generen el código de la aplicación a través de un menú contextual.
- Una vez generado el código de la aplicación, este se compile y se ejecute.

3 ARQUITECTURA

El sistema planteado consiste en la utilización de MDE para la resolución de problemas de optimización a través de algoritmos de inteligencia de enjambre implementados en un sistema multi-agente. El sistema deberá proporcionar un entorno donde el usuario sea capaz de describir el problema a resolver y cómo resolverlo. Esa descripción del problema debe ser interpretada por el sistema y generar el código necesario para implementar su solución.

En esta sección se describirá la arquitectura del sistema generado. En primer lugar se describirá el meta-modelo que describe el dominio. Después, se presenta el diseño de la aplicación que debe ser generada y por último se presentará un esquema global donde se detallan las partes del sistema y el flujo de información entre ellas.

3.1 Solución teórica

El sistema que se quiere conseguir mediante MDE debe ser capaz de resolver diversos problemas de optimización en grafos a través de:

- Algoritmos de inteligencia de enjambre.
- Transformaciones a otros problemas de los cuales sí se conozca la manera de solucionarlos.

La solución obtenida se ha realizado con un modelado en tres niveles, tal como se muestra en la siguiente ilustración.

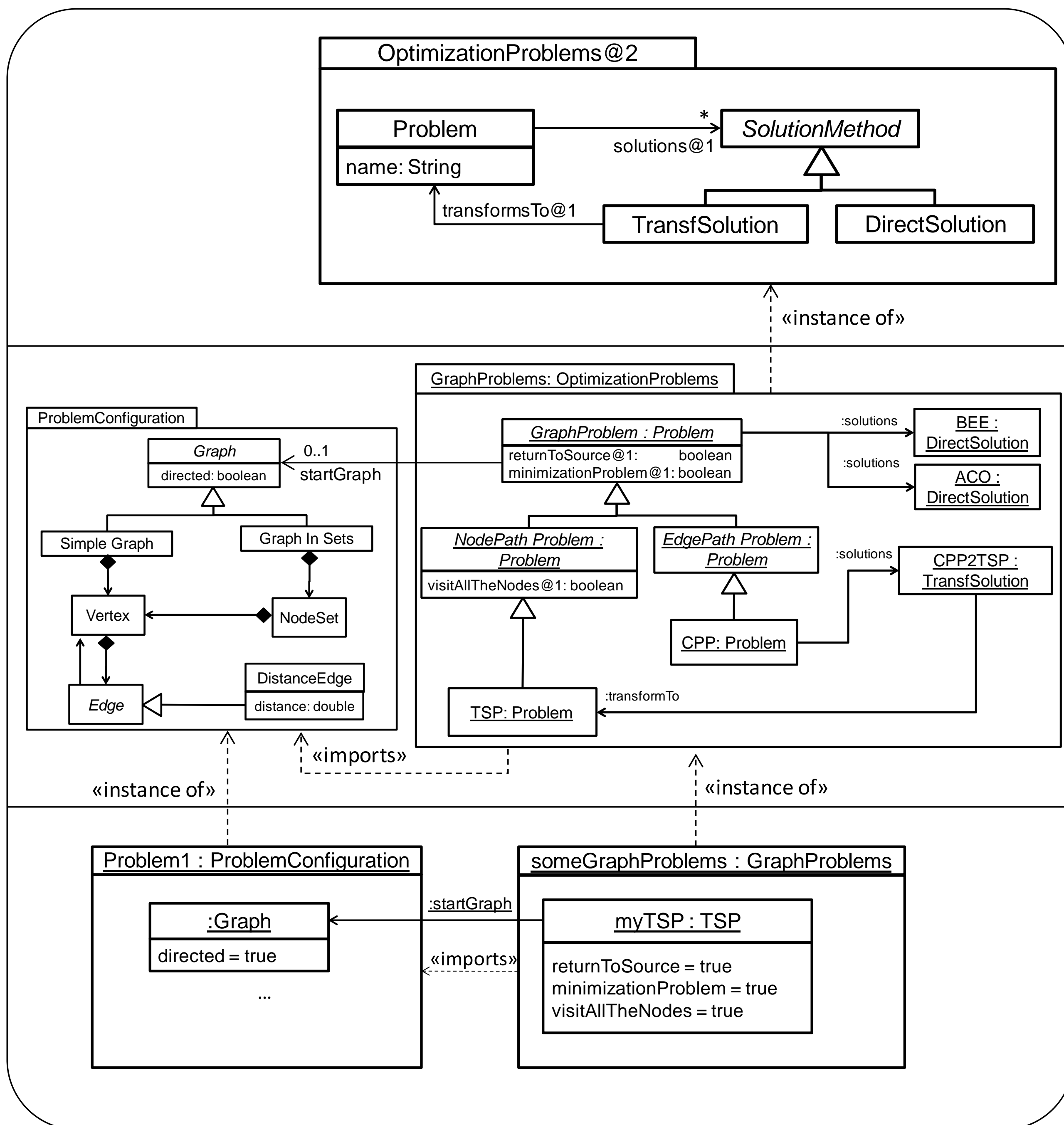

Ilustración 15 - Solución teórica en 3 niveles

En el primer meta-nivel se define el modelo *OptimizationProblem* de potencia 2. En él se incluyen la clase *Problem* y dos alternativas para solucionar el problema (herederas de la clase abstracta *SolutionMethod*): de forma directa o transformándolo a otro problema. Tanto las soluciones a un problema (relación *solutions*) como el problema al que se

transforma (relación *transformsTo*) tienen potencia 1, lo cual implica que deberán de ser instanciadas en el siguiente meta-nivel.

En el siguiente nivel encontramos el modelo que define la configuración del grafo con potencia 1. Este modelo contiene las clases necesarias para poder definir todos los problemas en grafos que se han identificado. En este nivel encontramos también el Clabject *GraphProblem*, que implementa *OptimizationProblem*. En este modelo se incluyen las soluciones que darán a un problema, es decir, se modelará si un problema se resuelve de forma directa, o mediante una o más transformaciones a otros.

En el último nivel se fijan los atributos pendientes para el problema (en nuestro caso, si debe regresar al origen, si es un problema de minimización y si debe visitar todos los nodos) y se modela el grafo del problema.

Este enfoque en tres niveles permite que, disponiendo de ciertas soluciones directas a problemas, el usuario pueda generar soluciones a nuevos problemas a través de transformaciones a los problemas que ya sepa resolver. Para que esto sea viable, el usuario deberá especificar cómo se hacen las transformaciones que ha modelado (generando el código o mediante otros medios).

3.2 Solución adoptada

La solución teórica a la que se llegó está diseñada en un meta-modelo de tres niveles. Ya que el sistema se va a desarrollar con Eclipse Modeling Framework [27], el cual sólo permite el modelado en dos niveles, se tuvo que ajustar la solución teórica.

Se decidió crear un meta-modelo en dos niveles en el cual:

- Las meta-clases de potencia 2 ahora pasarían a ser meta-clases abstractas.
- La instanciación de la meta-clase de potencia 2 ahora se traduce en herencia de dichas clases. Es decir, en lugar de instanciar *DirectSolution* o *TransforSolution* el meta-modelo tendrá clases que hereden de ellas.
- Se generan un número simbólico de configuraciones que permitiesen demostrar la manera de solucionar problemas de forma directa y mediante transformaciones a otros problemas.

Esta adaptación nos permite implementar nuestra solución con EMF pero a cambio implica que si en un futuro se quiere implementar un nuevo problema o una nueva solución se debe cambiar el meta-modelo para que incluya las nuevas clases necesarias, en lugar de instanciarlo dinámicamente como se haría en el diseño en 3 niveles.

A continuación se presenta el meta-modelo, donde se muestran ya las posibilidades de soluciones directas y por transformación que ofrece el sistema.

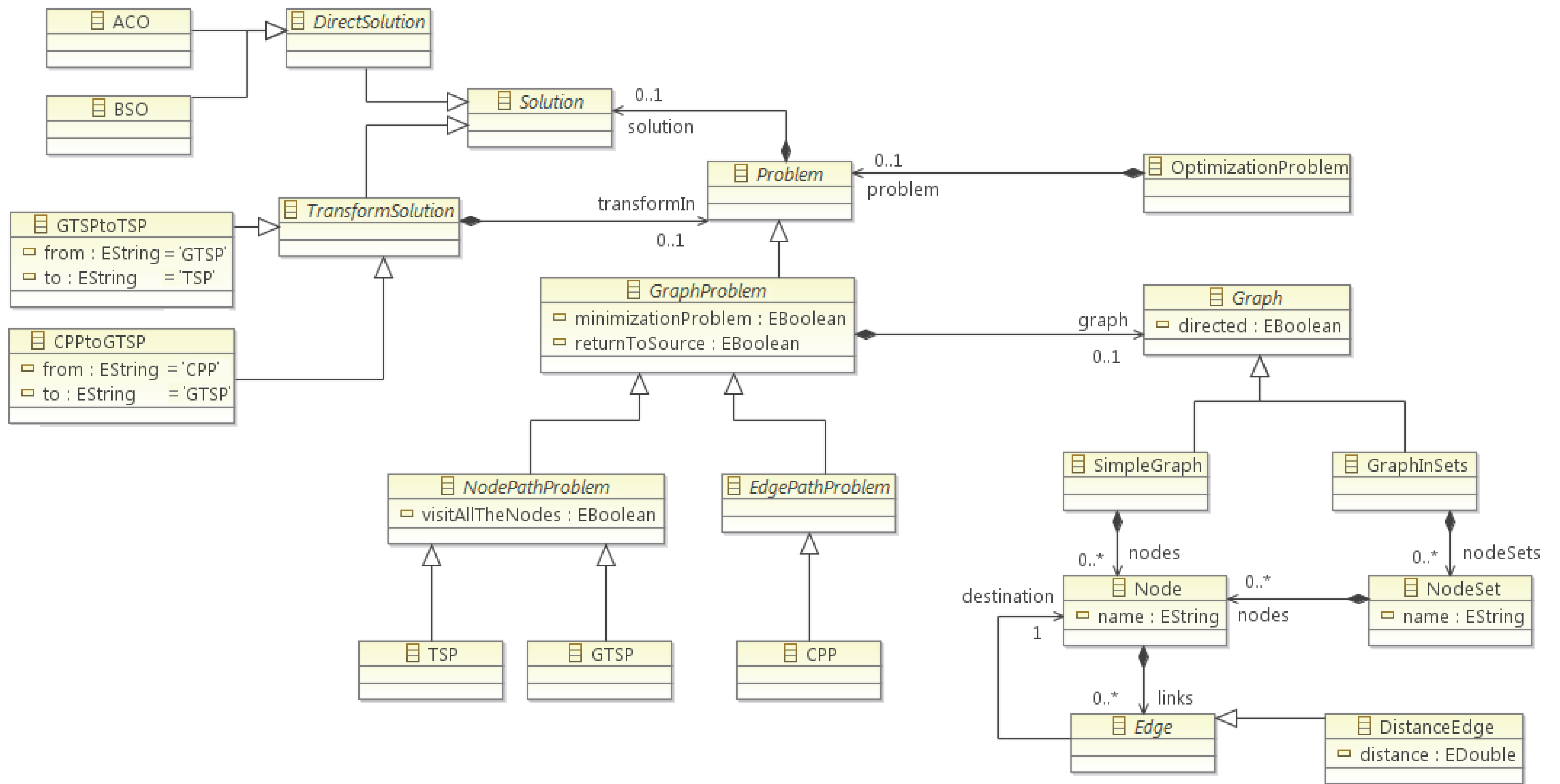

Ilustración 16 – Meta-modelo

Como se observa en el meta-modelo, la meta-clase raíz es *OptimizationProblem,* la cual tiene una referencia a la abstracta *Problem*. Tal como pasaba en la solución en 3 niveles, de la clase *GraphPhroblem* hereda una clase para problemas relacionados con recorrido de nodos (*NodePathProblem*) y otra para problemas de recorrido de aristas (*EdgePathProblem*). Las herederas de estas últimas son los tipos de problemas contemplados en el estudio: TSP, GTSP y CPP.

De nuevo, de manera similar a como indica el meta-modelo en 3 niveles, la clase *Solution* es heredada por *DirectSolution* y *TransformSolution*. Se han implementado dos soluciones directas, ACO y BSO, las cuales resuelven el TSP. Para el resto de problemas las soluciones permitidas se basan en transformación: el GTSP se resolverá mediante una conversión al TSP y el CPP mediante una transformación al GTSP (el cual deberá ser transformado al TSP).

En la sección 2.1 se han explicado el TSP, el GTSP y el CPP como problemas cerrados, es decir, que no dependen de ninguna variable. Sin embargo, en el meta-modelo se muestra como las clases que representan dichos problemas tienen algunos atributos. Estos atributos se han querido añadir para hacer que el sistema sea capaz no solo de resolver los problemas básicos, sino también de poder resolver sus variaciones. A continuación se explica el efecto que se consigue con cada uno de los atributos definidos:

- El atributo *minimizationProblem* determina si el problema a resolver será de minimización o de maximización.
- El atributo *returnToSource* indica si una vez recorrido todos los vértices o aristas se debe regresar al origen.
- El atributo *visitAllTheNodes* (sólo definido para problemas de recorrido de nodos) indica si la solución pasa por recorrer todos los nodos o sólo un subconjunto de ellos. En caso de que tome el valor *false,* en la interfaz gráfica de la aplicación se seleccionarán los nodos que debe recorrer.

En cuanto a los algoritmos ACO y BSO, explicados en las secciones 2.3.2 y 2.3.3 respectivamente, dependían de algunos parámetros configurables. Sin embargo, si nos fijamos en sus correspondientes meta-clases, vemos que no hay ningún atributo que los contemple. Esto se debe a que se decidió que dichos parámetros fuesen ajustables en tiempo de ejecución (a través de la interfaz gráfica), en lugar de hacerlo en la fase de modelado. De esta forma no es necesario instanciar otro modelo para cambiar alguno de los parámetros.

Aunque el sistema no otorga toda la libertad que sería deseable (ya que como se ha comentado, para añadir otra solución o problema se necesitaría cambiar el meta-modelo), sí muestra ofrece las posibilidades buscadas: solucionar problemas de forma directa y convertir los problemas a otros que se sepan solucionar ya sea de forma directa o de nuevo con otra transformación.

3.3 Diseño de la aplicación resultado

En este apartado se explicará el diseño de la aplicación resultado de la generación de código. A continuación se muestra un diagrama de clases de análisis donde se pueden ver las clases principales y sus relaciones.

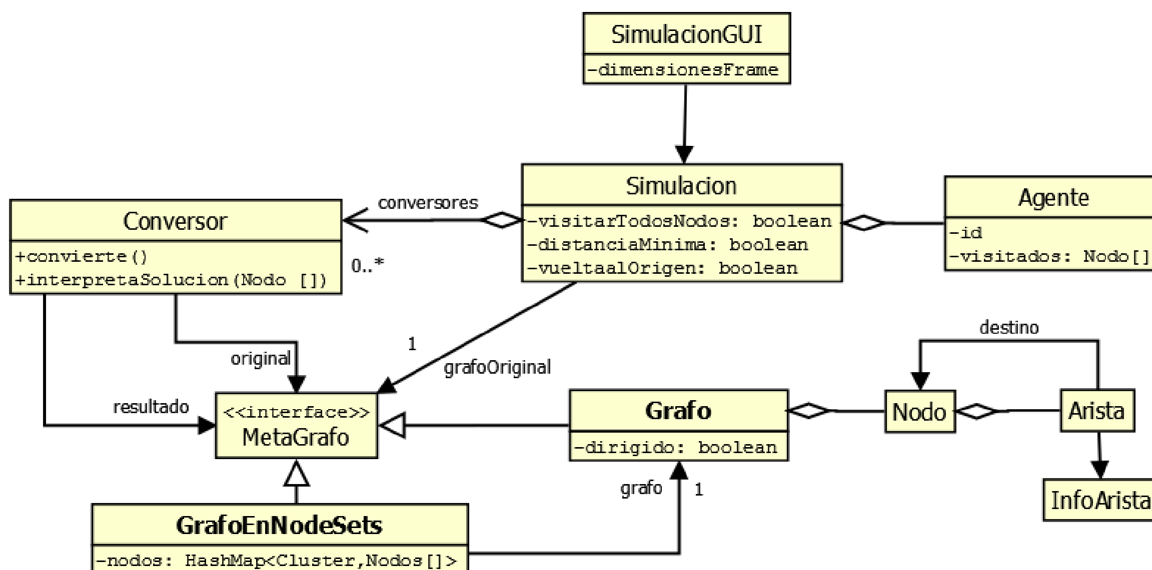

Ilustración 17 – Diagrama de clases de análisis de la aplicación generada

La clase central del diagrama es la clase *Simulacion*. Esta clase es la que carga el peso de la simulación ya que contiene una agregación de agentes, los conversores (si existen) y el grafo que contiene el problema a resolver.

Los atributos de la clase *Simulacion* que se muestran en la Ilustración 17 son asignados mediante la generación de código que realizan las plantillas EGL, a partir del modelo instanciado. Se puede observar también como la clase *Simulacion* está referenciada por la clase encargada de implementar la Interfaz Gráfica de Usuario (GUI).

A continuación se profundizará en el diseño de los aspectos claves de la aplicación. En primer lugar se explicará la implementación de las clases responsables de la lógica de la simulación. El diagrama de clases de la Ilustración 18 hace énfasis en dicha parte.

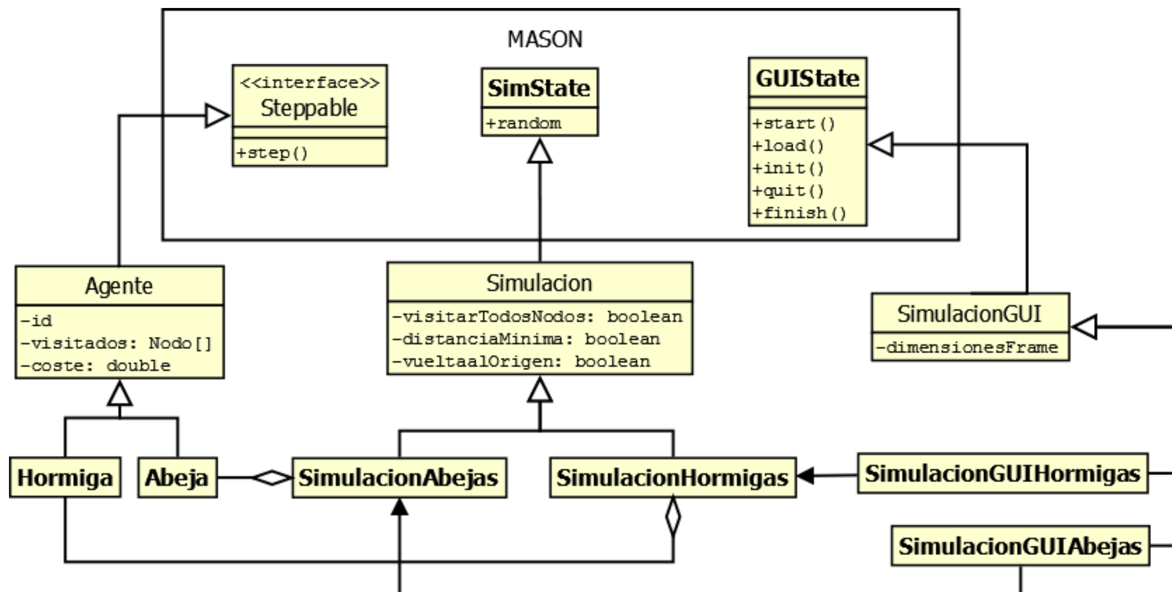

Ilustración 18 – Diagrama de clases detallado de la lógica de la simulación

En el diagrama se observan las clases más importantes de la simulación, así como sus relaciones tanto entre ellas como con las clases de la librería MASON.

- La clase *Agente* implementa la interfaz de MASON *Steppable*. Esta interfaz obliga a implementar el método *step()*, que determina el comportamiento de cada agente en cada paso de la simulación. La clase *Agente* es abstracta, siendo heredada por las clases *Hormiga* y *Abeja*.
- La clase *Hormiga* implementará el comportamiento de las hormigas que se indica el algoritmo Ant Colony descrito en la sección 2.3.2.
- De forma análoga, la clase *Abeja* implementará el comportamiento de las abejas que se describe en el apartado 2.3.3 .
- La clase abstracta *Simulación* tiene como herederas a las clases *SimulacionAbejas* y *SimulacionHormigas*. La clase *Simulacion* hereda de la clase de MASON *SimState*, siendo esta el motor de las simulaciones ya que es la encargada de planificar la ejecución de los distintos agentes a lo largo de las iteraciones.
- La clase de MASON *GUIState* es heredada por la abstracta *SimulaciónGUI* (que a su vez es heredada por *SimulacionGUIHormigas*, *SimulacionGUIAbejas*). Esta clase es la encargada de implementar la interfaz gráfica que se mostrará al usuario.

Se pensó en este diseño para que la aplicación fuese fácilmente extensible ya que si en un futuro se quisiesen implementar nuevos modelos de simulación bastaría con crear clases que hereden de las abstractas *Agente, Simulacion y SimulacionGUI* e implementar sus métodos.

Otro de los aspectos que se quiere explicar más en detalle es el del diseño de la parte de los grafos y los conversores. El siguiente diagrama de clases muestra las clases que lo componen y sus relaciones.

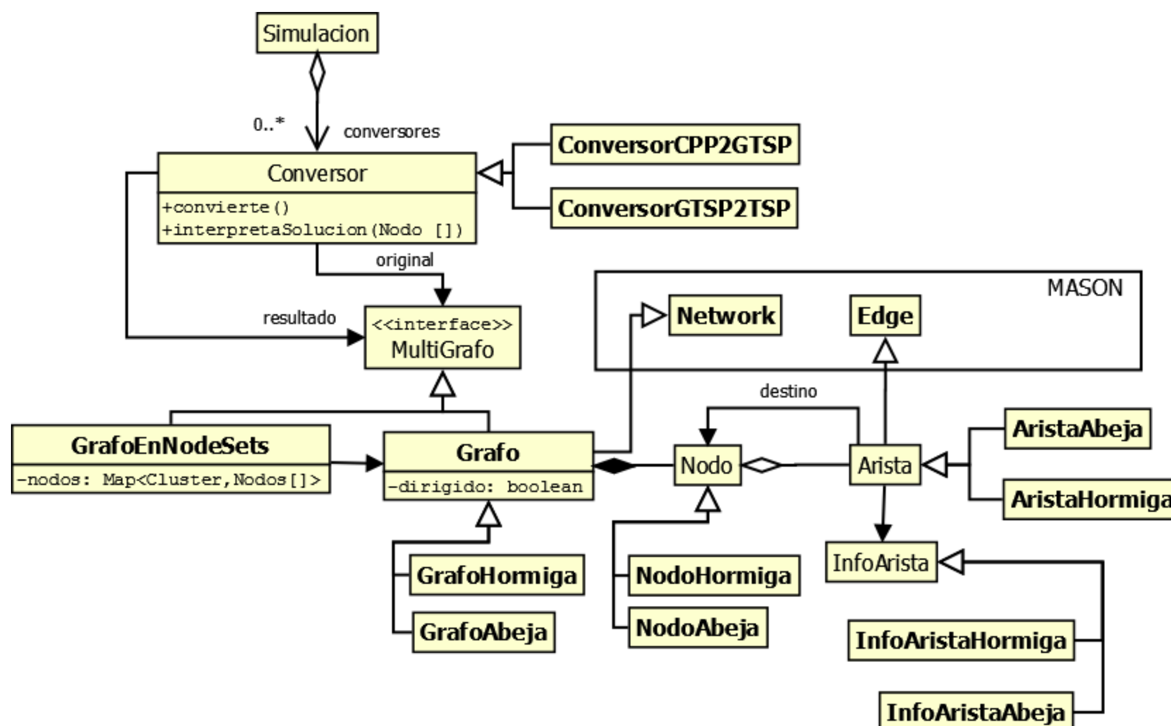

Ilustración 19 – Diagrama de clases detallado de los conversores y grafos

Los algoritmos utilizados para convertir de una instancia de un problema a otro son los explicados en la sección 2.1.4. Cada uno de ellos se implementa especializando la abstracta *Conversor,* lo cual les obliga a:

- Implementar el método *convierte()*, que transforma el problema de un tipo a otro.
- Implementar el método *interpretaSolucion(),* el cual recibe como argumentos la traza de nodos que soluciona el problema convertido y devuelve la traza de nodos del problema original.
- Mantener una referencia al grafo original y al convertido.

Dado que una simulación puede contar con varios conversores, estos se ordenarán de tal manera que siempre se sepa como trasladar una solución encontrada en el problema final al grafo del problema original. En la Ilustración 20 se muestra un esquema de la estructura de almacenamiento de los conversores.

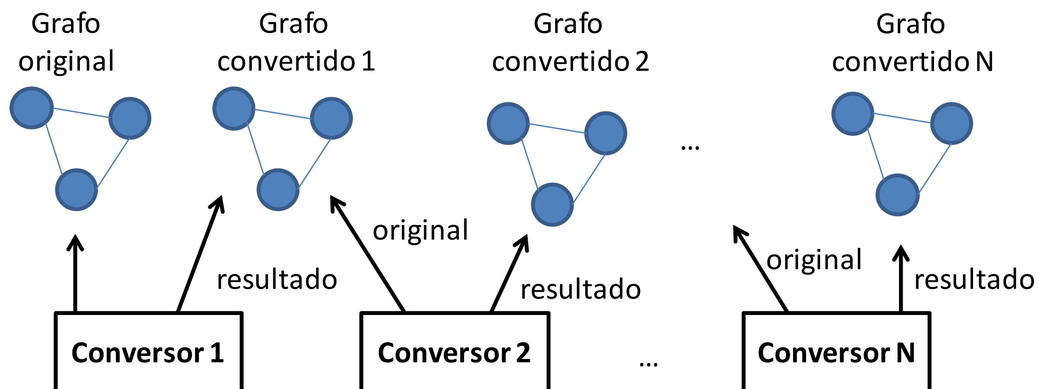

Ilustración 20 – Estructura de almacenamiento de los conversores

En cuanto a las clases que implementan el grafo cabe destacar:

- La clase *Grafo* hereda de la clase *Network* de MASON. Se decidió heredar de *Network* ya que esta clase implementa funcionalidades para añadir/eliminar/referenciar nodos y aristas en el grafo y permite que MASON pueda pintarlo.
- Las clases *Grafo*, *Nodo* y *Arista* son especializadas para que las recorran abejas y hormigas. Esto es debido a que cada uno de los agentes necesita guardar información distinta (en el caso de las hormigas se almacena la feromona, mientras que en el caso de las abejas se necesita llevar la cuenta del número de ellas que ha pasado por cada arista durante las últimas rondas).

3.4 Esquema global del sistema

Desde la definición del meta-modelo hasta la generación del código que implementa la solución se suceden distintas fases. En cada una de dichas fases se utilizan diversas herramientas de modelado y se genera cierto flujo de información. El diagrama de la Ilustración 21 recoge toda esta información dividiéndola en tres grupos:

- Generación del meta-modelo.
- Generación del editor textual para los modelos.
- Generación del código que implementa la solución.

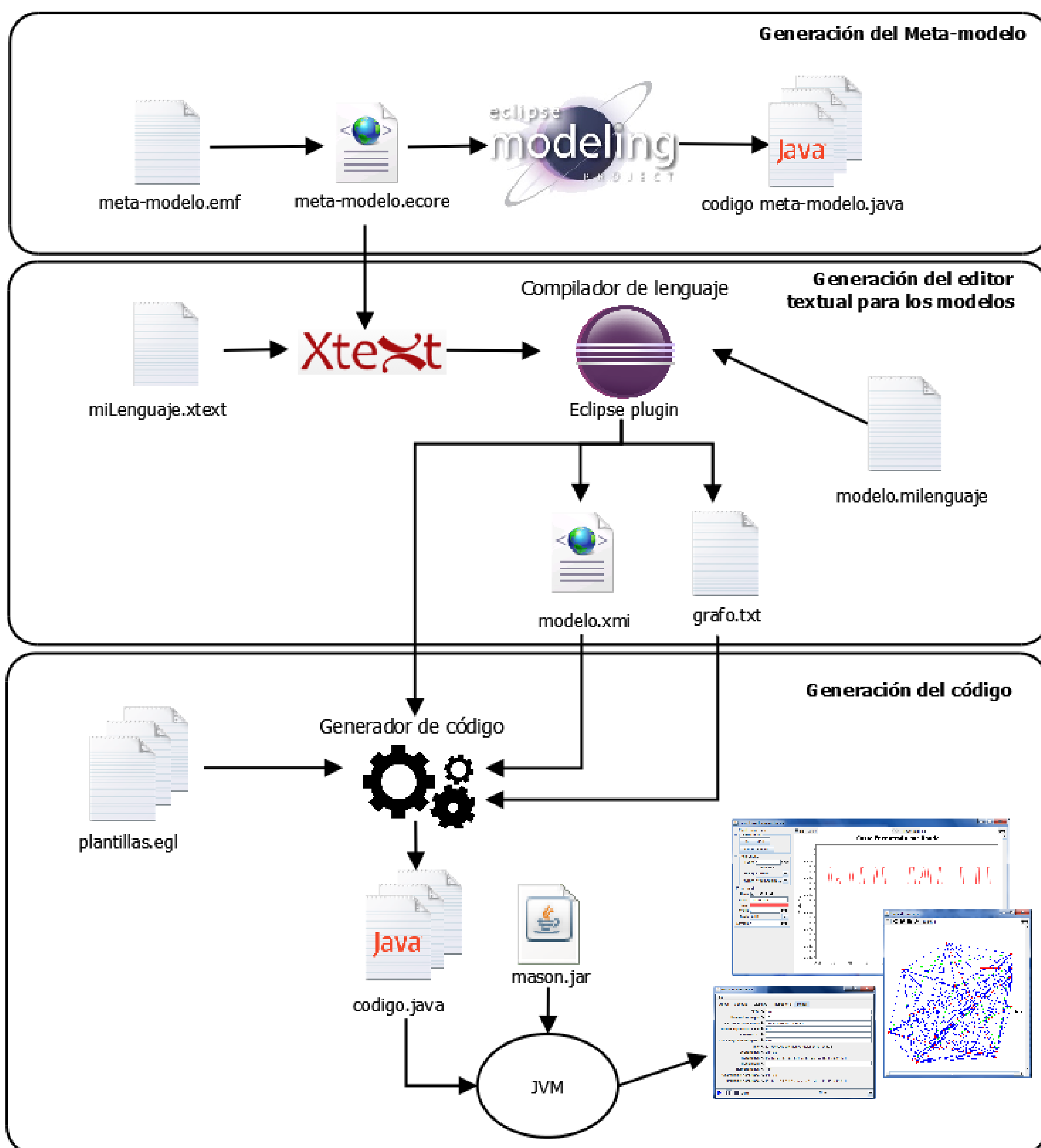

Ilustración 21 – Arquitectura de alto nivel de la aplicación

3.4.1 Generación del meta-modelo

En EMF los meta-modelos se definen mediante ficheros Ecore. Ya que la especificación de un meta-modelo a partir de su edición en un fichero Ecore es engorrosa, se prefirió utilizar EMFactic [28]. Este plugin permite definir el meta-modelo mediante un lenguaje sencillo para posteriormente generar el fichero Ecore.

Una vez se ha obtenido el fichero ecore, se utiliza EMF para generar el código Java del meta-modelo necesario para poder instanciarlo.

3.4.2 Generación del editor textual para los modelos

A partir del meta-modelo definido, la herramienta Xtext [31] genera un lenguaje textual para crear instancias de la misma. Se ha modificado el lenguaje por defecto que genera Xtext para conseguir uno que nos permita describir nuestros modelos de forma más cómoda e intuitiva. A continuación se muestra como se instanciaría en el lenguaje definido un TSP indicando que se resuelva mediante el algoritmo de hormigas:

```
Problem:
    TSP maximization problem , visiting all the nodes and
    coming back to the source
    directed Graph:
        a0: (a1,4.2), (a2,6.2)
        a1: (a0,9.2), (a1,12.2)
        a2: (a0,11.1), (a1,7), (a3,5.4)
        a3: (a1,6.1), (a0,3), (a2,5.5)
Solution:
    ACO algorithm
```

Se observa que la gramática definida es muy cercana al lenguaje natural, permitiendo que con una simple lectura se tenga el modelo que se está instanciado.

Una posible instanciación del GTSP podría ser la que se muestra a continuación:

```
Problem:
    GTSP maximization problem , visiting all the nodes and
    coming back to the source
directed Graph in Sets:
    nodeset1:
        a0: (a1,4.2), (a2,6.2)
        a1: (a0,9.2), (a1,12.2)
    nodeset2:
        a2: (a0,11.1), (a1,7), (a3,5.4)
        a3: (a1,6.1), (a0,3), (a2,5.5)

Solution:
    transform in TSP and use the ACO algorithm
```

Se observa cómo a pesar de que esta vez el problema se resuelve con una transformación, la complejidad en la definición del modelo no aumenta. Vemos también que la definición de conjuntos de nodos se ha resuelto de forma elegante.

El problema del cartero chino quedaría instanciado mediante el siguiente código:

```
Problem:
    CPP maximization problem
    coming back to the source
    directed Graph:
        a0: (a1,4.2), (a2,6.2)
        a1: (a0,9.2), (a1,12.2)
        a2: (a0,11.1), (a1,7), (a3,5.4)
        a3: (a1,6.1), (a0,3), (a2,5.5)
Solution:
    transform in GTSP and
    transform in TSP and use the BSO algorithm
```

Pese a que en los ejemplos presentados se han modelado los grafos sobre los que se solucionará el problema, no especificarlo también es válido. En este caso, cuando la aplicación arranque, el usuario deberá indicar qué grafo quiere utilizar (especificando la ruta del fichero que lo define).

Gracias al uso de Xtext, la gramática definida sólo permite al usuario definir lenguajes válidos, es decir, si se intenta configurar una solución no válida (como, por ejemplo, utilizar la conversión a TSP desde otro problema que no sea el GTSP) se marcará el lenguaje con error de sintaxis.

Como se observa en la Ilustración 21, a partir de este código se genera el modelo (en un fichero XMI) y el grafo que implementa el problema (en caso de que lo haya especificado el usuario).

3.4.3 Generación del código

Una vez se ha definido el meta-modelo y se ha desarrollado un mecanismo para su instanciación, se procedió a desarrollar un generador de código con plantillas EGL.

La aplicación final estará formada por código estático, el cual siempre está presente y no varía, y por código dinámico, la cual dependiente del modelo instanciado. El código estático corresponde al cuerpo de la simulación, a los algoritmos de swarm intelligence y de conversión de grafos, a la interfaz gráfica, etc. A continuación se explica la función del código dinámico generado en cada clase.

- Se utiliza en la clase principal:
    - Para crear el objeto de simulación (*SimulacionHormiga* o *SimulacionAbeja*) que corresponda a la *DirectSolution* indicada en el modelo[1].
    - Si se ha definido un grafo en el modelo, se le asigna a la simulación.

[1] Cabe recalcar que pese a que un problema se puede solucionar seleccionando una o varias transformaciones a otro problema, en última instancia deberá existir un algoritmo directo, siendo en nuestro caso ACO o BSO.

- En las clases de la simulación, el código generado:
  - Inicializa las variables del problema (si debe regresar al origen, si debe recorrer todos los nodos, etc.).
  - Si se ha seleccionado *TransformSolution*, crea una instancia de los conversores necesarios e invoca, por orden, al método que realiza la transformación.
  - Si la solución es una instancia de *TransformSolution*, invoca a los métodos que convierten la solución encontrada en el grafo convertido, a una solución de su grafo original.

4 EXPERIMENTOS

En esta sección se explican algunos de los experimentos realizados con el sistema implementado, para verificar tanto si la conversión de problemas funciona como si los algoritmos ACO y BSO consiguen buenas soluciones a los problemas.

Gracias al enfoque de MDE y al uso de EMF y Xtext, para generar el código que implemente nuestro modelo basta con seleccionar el fichero que lo describe y a través de un menú contextual seleccionar la opción de "Exportar modelo, generar código y ejecutar", tal como se muestra en la Ilustración 22.

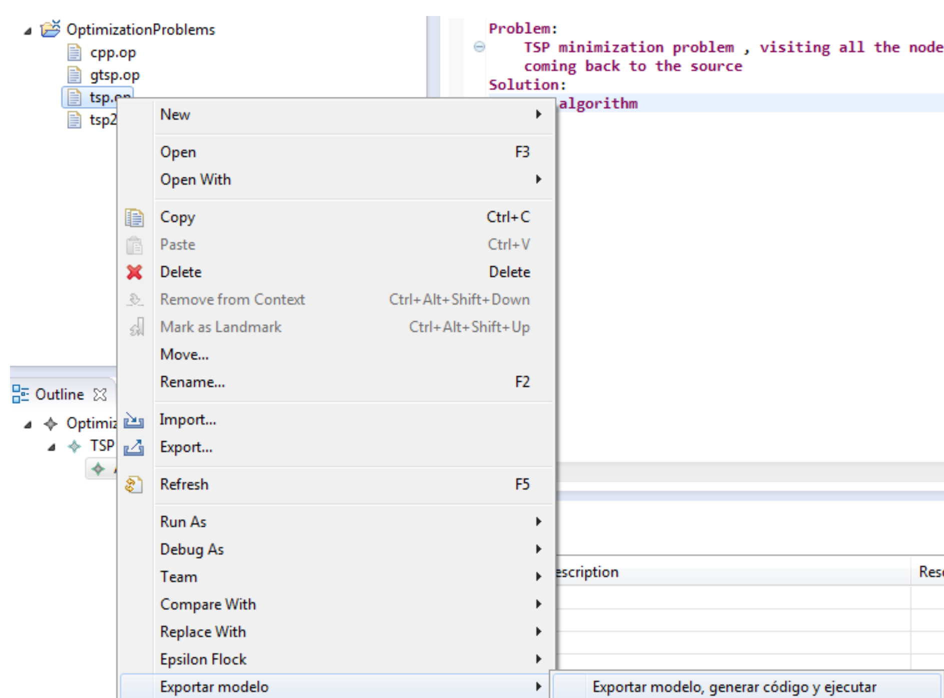

Ilustración 22 – Menú contextual para generar el código del modelo

4.1 Resolución del TSP

Ya que los algoritmos implementados de abejas y hormigas son para resolver el TSP[2], esté será el primer problema a probar. Para probar el TSP se utilizará, en lugar de especificar en el modelo el grafo a seguir, uno de los que se encuentran publicados en [38].

En primer lugar se probará el algoritmo ACO, para lo cual lo seleccionamos en el lenguaje y cuando se genere el código obtendremos la ventana mostrada en la Ilustración 23.

[2] Los algoritmos implementados son suficientemente flexibles para resolver el TSP con las variantes introducidas: que haya que maximizar o minimizar el coste de la ruta, que tenga que regresar o no al origen y que tenga que recorrer todos los nodos o sólo un subconjunto de ellos. En nuestro caso se probará la versión estándar.

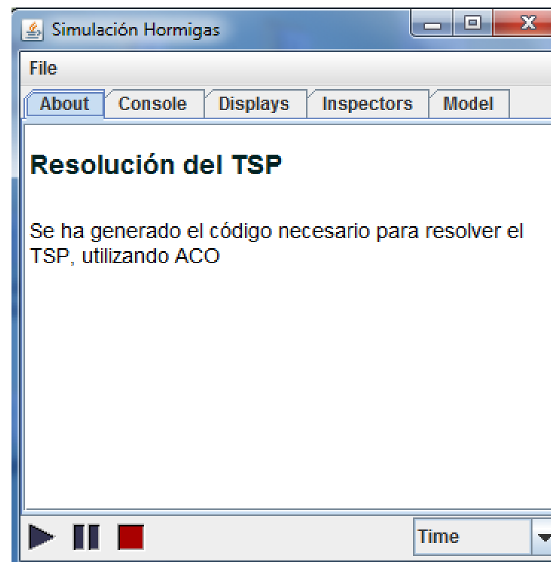

Ilustración 23 – Consola de configuración para resolver TSP con ACO (pestaña "About")

En la Ilustración 23 se muestra la consola de configuración de la simulación. Esta primera pestaña únicamente indica el problema a resolver y el método de resolución. Si pulsamos en la pestaña "Model", podremos ver y editar los parámetros ajustables del algoritmo.

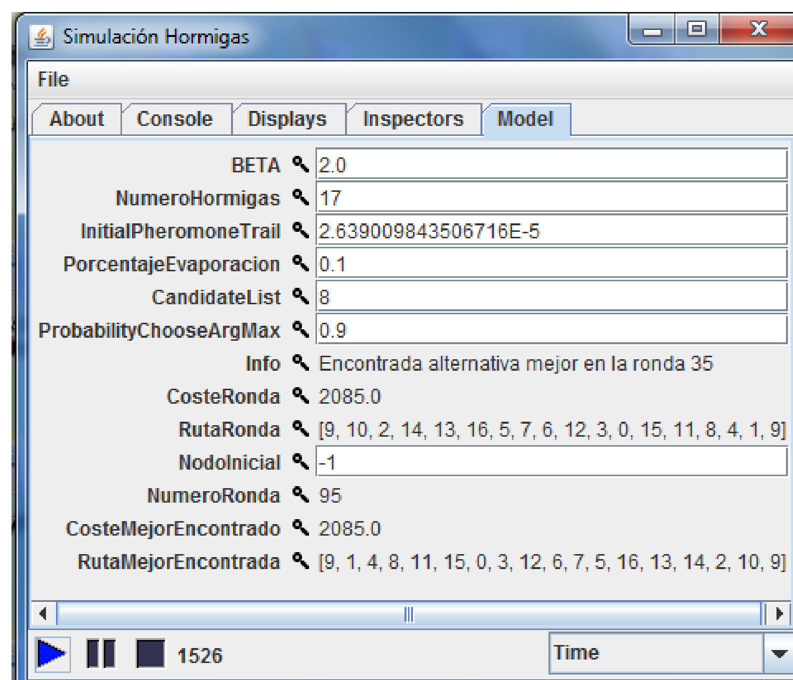

Ilustración 24 - Consola de configuración para resolver TSP con ACO (pestaña "Model")

En la Ilustración 24 se observa cómo la interfaz permite ajustar los parámetros del algoritmo, explicados en la sección 2.3.2. Además, en la parte inferior de esta pestaña se irá mostrando la información relativa al desarrollo del algoritmo:

- Ruta mejor encontrada hasta el momento y su coste.
- Ruta encontrada en la última ronda y su coste.
- Un campo donde se informará de la ronda en la que se ha encontrado la mejor solución hasta el momento.

Cuando se ejecuta la aplicación, además de la Consola de configuración, se abre otra ventana donde se muestra el grafo sobre el que se trabaja. A medida que avance la simulación, las aristas del grafo cambiarán de color en función de los siguientes criterios:

- Se pinta en rojo si la arista pertenece a la mejor ruta encontrada hasta el momento.
- Se pinta en verde si la arista pertenece a la ruta encontrada en la última ronda.
- En el resto de casos se pinta en azul.

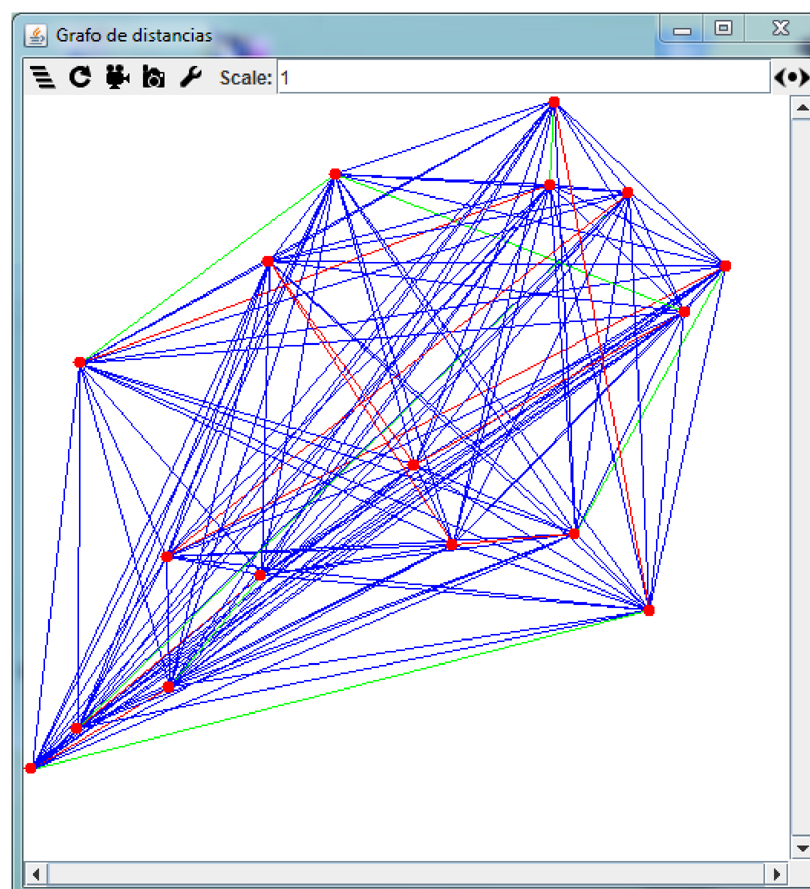

Ilustración 25 – Grafo de distancias para el TSP

El grafo de distancias que se muestra en la Ilustración 25 es interactivo, de tal forma que si hace doble click en alguno de los nodos o las aristas se añadirán en la pestaña "Inspectors", donde podremos ver en detalle sus propiedades e incluso modificarlas. En la

Ilustración 26 se observa la información que se ofrece de una arista en un problema resuelto por ACO.

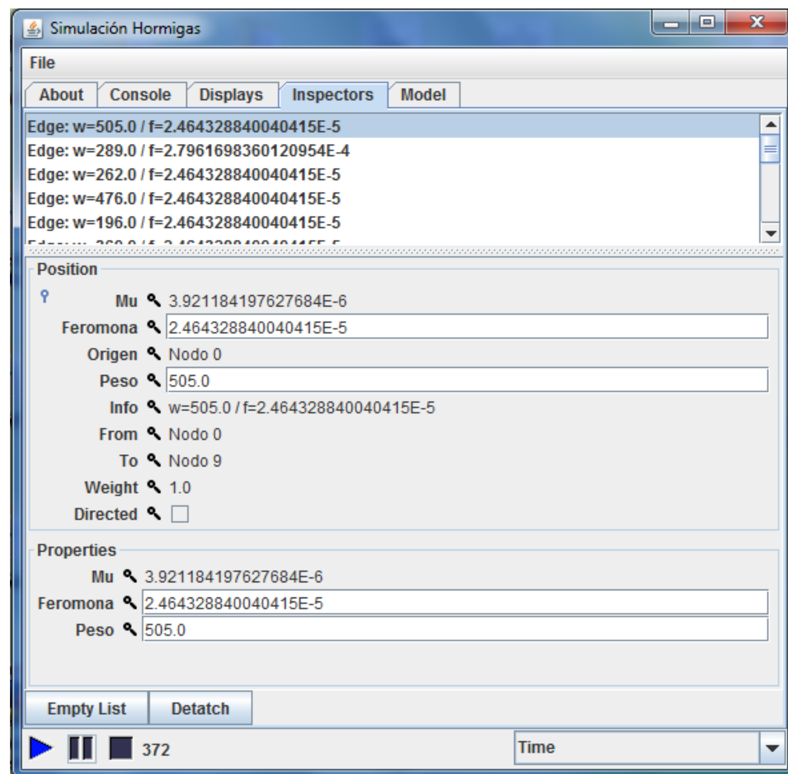

Ilustración 26 - Consola de configuración (pestaña "Inspectors")

La librería MASON proporciona soporte para generar distintos gráficos de las variables mostradas en la pestaña “Model”. Podemos aprovechar esto para visualizar como varía el coste de la solución encontrada por las hormigas a lo largo del tiempo.

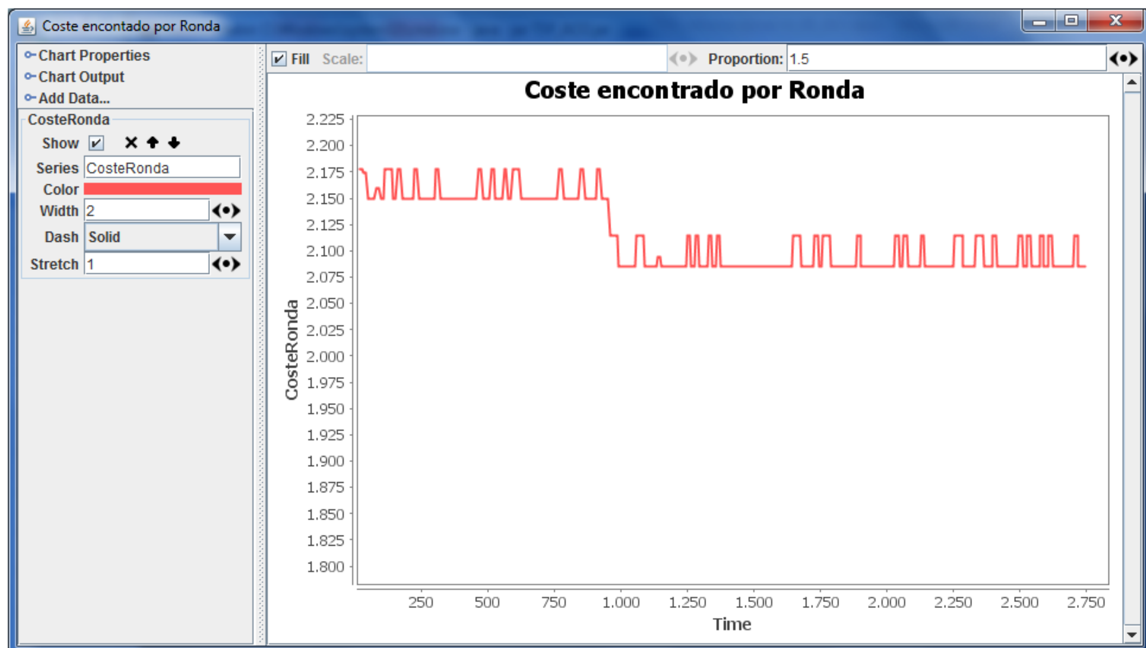

Ilustración 27 – Evolución del coste de la solución encontrada por ACO

En la gráfica mostrada en la Ilustración 27 se puede observar como los valores que se obtienen a lo largo de las rondas fluctúan a lo largo del tiempo, con la propiedad de que nunca se está muchas iteraciones sin obtener (o mejorar) la mejor solución obtenida hasta el momento. En este problema, se ha encontrado la solución óptima (de coste 2085) en pocas iteraciones.

A continuación se cambia el modelo, seleccionado el mismo problema pero esta vez que sea solucionado por el algoritmo BSO. Cuando se ejecuta el código generado se observa que la Consola de configuración ha cambiado un poco: en la pestaña “About” esta vez se indica que la solución se realiza mediante BSO y en la pestaña “Model” se muestran los parámetros de configuración del BSO (detallados en la sección 2.3.3).

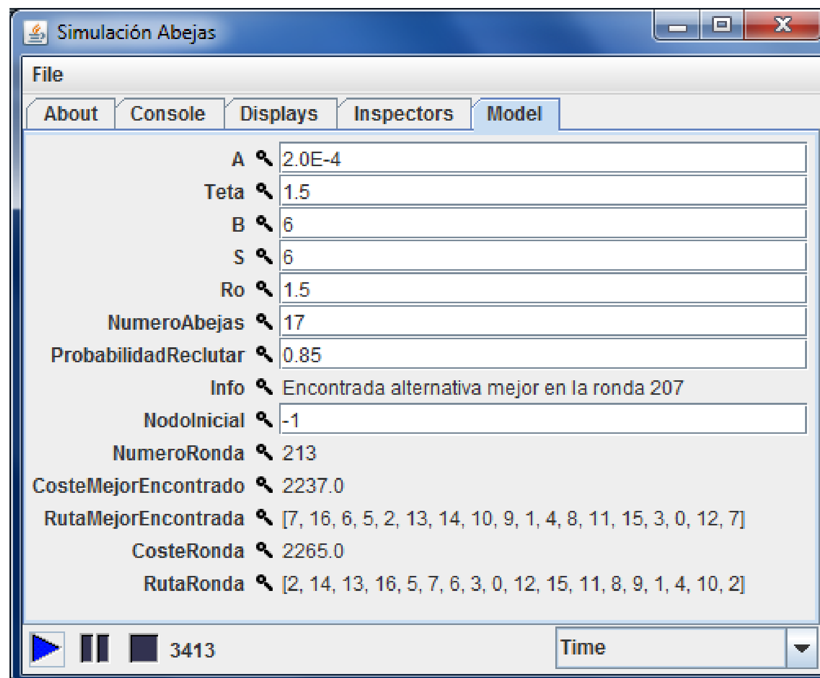

Ilustración 28 – Consola de configuración para resolver TSP con BSO (pestaña "Model")

Si ejecutamos la simulación y generamos un gráfico de los costes que se obtienen en cada ronda, podremos ver la evolución de los costes con BSO y compararlo con lo que sucedía en ACO. En la Ilustración 29 se muestra esta evolución.

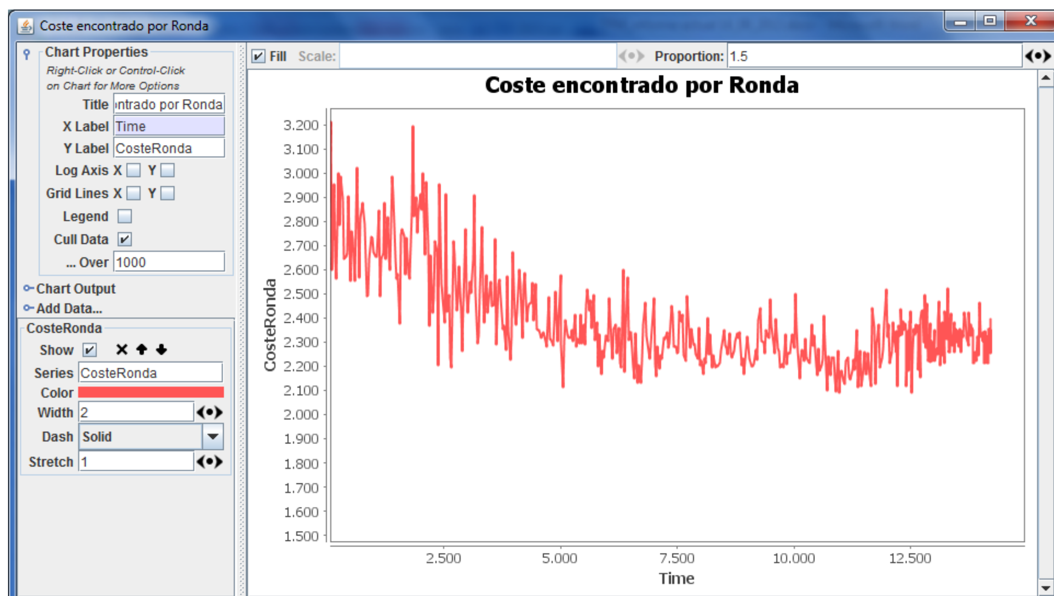

Ilustración 29 – Evolución del coste de la solución encontrada por BSO

Se observa que en este caso hay mucha más variedad de resultados. Esto puede deberse a que el algoritmo BSO es más permisivo con respecto a probar rutas ya que no existe una lista de candidatos de la cual hay que elegir un nodo. El algoritmo también ha encontrado la solución óptima, pero lo ha conseguido en más iteraciones.

4.2 Resolución del GTSP

A partir del grafo seleccionado para probar el TSP, se creará un grafo dividido en 4 conjuntos de forma aleatoria. Una vez más, se verificarán las soluciones que ofrecen ACO y BSO. Al ejecutar el código generado para el modelo, se observa que la interfaz obtenida es igual a la que se obtenía para el TSP (excepto para la pestaña "About", la cual da información sobre el problema que se va a resolver). Los motivos de que la interfaz no varíe son los siguientes:

- Como el problema se convierte a un TSP, los parámetros que ACO o BSO necesitan para resolverlo no varían y por lo tanto la Consola de configuración tampoco.
- Ya que en la conversión del GTSP al TSP (explicada en la sección 2.1.4) no se generan nodos nuevos en el grafo (únicamente se cambian las aristas origen y se añaden algunas para generar un ciclo de coste) ni se ha implementado la forma de separar visualmente los conjuntos de nodos, el grafo parece el mismo aunque no lo sea.

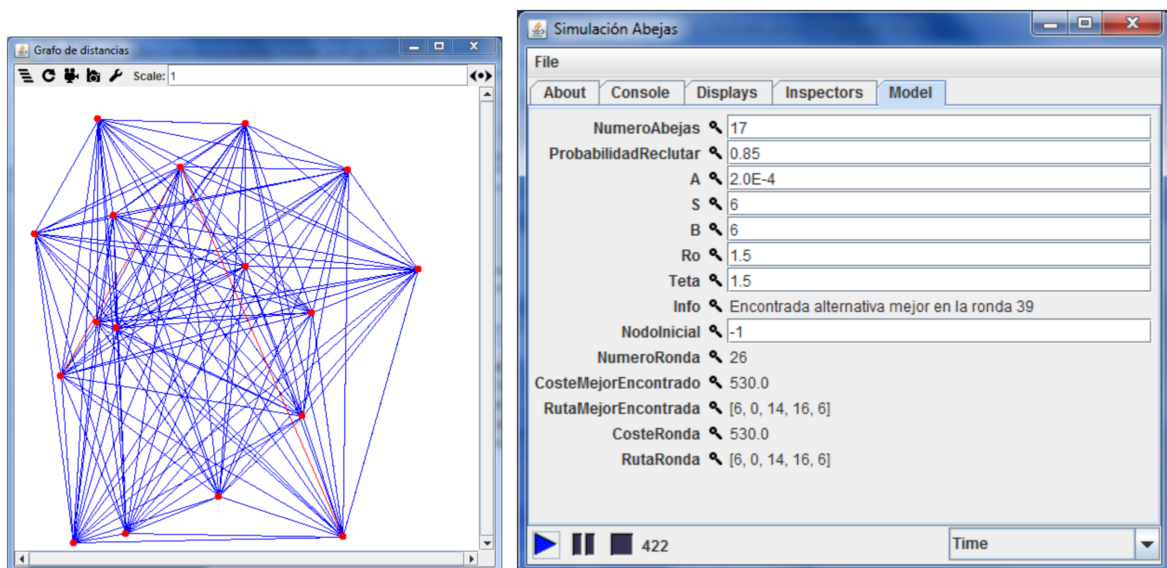

Ilustración 30 - Instante de la resolución del GTSP por BSO

Como se observa en la Ilustración 30, la solución que se obtiene es más corta, ya que en lugar de visitar todos los nodos se visita únicamente uno de cada conjunto. Solucionar el GTSP a través de una transformación al TSP es una buena estrategia ya que no se aumenta la dimensión del grafo con el que se trabaja y tanto ACO como BSO ofrecen buenos resultados en tiempos más cortos que el TSP (ya que los caminos incluyen menos nodos).

Todas la funcionalidad de la interfaz gráfica mostrada en el ejemplo del TSP (gráficas de evolución del coste con el tiempo, inspectores en las aristas y nodos, etc.) están también disponibles en este problema.

4.3 Resolución del CPP

Para probar la resolución del problema del cartero chino se ha tomado el mismo grafo usado como ejemplo para resolver el TSP. En primer lugar se creará un modelo que solucione este problema con una conversión al GTSP (y este a su vez, con una conversión al TSP) y mediante ACO. Al ejecutar el código que genera el modelo observamos que la Consola de configuración no ha variado, pero sí el grafo de distancias.

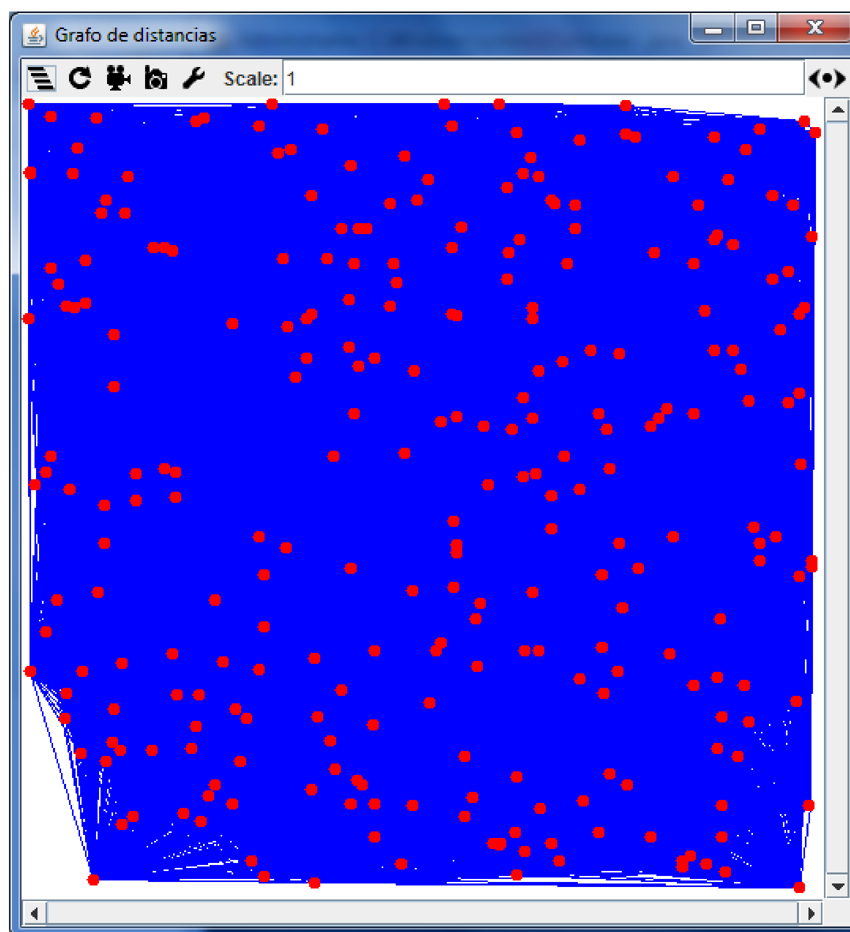

Ilustración 31 - Grafo de distancias para el CPP

Como se observa en la Ilustración 31, a pesar de que el grafo de nuestro problema inicial es el mismo, el grafo resultado es mucho mayor. Esto es debido a que el grafo original es un grafo completo (es decir, contiene todas las aristas posibles) y el algoritmo de conversión del CPP al GTPS (explicado en la sección 2.1.4) genera un nodo en el GTSP por cada arista existe en el CPP.

Dado el gran incremento del tamaño del grafo (no sólo se crea un nodo por cada arista sino que además luego se genera un grafo completo) tanto ACO como BSO, pese a

obtener buenos resultados, tardan bastante en finalizar cada ronda. Es posible que para este problema sea más adecuado implementar una solución directa a utilizar la solución basada en una transformación.

Del mismo modo que pasaba con el TSP y el GTSP, todas las utilidades de la interfaz gráfica están disponibles para el CPP.

5 TRABAJOS RELACIONADOS

Dado que el sistema incorpora conceptos de distintos ámbitos, se han buscado trabajos relacionados para cada uno de ellos. En primer lugar se presentan los trabajos que tiene relación con el desarrollo de sistemas multi-agente mediante técnicas de MDE. A continuación, se presentan otras herramientas diseñadas para resolver problemas de optimización de forma colaborativa (ya sea con métodos de inteligencia de enjambre o no). Después, se revisan algunos trabajos en los que se utiliza el enfoque de MDE para el planteamiento y la resolución de problemas de búsqueda. Por último, se presenta una librería específica para la resolver problemas en grafos seguida de algunas conclusiones.

Existen varias líneas de investigación en las cuales se incorpora la MDE al desarrollo de sistemas multi-agente. Normalmente estos sistemas están orientados a conseguir desarrollar un framework suficientemente flexible para poder desarrollar un MAS con él. Este es el caso de BOCHICA [39], un framework pensado para ofrecer un lenguaje común con el que puedan comunicarse los investigadores del entorno de los agentes y los desarrolladores. El enfoque de BOCHICA parte de que no es acertado intentar diseñar un meta-modelo o un DSL que contenga todos los conceptos que puedan aparecer en cada una de las aplicaciones basadas en agentes. En lugar de ello, optan por lo siguiente:

- Diseñar un DSL que contenga los conceptos genéricos aplicables a todas las aplicaciones basadas en agentes.
- Ofrecen diversas interfaces donde el usuario puede extender el sistema para adecuarlo a sus necesidades.

BOCHICA está desarrollada en EMF y está basada en el Domain Specific Modeling Languaje for Multiagent Systems (DSML4MAS), presentado en [40].

En [41] intentan probar como el enfoque de ingeniería dirigida por modelos es adecuado para el desarrollo de sistemas multi-agente. Los autores aplican el enfoque MDE a la metodología de software orientada a agentes Prometheus [42]. Los autores, a partir del Prometheus UML meta-model (PUMM) implementado con EMF, consiguen transformarlo al Prometheus Ecore meta-model (PEMM) y posteriormente generar el código de la aplicación.

MASDK (Multi-Agent System Development Kit), presentado en [43], proporciona un entorno donde desarrollar MAS siguiendo la metodología Gaia [44]. MASDK propone un entorno eficiente y eficaz donde la especificación del MAS se haga a través del diseño de los diagramas que se identifican en la Ilustración 32.

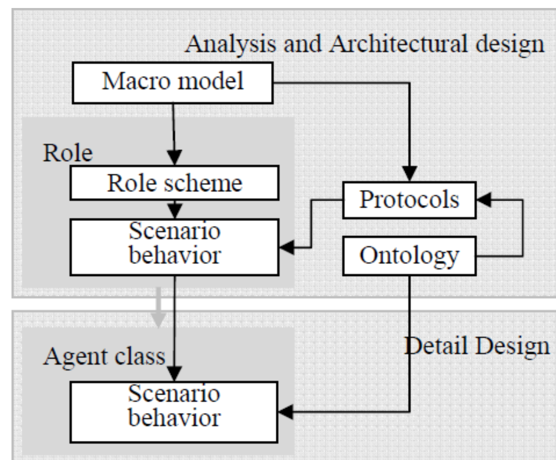

Ilustración 32 Tipos de diagramas y sus relaciones en MASDK (tomada de [43])

En cuanto al desarrollo de herramientas de resolución de problemas de optimización a través de métodos de swarm intelligence, en [45] se presenta la herramienta de simulación TSPAntSim, la cual permite resolver el TSP por medio de algoritmos de ACO. TSPAntSim está desarrollada en Java e implementa varios métodos basados en hormigas, entre los que se encuentra el Ant System y Ant Colony System, ambos explicados en la sección 2.3.2 y el último implementado en nuestro sistema. En la Ilustración 33 se muestra una captura de esta herramienta.

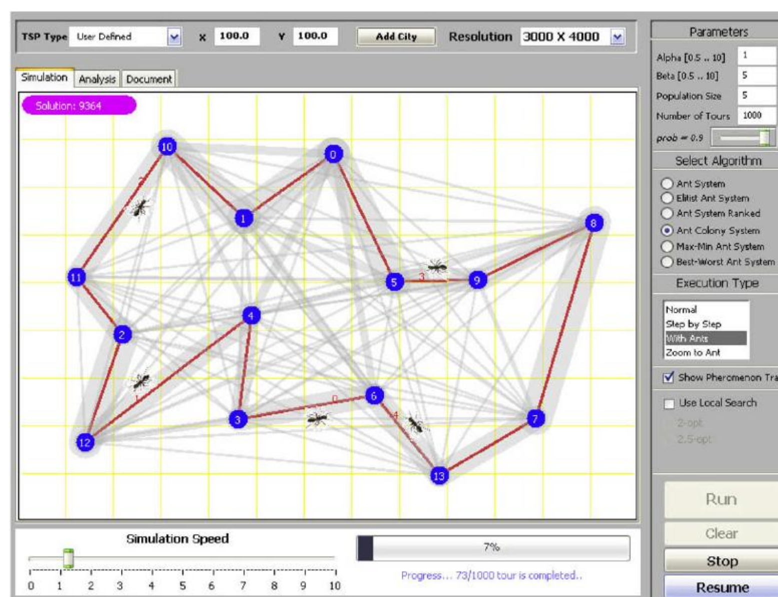

Ilustración 33 – Captura de pantalla de TSPAntSim (tomada de [45])

El framework MANGO (MultiAgent ENvironment for Global Optimization) [46] tiene como objetivo proporcionar un entorno donde solucionar problemas de optimización mediante agentes que cooperan. Al contrario que sucedía con TSPAntSim, MANGO no está pensado para un problema concreto ni implementa ningún algoritmo sino que proporciona el

entorno donde desarrollar problemas de optimización. Las principales características de MANGO son las siguientes:

- En un sistema centralizado, en el cual el agente especial *Directory Agent* conoce a todos los agentes del sistema y los servicios que ofrece.
- Los agentes son típicamente heterogéneos, es decir, cada agente puede realizar unas tareas distintas.
- Las comunicaciones entre los agentes se implementan sobre la tecnología Java Messaging Service (JMS).
- Los agentes en MANGO pueden proporcionar servicios a otros agentes, consumir servicios de otros agentes, o realizar las dos cosas al mismo tiempo.
- El diseño de un agente en MANGO parte por decidir:
    1) La tarea que seguirá el agente dentro del algoritmo que resuelve el problema de optimización tratado.
    2) La información que proporcionará al resto de agentes.
    3) La información que necesitará del resto de agentes.
- Cuando un agente requiera de los servicios de otro, le preguntará al *Directory Agent*, ya que él conoce a todos los agentes y sus servicios.

En cuanto a la aplicación de MDE para plantear y resolver problemas NP, en [47] explican un framework diseñado para resolver problemas de búsqueda. Este framework trata de combinar los puntos fuertes de tres enfoques distintos: satisfactibilidad proposicional (o SAT), problemas de satisfacción de restricciones (o CSP) y programación mediante conjuntos de repuestas (o ASP). En [48] utilizan también MDE para resolver un tipo de problema de optimización multi-objetivo.

Si nos centramos en la representación y resolución de problemas en grafos, existen librerías especializadas en ellos. Este es el caso de LEMON (Library for Efficient Modeling and Optimization in Networks) [49], una librería genérica de grafos desarrollada en C++. La clara separación de LEMON entre los algoritmos y estructuras de datos permite una integración ágil con el problema del usuario. Esta librería destaca por el gran número de algoritmos de optimización que implementa y por su buen rendimiento.

En esta revisión de trabajos se han encontrado entornos y herramientas diseñadas para la resolución de problemas de optimización, pero ninguna de ellas reúne todas las características que ofrece el framework presentado en este trabajo. La herramienta TSPAntSim proporciona un entorno donde resolver problemas mediante swarm intelligence pero, a diferencia de nuestro sistema, está limitado a un solo problema y a un determinado número de soluciones (todas basadas en hormigas). El framework MANGO sienta las bases técnicas para desarrollar un MAS colaborativo, pero no constituye en sí un entorno donde poder definir y resolver problemas de optimización. LEMON está diseñada para ser una librería de grafos, y no constituye un entorno ágil dónde definir problemas de optimización.

El aporte que más importante que proporciona nuestro sistema con respecto a los de los trabajos relacionados es la capacidad de resolver un problema mediante una o varias transformaciones a otros problemas. Haber aplicado un enfoque de MDE al sistema otorga al usuario la capacidad de modelar estas transformaciones para conseguir resolver un problema de optimización sin haber programado el algoritmo que lo resuelve.

6 CONCLUSIONES Y TRABAJO FUTURO

En este trabajo se expone cómo mediante técnicas de MDE se ha desarrollado un sistema capaz de resolver problemas de optimización sobre grafos de manera flexible. Para resolver estos problemas se han implementado, sobre un MAS implementado con MASON, dos algoritmos de swarm intelligence: Ant Colony Optimization y Bee Swarm Optimization.

Gracias al planteamiento siguiendo MDE, se consigue un entorno donde el usuario determina los detalles de su problema de optimización y el método de resolución. El uso de MDE permite que el usuario tenga la posibilidad de modelar una solución para un problema de optimización transformándolo a otro. Estas conversiones han permitido que pese a que los algoritmos implementados sólo están diseñados para resolver el TSP, el sistema consiga resolver también el GTSP y el CPP. Esta potencia en cuanto a capacidad de resolución de problemas a veces, como se veía en el experimento de la sección 4.3, lleva acarreada una perdida en rendimiento.

La aplicación de MDE con las herramientas de EMF ha permitido que, en un mismo entorno, a partir de una instancia del meta-modelo se genere el código de la aplicación, se compile y se ejecute. Además, la instanciación se realiza con una gramática cercana al lenguaje natural, que dispone de autocompletado y detección de errores sintácticos. Esta instanciación es mucho más sencilla que programar código MASON y consigue un aumento de la velocidad de desarrollo.

En el trabajo se muestra cómo los algoritmos de swarm intelligence son una forma interesante de abordar los problemas de optimización. Estos algoritmos ofrecen una solución heurística a un problema de optimización, la cual irá mejorando a medida que pase el tiempo de simulación. En cuanto a los algoritmos implementados, pese a que sus parámetros determinan su comportamiento, se han detectado tendencias generales. El algoritmo de ACO consigue soluciones cercanas al óptimo en pocas iteraciones, ya que su lista de candidatos obliga a hacer búsquedas más dirigidas. En cuanto a BSO, al dejar mayor libertad para hacer las búsquedas, hace que se prueben más caminos hasta encontrar una buena solución.

El desarrollo del sistema mediante un MAS implementado con la herramienta MASON ha contribuido a conseguir una implementación ágil y a construir un entorno fácilmente extensible. El uso de MASON ha permitido obtener, invirtiendo poco tiempo en su desarrollo, una interfaz gráfica con mucha funcionalidad que permite arrancar, pausar y detener las simulaciones, ajustar los parámetros de los algoritmos, visualizar en tiempo real los resultados (tanto de forma textual cómo de forma gráfica en el grafo), obtener gráficas con la evolución de la solución, entre otras.

Se ha delegado para trabajo futuro el incluir dos nuevos tipos de solución: las soluciones competitivas y las soluciones cooperativas. Para poder incluir estas nuevas soluciones como clases herederas de *SolutionMethod*, habría que ajustar el meta-modelo tal como se muestra en la Ilustración 34.

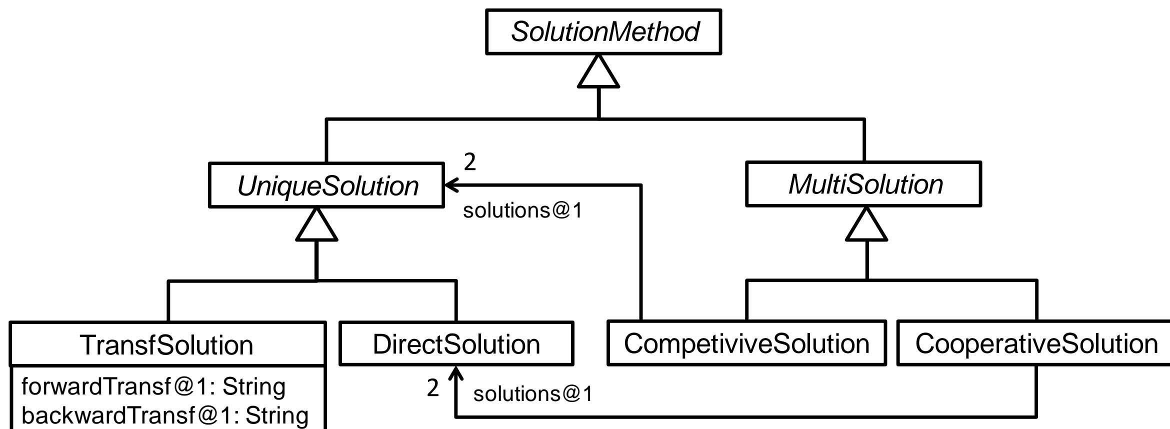

Ilustración 34 - Meta-modelo de las soluciones

La solución competitiva estaría formada por dos soluciones a un problema, pudiendo ser ambas directas, ambas mediante transformaciones o una de cada. Dicha solución competitiva ejecutaría a la vez las dos soluciones seleccionadas, permitiendo al usuario ver la evolución de ambos algoritmos.

En las soluciones cooperativas se elegirán dos soluciones directas y mediante métodos de combinación de heurísticas se obtendrá una nueva solución. En [50] explican uno de estos métodos, la metaheurística, la cual se define como un proceso maestro que guía y modifica las operaciones de sus heurísticas subordinadas para producir soluciones de mayor calidad. El incorporar una combinación de soluciones al sistema (mediante metaheurística u otros medios) podría ofrecer una nueva solución al sistema que mejorase la calidad individual de ACO y BSO.

Otras tareas se han delegado también al futuro, como la de incorporar otro dominio de problemas de optimización en grafos (por ejemplo, problemas en una red de flujo) y la de conseguir que más parámetros de los algoritmos tomen su valor por defecto en función del grafo que instancia el problema (actualmente algunos ya lo hacen).

BIBLIOGRAFÍA

[1] T. Stahl, M. Völter, J. Bettin, A. Haase y S. Helsen, Model-Driven Software Development, John Wiley & Sons, 2006.

[2] D. Avis, A. Hertz y O. Marcotte, Graph Theory and Combinatorial Optimization, Springer, 2005.

[3] M. R. Garey y D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H.Freeman and company, 1979.

[4] E. Bonabeau, M. Dorigo y G. Theraulaz, Swarm Intelligence: From natural to Artificial Systems, New York; Oxford: Oxford University Press, 1999.

[5] C. E. Noon y J. C. Bean, «An Efficient Transformation of the Generalized Traveling Salesman Problem,» *INFOR,* vol. 31, pp. 39-44, 1993.

[6] G. Laporte, «Modeling and solving several classes of arc routing problems as traveling salesman problems,» *Computers & Operations Research,* vol. 24, pp. 1057-1061, 1997.

[7] N. R. Jennings, K. Sycara y W. Michael, «A Roadmap of Agent Research and Development,» *Autonomous Agents and Multi-Agent Systems,* nº 1, pp. 7-38, 1998.

[8] M. J. Wooldridge, An Introduction to MultiAgent Systems, John Wiley & Sons, 2009.

[9] M. Wooldridge y N. R. Jennings, «Intelligence agents: theory and practice,» *The Knowledge Engineering Review,* vol. 10, nº 2, pp. 115-152, 1995.

[10] N. Gilbert y K. G. Troitzsch, Simulation for the social scientist, Philadelphia: Open University Press Philadelphia, 1999.

[11] M.-P. Gleizes, V. Camps, J.-P. Georgé y D. Capera, «Engineering Systems Which Generate Emergent Functionalities,» de *Engineering Environment-Mediated Multi-Agent Systems*, Springer Berlin Heidelberg, 2008, pp. 58-75.

[12] D. Teodorovic, «Transport Modeling by Multi-Agent Systems: A Swarm Intelligence Approach,» *Transportation Planning and Technology,* vol. 26, nº 4, pp. 289-312, 2003.

[13] L. Panait y S. Luke, «Ant Foraging Revisited,» de *Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE-IX)*, 2004, pp. 569-574.

[14] L. Panait y S. Luke, «A Pheromone-Based Utility Model for Collaborative Foraging,» de *In Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-2004)*, 2004, pp. 36-43.

[15] G. Reinelt, The Traveling Salesman: Computational Solutions for TSP Applications, Springer, 1994.

[16] K. Von Frisch, Bees: Their Vision, Chemical Senses and Language. (Revised edn), New York: Ithaca, 1976.

[17] H. Drias, S. Saged y S. Yahi, «Cooperative bees swarm for solving the maximum weighted satisfiability problem,» *Computational intelligence and bioinspired systems,* vol. 3512, pp. 318-325, 2005.

[18] P. Lucic y D. Teodorovic, «Transportation Modeling: An Artificial Life Approach,» *Proceedings. 14th IEEE International Conference on In Tools with Artificial Intelligence (ICTAI 2002),* pp. 216-223, 2002.

[19] J. de Lara y E. Guerra, «Deep Meta-Modelling with MetaDepth,» *Lecture Notes in Computer Science,* vol. 6141, nº 48, pp. 1-20, 2010.

[20] C. Atkinson y T. Kühne, «Reducing accidental complexity in domain models,» *Software & Systems Modeling,* vol. 7, pp. 345-359, 2008.

[21] A. Rob, «Survey of Agent Based Modelling and Simulation Tools,» Science & Technology Facilities Council, Warrington, 2010.

[22] «Swarm,» [En línea]. Available: http://swarm.org/.

[23] «Multi-Agent Modeling Language,» [En línea]. Available: http://www.maml.hu/.

[24] «AndroMeta,» [En línea]. Available: http://andrometa.net.

[25] «The Repast Suite,» [En línea]. Available: http://repast.sourceforge.net.

[26] «MASON Multiagent Simulation Toolkit,» [En línea]. Available: http://cs.gmu.edu/~eclab/projects/mason/.

[27] «Eclipse Modeling Framework,» [En línea]. Available: http://www.eclipse.org/modeling/emf/.

[28] «Emfatic,» [En línea]. Available: http://wiki.eclipse.org/Emfatic.

[29] «Graphical Modeling Framework,» [En línea]. Available: http://www.eclipse.org/modeling/gmp/.

[30] «Epsilon,» [En línea]. Available: http://www.eclipse.org/epsilon/.

[31] «Xtext,» [En línea]. Available: http://www.eclipse.org/Xtext/.

[32] «Acceleo,» [En línea]. Available: http://www.eclipse.org/acceleo/.

[33] «Epsilon Generation Language,» [En línea]. Available: http://eclipse.org/epsilon/doc/egl/.

[34] «MOF Model To Text Transformation Language (MOFM2T),» [En línea]. Available: http://www.omg.org/spec/MOFM2T/1.0/.

[35] J. de Lara, E. Guerra y J. Sánchez Cuadrado, «metaDepth: A framework for deep meta-modelling,» [En línea]. Available: http://astreo.ii.uam.es/~jlara/metaDepth/.

[36] D. S. Kolovos, R. F. Paige y F. Polack, «The Epsilon Object Language (EOL),» *ECMDA-FA,* vol. 4066, pp. 128-142, 2006.

[37] D. S. Kolovos, R. F. Paige y F. Polack, «The Epsilon Transformation Language,» de *LNCS*, vol. 5063, Springer, 2008, pp. 46-60.

[38] «TSP: Data for the Traveling Salesperson Problem,» [En línea]. Available: http://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html.

[39] S. Warwas, K. Fischer, M. Klusch y P. Slusallek, «BOCHICA: A Model-driven Framework for Engineering Multiagent Systems.,» *ICAART 1,* pp. 109-118, 2012.

[40] C. Hahn, «A domain specific modeling language for multiagent systems.,» *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008),* pp. 233-240., 2008.

[41] J. M. Gascueña, E. Navarro y A. Fernández-Caballero, «Model-driven engineering techniques for the development of multi-agent systems,» *Engineering Applications of Artificial Intelligence,* vol. 25, pp. 159-173, 2012.

[42] L. Padgham y M. Winikoff, Developing Intelligent Agent Systems: A Practical Guide, New York: New York Halsted Press , 2004.

[43] V. Gorodetsky, O. Karsaev, V. Konushy y V. Samoilov, «Model-Driven Engineering of Multi Agent Systems,» *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology ,* vol. 2, pp. 83-86, 2008.

[44] F. Zambonelli, N. R. Jennings y M. Wooldridge, «Developing multiagent systems: The Gaia methodology,» *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 12, pp. 317-370 , 2003.

[45] U. Aybars y D. Aydin, «An interactive simulation and analysis software for solving TSP using Ant Colony Optimization algorithms,» *Advances in Engineering Software,* vol. 40, pp. 341-349, 2009.

[46] A. Günay, F. Öztoprak, I. Birbil S. y P. Yolum, «Solving Global Optimization Problems

Using MANGO.,» *Lecture Notes in Computer Science,* pp. 783-792, 2009.

[47] D. G. Mitchell y E. Ternovska, «A Framework for Representing and Solving NP Search Problems,» de *The Twentieth National Conference on Artificial Intelligence*, Pittsburgh, 2005, pp. 430-435.

[48] F. R. Burton, R. F. Paige, L. M. Rose, D. S. Kolovos, S. Poulding y S. Smith, «Solving Acquisition Problems Using Model-Driven Engineering,» *Lecture Notes in Computer Science,* vol. 7349, pp. 428-443, 2012.

[49] «LEMON Graph Library,» [En línea]. Available: http://lemon.cs.elte.hu/trac/lemon.

[50] C. Blum y A. Roli, «Metaheuristics in combinatorial optimization: Overview and conceptual comparison,» *ACM Computing Surveys (CSUR),* vol. 35, pp. 268-308, 2003.