Desarrollo Software Dirigido por Modelos de Sistemas Multi-Robot Siguiendo VigilAgent

José M. Gascueña, Elena Navarro y Antonio Fernández-Caballero

Universidad de Castilla-La Mancha, Departamento de Sistemas Informáticos &
Instituto de Investigación en Informática de Albacete (I3A), 02071-Albacete, Spain
{JManuel.Gascuena, Elena.Navarro, Antonio.Fdez}@uclm.es

Resumen Este artículo describe la experiencia adquirida al aplicar la metodología *VigilAgent* para desarrollar una aplicación de vigilancia robótica. La metodología produce aplicaciones software documentadas desde la captura de requisitos hasta la implementación. Aprovecha la base tecnológica de las transformaciones de modelos que forman parte del enfoque de Desarrollo de Software Dirigido por Modelos (DSDM). En este caso, los agentes son los elementos clave de abstracción de los modelos especificados para desarrollar aplicaciones de vigilancia. Por una parte, es necesaria una transformación modelo a modelo (M2M) porque a lo largo del proceso de desarrollo de *VigilAgent* se utilizan dos lenguajes de modelado diferentes, correspondientes a las metodologías Prometheus e INGENIAS, respectivamente. Por otra parte, se realiza una transformación modelo a texto (M2T) para generar código para el marco de trabajo ICARO-T a partir del modelo de INGENIAS. El caso de estudio presentado para ilustrar la propuesta es el de un equipo de robots que colaboran para hacer frente a las alarmas que se producen en un entorno de vigilancia simulado.

Keywords: Desarrollo Dirigido por Modelos, Sistemas Multiagente, Ingeniería del Software Orientada a Agentes, Robots Móviles, Vigilancia.

1. Introducción

El Desarrollo Dirigido por Modelos (DDM) es un enfoque que está adquiriendo un interés cada vez mayor para desarrollar software. La idea básica sobre la que gira la filosofía de este enfoque es elevar el nivel de abstracción en el que trabaja el desarrollador con el fin de explotar los modelos como la pieza angular del desarrollo. Esto tiene las siguientes consecuencias en el ciclo de vida de desarrollo: (1) se invierte un mayor tiempo en el análisis y diseño de los modelos, (2) se reduce el tiempo necesario para realizar la tarea de codificación porque puede desarrollarse un generador de código que automatiza gran parte de esta tarea, siendo los programadores responsables de completar aquellas partes que o bien se ha decidido no generar automáticamente o simplemente no se han podido generar, (3) se obtiene una mayor calidad en el código, pues el código generado por el generador no contiene errores, (4) se incrementa la

productividad porque se reduce el tiempo invertido en la fase de codificación y se hace un esfuerzo por corregir todos los errores en las fases más tempranas del ciclo de vida, evitando de esta forma el efecto de "bola de nieve", y (5) se mejora la portabilidad debido a que adoptar una nueva tecnología sólo requiere desarrollar un nuevo generador de código, pues los modelos son independientes de cualquier tecnología de implementación. Además, DDM también proporciona interoperabilidad entre sistemas heterogéneos gracias a la especificación de puentes entre tecnologías diferentes. En resumen, utilizar un enfoque DDM para desarrollar aplicaciones software permite obtener grandes beneficios en aspectos fundamentales tales como la productividad, la portabilidad, la interoperabilidad y el mantenimiento [20], [23].

En el enfoque DDM, las tecnologías de transformación de modelos [6] juegan un papel esencial para lograr transformar modelos descritos con un alto nivel de abstracción en modelos especificados a un nivel de abstracción más bajo. Por una parte, las transformaciones modelo a modelo (M2M) transforman un modelo origen en un modelo destino situado en el mismo o diferente nivel de abstracción. Por otra parte, las transformaciones modelo a texto (M2T) son otra pieza clave de DDM porque automatizan el último paso del proceso al generar el código fuente final del sistema. La diferencia entre M2M y M2T es que el modelo destino obtenido, tras ejecutar la transformación, en el primer caso es una instancia de un metamodelo destino, mientras que en el segundo es simplemente un documento en formato textual, generalmente de tipo String.

En la actualidad, cada vez es más común utilizar robot móviles para asistir a los humanos en la realización de tareas de vigilancia. Los beneficios potenciales son ahorrar costes, no exponer a los humanos a situaciones peligrosas y realizar las rutinas de vigilancia de una forma más eficaz ya que los humanos se aburren cuando llevan a cabo tareas que requieren muchas horas de trabajo [1]. A veces, las capacidades de un único robot son suficientes para realizar una tarea como la detección y seguimiento de un intruso [9]. Sin embargo, también es común plantear escenarios que requieren la colaboración de múltiples robots móviles para satisfacer un objetivo común tal como el de explorar o cubrir un área extensa [22]. Estas tareas complejas se pueden resolver con éxito incorporando tecnologías de los sistemas multiagente [24] debido a características inherentes en los agentes tales como (1) la autonomía, necesaria para alcanzar objetivos individuales y colectivos; y (2) la sociabilidad, indispensable para formar equipos de robots que colaboran para afrontar las tareas planteadas.

Las observaciones anteriores ponen de manifiesto la viabilidad de utilizar los métodos, las técnicas y las herramientas presentes en el área del paradigma de los agentes [25] para desarrollar sistemas de vigilancia robótica. De hecho, hay algunos trabajos que utilizan las metodologías basadas en agentes Cassiopeia [3], MaSE [8], PASSI [4] y Prometheus [14] para analizar y diseñar sistemas robóticos. Sin embargo, no abordan la tarea de generación de código a partir de los modelos. Nuestra propuesta es aprovechar las características sobresalientes de varias metodologías existentes para producir aplicaciones de vigilancia bien documentadas desde la especificación de requisitos hasta la implementación

[10]. Esta idea se apoya en la hipótesis ampliamente aceptada de que no existe una única metodología útil para todos los desarrolladores sin algún nivel de personalización [5]. Habitualmente se combinan técnicas y herramientas propuestas en diferentes metodologías.

En este artículo se describe cómo se aplica el proceso de nuestra metodología de desarrollo de aplicaciones multiagente, denominada *VigilAgent*, para abordar el problema de una colección de robots que patrullan alrededor de un entorno de vigilancia. Esta metodología no se ha desarrollado desde cero sino que reutiliza, para el modelado, fragmentos de las metodologías Prometheus [18] e INGENIAS [19], y el marco de trabajo ICARO-T [11] para la implementación. Por una parte, resaltar que en un determinado instante de la metodología *VigilAgent* se realiza una transformación M2M para transformar algunos modelos de Prometheus en modelos de INGENIAS, pues dichas metodologías no utilizan el mismo lenguaje de modelado para describir los modelos. Por otra parte, se aplica finalmente una transformación M2T para transformar los modelos de INGENIAS en código de ICARO-T. Por lo tanto, *VigilAgent* aprovecha el background tecnológico de las habituales transformaciones de modelos del enfoque DDM y los agentes son los elementos clave de abstracción de los modelos especificados para desarrollar aplicaciones de vigilancia.

El resto del artículo se estructura de la manera siguiente. En la sección 2 se ofrece una visión general de la metodología *VigilAgent*, justificando por qué se integran las tecnologías Prometheus, INGENIAS e ICARO-T. A continuación, la sección 3 introduce un caso de estudio que ilustra cómo se aplica *VigilAgent*. Finalmente, la sección 4 ofrece algunas conclusiones.

2. Visión General de la Metodología VigilAgent

El ciclo de vida software de la metodología *VigilAgent* cubre las 5 fases siguientes:

1. *Especificación del sistema*. El analista identifica los requisitos del sistema y el entorno en el que está situado a partir de la descripción del problema obtenida fruto de las reuniones establecidas con el cliente.
2. *Diseño arquitectónico*. El arquitecto del sistema determina los tipos de agentes que existen en el sistema y cómo interactúan.
3. *Diseño detallado*. El diseñador de agentes y el diseñador de aplicaciones colaboran para concretar la estructura interna de cada entidad que forma parte de la arquitectura global del sistema obtenida en la fase anterior.
4. *Implementación*. El desarrollador de software genera y completa el código de la aplicación.
5. *Despliegue*. El gestor de despliegue despliega la aplicación conforme a un modelo establecido.

Es relevante resaltar varios detalles. El primero es que las dos primeras fases de *VigilAgent*, especificación del sistema y diseño arquitectónico, se corresponden con las dos primeras fases de la metodología Prometheus. Otro detalle, es que la

tercera fase de *VigilAgent* (diseño detallado) se lleva a cabo utilizando modelos de la metodología INGENIAS. Finalmente, destacar que el código se genera y despliega en el marco de trabajo ICARO-T. A continuación se describen cuáles son las razones que justifican por qué se ha realizado la integración entre las tecnologías citadas.

Prometheus es significativa debido a las guías que ofrece para identificar los agentes y sus interacciones. Otra ventaja de Prometheus es el uso explícito del concepto de escenario, estrechamente relacionado con el lenguaje específico utilizado en el dominio de la vigilancia. De hecho, una aplicación de vigilancia se desarrolla para hacer frente a un conjunto de escenarios. Sin embargo, observar que la última fase de Prometheus no se ha integrado en *VigilAgent* porque se centra en un tipo de agente concreto, los agentes belief-desire-intention (BDI) [21], y cómo se transforman las entidades obtenidas durante el diseño en conceptos utilizados en el lenguaje de programación orientado a agentes denominado JACK [2]. Esto supone, en principio, una perdida de generalidad. Por el contrario, INGENIAS facilita un proceso general para transformar los modelos especificados durante la fase de diseño en código ejecutable. Sin embargo, INGENIAS no ofrece guías para identificar las entidades del modelo; es la experiencia del desarrollador la que determina su identificación. Por lo tanto, la metodología *VigilAgent* no se ha desarrollado desde cero sino que integra fragmentos de Prometheus e INGENIAS para explotar las ventajas que ofrecen.

Respecto a la implementación, se ha seleccionado el marco de trabajo ICARO-T porque proporciona componentes software de alto nivel que facilitan el desarrollo de aplicaciones de agentes. Además, es independiente de la arquitectura de agente, es decir, el desarrollador puede desarrollar nuevas arquitecturas e incorporarlas en el marco de trabajo. Esto es una clara diferencia respecto a otros marcos de trabajo de agentes tales como JACK o JADE [2], los cuales proporcionan un middleware, en lugar de una arquitectura extensible, para establecer comunicaciones entre los agentes. Otras ventajas adicionales son las funcionalidades que ya están implementadas en el marco de trabajo para realizar la gestión de componentes, la inicialización y el cierre de la aplicación, reduciendo de esta forma el trabajo que tiene que hacer el desarrollador cuando aborda la tarea de implementación y garantizando que los componentes están bajo control. Estas últimas funcionalidades, usualmente, no las ofrecen otros marcos de trabajo.

3. Caso de Estudio: Robots Móviles Colaborativos

El caso de estudio seleccionado para ilustrar el desarrollo de aplicaciones de vigilancia basadas en robots, siguiendo el proceso de la metodología *VigilAgent*, aborda la colaboración entre varios robots móviles para llevar a cabo una tarea de vigilancia común en un polígono industrial, tal y como se describe a continuación.

Los robots navegan aleatoriamente por rutas de vigilancia predefinidas en un entorno simulado. Cuando se produce una alarma en un edificio, a un robot se le asigna el rol de jefe, tres robots serán subordinados y el resto estarán esperando

en la retaguardia para recibir órdenes del jefe (por ejemplo para reemplazar a un robot subordinado averiado). Los fallos los descubre el propio robot cuando alguno de sus dispositivos (e.j. sonar, laser, cámara, etc) no funciona de forma correcta. Cuando no hay alarma o la alarma ha sido cubierta los robots tienen asignado el rol denominado "Patrulla".

Los robots pueden percibir que ha ocurrido una alarma de dos formas diferentes: (1) el vigilante de seguridad notifica a los robots que ha ocurrido una alarma y dónde ha tenido lugar, (2) el robot está equipado para percibir una alarma cuando está suficientemente cerca a la esquina de un edificio y por lo tanto no tiene que esperar a que lo anuncie el vigilante. La alarma se cubre cuando una coalición de robots (un jefe y tres subordinados) rodean el edificio, es decir, los robots que forman la coalición se encuentran en las cuatro esquinas del edificio donde ha ocurrido la alarma.

En este caso de estudio se asumen las tres hipótesis siguientes: (1) no pueden ocurrir simultáneamente varias alarmas, (2) las calles son lo suficientemente anchas para evitar que los robots puedan chocarse y (3) los robots se mueven saltando de esquina a esquina.

3.1. Especificación del Sistema

El analista inicia la fase de especificación del sistema desarrollando un *diagrama de visión general del análisis*, el cual muestra un esbozo de las interacciones que se producen entre el sistema y el entorno (véase Fig. 1a). A este nivel, en primer lugar, se ha identificado un actor *SensorAlarma* que representa el dispositivo montado sobre el robot que detecta la alarma. Además, hay un actor *IGU* para representar la interfaz gráfica de usuario que soporta la interacción humana con el sistema, es decir, muestra al guarda de seguridad la actividad de monitorización, y los comandos que puede enviar al sistema para notificar una alarma, simular el fallo de un robot, etc. Por una parte, la información que proviene del entorno se identifica como una percepción. Por ejemplo, el comando emitido por el guarda de seguridad para notificar a los robots que se ha disparado una alarma (*guardaDetectaAlarma*) y la señal capturada automáticamente por el robot cuando está cerca del edificio con alarma (*robotDetectaAlarma*). Por otra parte, toda operación realizada por el sistema sobre los actores se identifica como una acción. Por ejemplo, un mensaje de alerta mostrado en la interfaz de usuario para notificar que se ha atendido una alarma (*NotificarAlarmaAtendida*). Finalmente, se establecen relaciones con los escenarios identificados para navegar por el entorno cuando hay alarma y cuando no (véase los escenarios *Alarma* y *Patrullando*, respectivamente).

Cada escenario se detalla en el *diagrama de escenarios* mediante una secuencia de pasos - etiquetados como una acción (A), una percepción (P), un objetivo (O), u otro escenario (S) - que representa una posible ejecución del sistema. Por ejemplo, la Fig. 1b ilustra el proceso realizado por el sistema para atender una alarma. Este escenario comienza cuando se ha producido una alarma y se ha percibido utilizando el sensor instalado en el robot (paso 1). Alternativamente, el escenario también puede iniciarse con la percepción

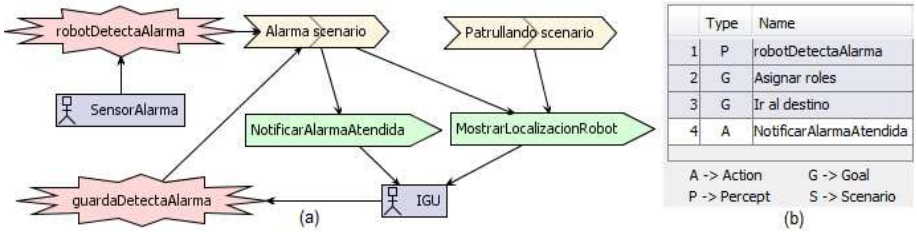

Figura 1. (a) Diagrama de visión general del análisis; **(b)** Escenario *Alarma*

guardaDetectaAlarma. Después, se inicia el proceso para asignar roles a cada robot con el fin de tratar la alarma (paso 2). Posteriormente, los robots que tienen el rol de jefe y subordinado se dirigen hacia la esquina destino que tienen asignada (paso 3). El escenario termina mostrando un mensaje al usuario cuando los cuatro robots están rodeando el edificio en el que se produjo la alarma (paso 4).

Resaltar que un escenario tiene un objetivo representando la meta a alcanzar en el escenario. Este objetivo se descompone en nuevos subobjetivos para denotar cómo alcanzar el objetivo padre. Por ejemplo, el objetivo general *Alarma* relacionado con el escenario *Alarma* se ha refinado en dos objetivos (*Asignar roles* e *Ir al destino*).

3.2. Diseño Arquitectónico

Una tarea relevante en la fase de diseño arquitectónico es definir las entidades de conversación (protocolos de interacción, PI) para describir qué debe ocurrir para alcanzar los objetivos y escenarios especificados. En este caso de estudio, los PI se utilizan para representar gráficamente (i) las interacciones entre robots, y (ii) las interacciones entre un robot y el entorno. Por ejemplo, la Fig. 2 muestra la estructura interna del protocolo de interacción *Alarma_PI*. Como se puede observar, están implicados dos actores (representados en el diagrama con cuadrados punteados) y los roles que pueden desempeñar cuatro robots para hacer frente a una situación de alarma. La etiqueta LOOP que aparece en la esquina superior de las cajas denota que el contenido de la región que aparece dentro de la caja se repite mientras se cumple la condición expresada entre corchetes. La etiqueta OPT indica que el contenido de la caja se ejecuta una vez si se satisface la condición expresada. La etiqueta ALT se utiliza para representar que sólo se ejecutará una de las regiones incluidas dentro de la caja ALT. En primer lugar, el robot está patrullando y captura una percepción enviada por el actor *IGU* o *SensorAlarma* cuando se dispara la alarma. Esta percepción contiene el identificador del edificio en el que se ha producido la alarma. En segundo lugar, si se trata del robot más cercano al edificio entonces se envía a si mismo un mensaje *designaJefe* con el fin de convertirse en el robot jefe. Seguidamente, por una parte, el jefe envía un mensaje *designaSubordinado* a los siguientes tres robots más cercanos para convertirlos en robots subordinados

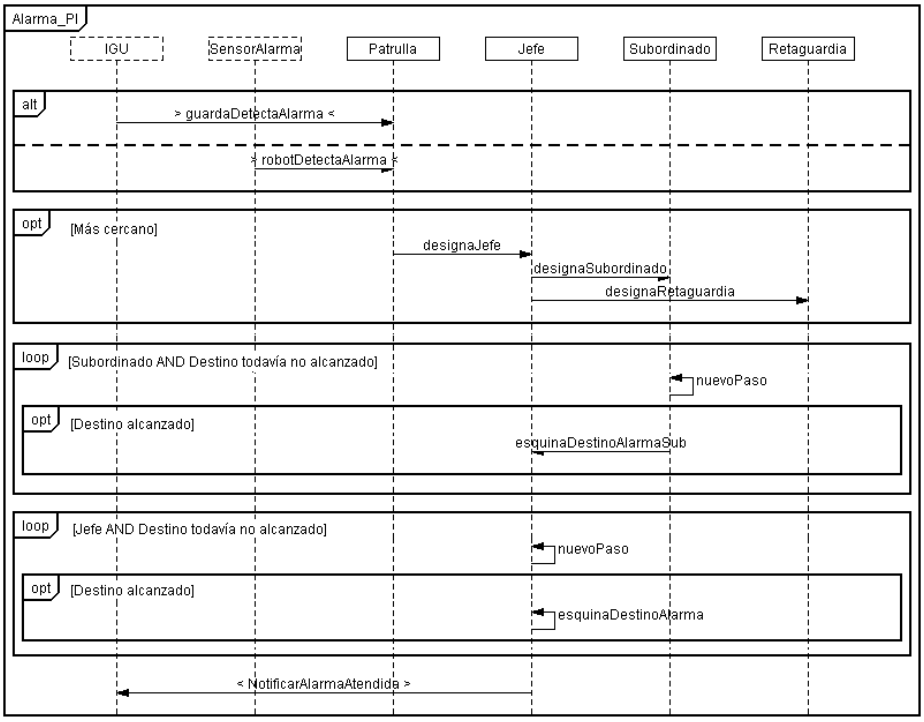

Figura 2. Protocolo de interacción Alarma

y notificarles la esquina destino hacia la que deben dirigirse. Por otra parte, envía un mensaje *designaRetaguardia* a los demás robots. A partir de este momento, cada robot subordinado está continuamente enviándose a sí mismo un mensaje *nuevoPaso*, para moverse hacia su esquina destino, hasta que alcance la esquina destino asignada; lo cual se comunica al robot jefe enviando un mensaje *esquinaDestinoAlarmaSub*. Se sigue una aproximación similar para que el jefe vaya hacia la esquina destino asignada al jefe. Finalmente, el jefe muestra un mensaje de texto en la *IGU* para comunicar que se ha atendido la alarma (cuatro robots rodean el edificio). Además, se han especificado dos protocolos para describir la gestión de fallos y las rondas realizadas por los robots cuando no hay alarma, respectivamente.

Finalmente, otra de las tareas realizadas durante la fase de diseño arquitectónico es la identificación de la información gestionada por los agentes o las creencias que describen el conocimiento que tienen los agentes sobre el entorno o de sí mismos. Por ejemplo, se utiliza el dato *LocalizacionRobot* para almacenar la ubicación de los robots. El dato *Entorno* ofrece información sobre el entorno simulado (las dimensiones del polígono industrial, el edificio en el que ha tenido lugar la alarma, los robots que no funcionan bien y la ubicación inicial de los robots). Señalar que, en este caso, estos datos son públicos para todas las

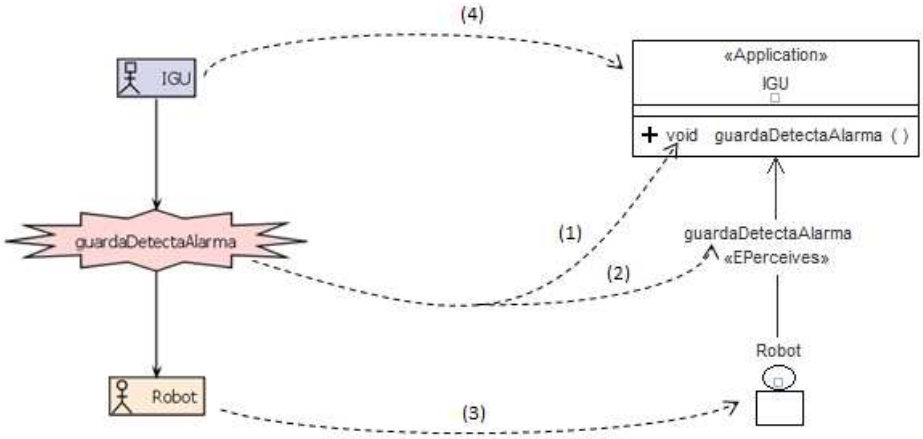

Figura 3. Transformando información relacionada con las percepciones de Prometheus en INGENIAS

instancias de los agentes, es decir, cualquier agente (robot) conoce la ubicación de los demás y la información sobre el entorno.

3.3. Diseño Detallado

En la fase de diseño detallado los modelos de *VigilAgent* se especifican utilizando conceptos de INGENIAS, los cuales son diferentes de los utilizados en las fases anteriores. Por ello, para poder avanzar en el modelado es necesario transformar información obtenida previamente a términos de INGENIAS. Se plantean cuatro correspondencias conceptuales [12] para transformar estructuras en las que aparecen percepciones, acciones, mensajes y datos relacionados con los agentes. Estas correspondencias se han inferido teniendo en cuenta la definición de estos conceptos y cómo se puede modelar cada estructura de Prometheus con una estructura de INGENIAS equivalente.

Por ejemplo, una percepción es una pieza de información del entorno recibida por medio de un sensor. Las percepciones las envían los actores (Actor $\rightarrow$ Percepción) y las reciben los agentes (Percepción $\rightarrow$ Agente). En INGENIAS, cualquier software o hardware que interactua con el sistema y que no puede diseñarse como un agente se considera que es una aplicación, y cada agente que percibe cambios en el entorno debe estar en el *modelo del entorno* asociado a una aplicación. Por lo tanto, tal y como se muestra en la figura 3 (flecha 1), las percepciones de un agente de Prometheus pueden dispararse en INGENIAS especificando en una *aplicación* una colección de *operaciones*. En Prometheus, una percepción tiene un campo *Information carried* para expresar la información que contiene. En INGENIAS, como se muestra en la figura 3 (flecha 2), esta información se escribe en un tipo de evento denominado *ApplicationEventSlots* asociado a la relación *EPerceives* establecida entre el agente y la aplicación

correspondiente. Obsérvese que los conceptos de agente y actor de Prometheus se traducen en los conceptos de agente y aplicación de INGENIAS (véase las flechas 3 y 4, respectivamente).

Una vez que se ha realizado la transformación se deben llevar a cabo nuevas actividades para completar el modelado. En primer lugar, es necesario identificar las tareas realizadas por los agentes por cada percepción o mensaje recibido. Después se especifica el comportamiento de cada agente utilizando información de las tareas a ejecutar, las percepciones y los mensajes recibidos. El comportamiento se modela con un autómata, en el que los estados representan situaciones concretas del ciclo de vida del agente. Por ejemplo, el diagrama de estados correspondiente al agente reactivo que controla un robot se interpreta como sigue (véase Fig. 4) - se utiliza el término evento para hacer referencia a una percepción o a un mensaje.

- Hay tres tipos de estados: inicial (*EstadoInicial*), final (*EstadoFinal*) e intermedio (e.j., *DeteccionAlarma*, *Retaguardia* ...).
- El agente comienza la ejecución cuando recibe un evento comenzar. Esto provoca que el agente cambie al estado *RevisarPoligono* y ejecute la acción *inicializarRobot*. El agente no consulta si hay eventos asociados con las transiciones que salen del estado *RevisarPoligono* hasta que haya acabado de ejecutar la acción. Una vez que se ha inicializado el agente, navega aleatoriamente (iterando en el estado *RevisarPoligono*) hasta que se dispara una alarma (llega un evento *robotDetectaAlarma* o *guardaDetectaAlarma*). En ese caso, se ejecuta la acción *notificarPosicion* para determinar quién es el jefe. Después, en la acción *asignarRoles* se asigna a los demás agentes los roles de subordinado y retaguardia. El agente que tiene el rol de jefe entra en los estados delimitados por la frontera 'Rol Jefe' mientras que los agentes con otros roles van a los estados delimitados por la frontera 'Roles Retaguardia y Subordinado'.
- Hay un tipo particular de transición, la transición universal, que es válida para cualquier estado del autómata. Una transición universal tiene lugar para un evento dado; la acción se ejecuta y el autómata transita al siguiente estado, independientemente del estado en el que se encontraba el autómata. En este caso de estudio sólo hay una transición universal la cuál se ha representado con una nota en la parte inferior derecha de la Fig. 4. Significa que si hay un evento *reinicia* en la primera posición de la cola de eventos del agente entonces se ejecutará la acción *reiniciar* y se pasará después al estado *RevisarPoligono*, independientemente del estado en el que se encontrara previamente el agente.
- En la acción *inicializarRobot* el agente se envía a si mismo el evento *nuevoPaso* para empezar a mover el robot. La acción *finalizar* marca al agente como averiado. La acción *reiniciar*, incluida en la transición universal, reinicia el proceso de simulación, y después de hacer esto, el agente se envía a si mismo un evento *nuevoPaso*. El resto de las acciones se describen en la Tabla 1, mencionándose el rol jugado por el agente solamente cuando es significativo.

Cuadro 1. Descripción de las acciones del autómata

Acción	Descripción
mover	El agente se envía a sí mismo un evento *nuevoPaso* si no hay alarma.
notificarPosicion	Determina si el agente se convierte en jefe. El jefe es el agente que más cerca está de la alarma y los empates se resuelven a favor del agente que tenga un índice menor. El agente se envía a sí mismo un evento *designaJefe* si se proclama jefe.
permanecerEnRetaguardia	El agente actualiza su rol como retaguardia y conoce quién es el jefe. El robot no se mueve mientras tiene este rol.
subordinar	El agente (1) actualiza su rol como subordinado y conoce quién es el jefe, y (2) se envía a sí mismo un evento *nuevoPaso*.
asignarRoles	El agente (1) actualiza su rol como jefe, (2) asigna a que esquina debe ir el jefe, (3) asigna a que esquinas deben ir los tres siguientes agentes más cercanos a la alarma y les envía un evento *designaSubordinado* que contiene la siguiente información: la esquina destino que deben ocupar para cubrir la alarma y quién es el jefe, (4) envía a los demás agentes un evento *designaRetaguardia* y finalmente (5) se envía a sí mismo un evento *nuevoPaso*. Consultar la acción *notificarPosicion* para conocer cómo se resuelven los empates.
irAEsquinaLibre	Si el agente jefe/subordinado está en la esquina destino asignada para cubrir la alarma entonces se envía a sí mismo un evento *esquinaDestinoAlarma*; en otro caso determina cuál es la esquina a la que se moverá a continuación y se envía a sí mismo un evento *nuevoPaso*.
notificarJefeError	Si el agente está en la retaguardia entonces envía un evento *errorRetaguardia* al jefe; mientras que si es un agente subordinado entonces le envía un evento *errorSubordinado* que contiene su número de identificación. En ambos casos, el agente marca el robot como averiado.
informarJefe	El agente subordinado envía al agente jefe un evento *esquinaDestinoAlarmaSub* que contiene su número de identificación.
marcarEsquinaDestinoAlcanzadaSub	El agente jefe (1) marca la esquina destino que ha ocupado el agente subordinado, (2) incrementa el número de esquinas ocupadas y (3) muestra un mensaje al usuario si se han ocupado las cuatro esquinas para tratar la alarma.
marcarEsquinaDestinoAlcanzadaJefe	El agente jefe (1) marca la esquina destino que ha ocupado, (2) incrementa el número de esquinas ocupadas y (3) muestra un mensaje al usuario si se han ocupado las cuatro esquinas para tratar la alarma.
seleccionarSubordinadoEnRetaguardia	El agente jefe (1) incrementa el número de robots averiados, (2) identifica el agente retaguardia más cercano a la esquina destino que debía ocupar el agente subordinado averiado y (3) envía un evento *designaSubordinado* que contiene la esquina destino que deberá ocupar para cubrir la alarma y el número identificador del jefe.
marcarFalloRetaguardia	El agente jefe (1) marca el agente retaguardia que envió el evento *errorRetaguardia* como averiado y (2) incrementa el número de robots averiados.
generarReasignacionRoles	El agente jefe (1) se marca a sí mismo como averiado, (2) incrementa el número de robots averiados y (3) envía al resto de agentes un evento *reasignaRoles* que contiene la localización del edificio dónde saltó la alarma.

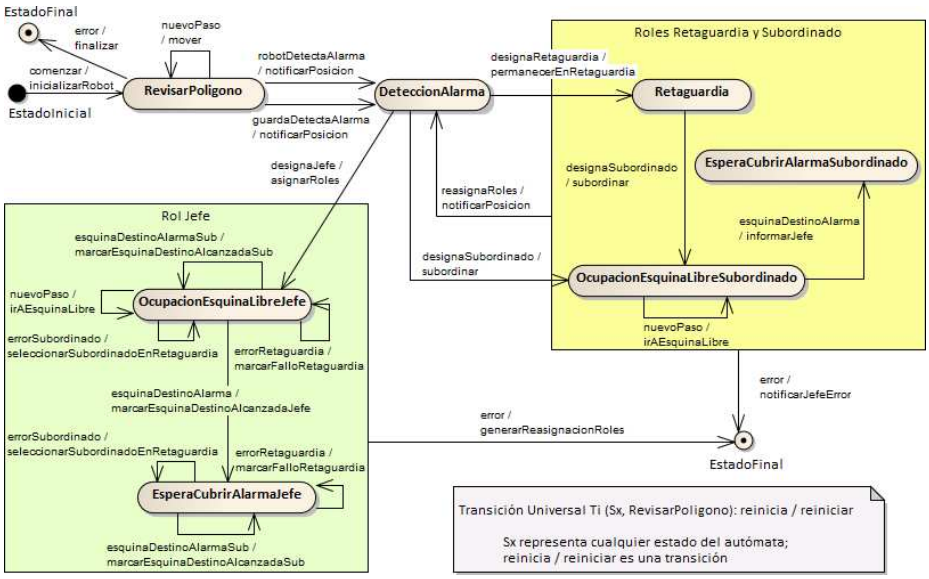

Figura 4. Diagrama de estados para controlar el comportamiento del robot

3.4. Implementación y Despliegue

En esta fase encontramos la segunda contribución importante de la metodología *VigilAgent*. Nuestro aproximación considera que la herramienta que ofrece soporte a INGENIAS, Ingenias Development Kit (IDK) [19], es una herramienta de agentes excepcional para desarrollar una transformación modelo a texto (M2T) para generar código para cualquier lenguaje destino elegido, ICARO-T en *VigilAgent*, porque proporciona las funcionalidades necesarias para desarrollar nuevos módulos capaces de llevar a cabo esta tarea. Estos módulos se desarrollan siguiendo un proceso general basado en la definición de plantillas específicas para cada plataforma destino y procedimientos para extraer información de los modelos de INGENIAS. En la literatura no existen propuestas para establecer correspondencias entre los modelos de INGENIAS y el código de ICARO-T. Así, nuestra contribución para cubrir este vacío ha sido precisamente su identificación y el desarrollo de módulos para realizar automáticamente la transformación M2T desde INGENIAS a ICARO-T.

Para facilitar la implementación y el despliegue, se han desarrollado módulos para generar automáticamente el código ICARO-T, del sistema que se está desarrollando, a partir de los modelos especificados con IDK. El proceso seguido para utilizar nuestros módulos se describe como sigue (véase Fig. 5). (1) Se utiliza el módulo generador INGENIAS ICARO-T Framework (IIF) para generar código automáticamente a partir de la especificación del diseño detallado. IIF genera varios ficheros XML que describen el comportamiento de cada agente, clases java para cada agente y aplicación, y el fichero XML que

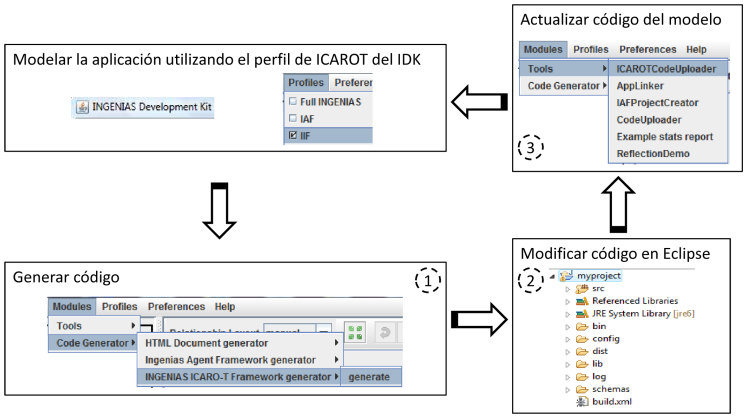

Figura 5. Utilización de módulos IDK

describe el despliegue de la aplicación. (2) El desarrollador inserta manualmente código en las regiones protegidas del código generado e implementa las nuevas clases que necesite. (3) El desarrollador utiliza el módulo ICAROTCodeUploader para actualizar el modelo con las modificaciones introducidas en las regiones protegidas. El proceso de actualización consiste en salvar en varios ficheros los contenidos de las regiones protegidas de tal forma que al volver a generar código utilizando el módulo IIF se puedan incluir en las regiones protegidas, a partir del contenido de los ficheros creados por el módulo ICAROTCodeUploader, los cambios que se han introducido manualmente. Finalmente, el gestor del despliegue ejecuta el script generado por el módulo IIF para lanzar la aplicación desarrollada para simular el sistema de vigilancia modelado.

4. Conclusión

Una vez que se ha descrito el desarrollo de una aplicación utilizando *VigilAgent* se pueden extraer las conclusiones siguientes. Al principio, los usuarios pueden requerir un cierto tiempo para aprender a utilizar *VigilAgent*, pues deben acostumbrarse a utilizar diferentes términos que tienen el mismo significado en función de de tecnología utilizada en cada fase (Prometheus e INGENIAS para el modelado e ICARO-T para la implementación). Sin embargo, este inconveniente se supera gracias a las dos transformaciones que se ejecutan automáticamente. La M2M se ha especificado en el lenguaje QVT [17] y el motor de Medini [15] la ejecuta. Esta transformación se ha especificado en base a dos metamodelos. Respecto a Prometheus, se ha desarrollado un metamodelo, en lenguaje Ecore, de los conceptos utilizados en la metodología Prometheus. Para ello, el punto de partida para describirlo ha sido la representación en UML del metamodelo de Prometheus [7] y nuestra experiencia adquirida para modelar aplicaciones basadas en agentes utilizando la herramienta Prometheus Design Tool que ofrece soporte a Prometheus [14]. Por otra parte, respecto a INGENIAS,

se ha utilizado el metamodelo de objetos de INGENIAS desarrollado en lenguaje Ecore disponible en la herramienta INGENIAS Development Kit modernizada [16]. La M2T está implementada en los módulos desarrollados para el IDK.

Precisar que el tiempo invertido para aprender cómo desarrollar módulos del IDK e implementar los módulos *INGENIAS ICARO-T Framework generator* e *ICAROTCodeUploader*, citados en la sección 3.4, ha sido de dos meses y medio. Este esfuerzo se ve recompensado cuando se modelan e implementan nuevas aplicaciones. Por ejemplo, *VigilAgent* se ha validado también para modelar e implementar un sistema de control de entrada salida a un recinto [13].

Finalmente, comentar que a nuestra propuesta metodológica le hemos dado el nombre *VigilAgent* para resaltar una de los objetivos principales de nuestras líneas de investigación: desarrollar sistemas de vigilancia (Vigil) utilizando tecnologías de agentes (Agent). Sin embargo, es conveniente señalar que la propuesta se puede aplicar a otros dominios de aplicación debido a que las tecnologías utilizadas son de propósito general, es decir, no están ligadas a un dominio específico. Por lo tanto, en el futuro se prevé continuar con la fase de experimentación para aplicar la metodología *VigilAgent* a otros nuevos casos de estudio o dominios.

5. Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación bajo los proyectos TIN2010-20845-C03-01 y CENIT A-78423480, y por la Junta de Comunidades de Castilla-La Mancha bajo los proyectos PII2I09-0069-0994 y PEII09-0054-9581.

Referencias

1. D.A. Anisi, J. Thunberg. Survey of patrolling algorithms for surveillance UGVs. Scientific report, 2007. Available at `www.math.kth.se/~anisi/articles/CTAPP_survey.pdf`
2. R.H. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni. Multi-Agent Programming: Languages, Platforms and Applications. Springer, 2005.
3. A. Collinot, A. Drogoul, P. Benhamou. Agent oriented design of a soccer robot team. In *2nd International Conference on Multiagent Systems* pp. 41-47, 1996.
4. M. Cossentino, L. Sabatucci, A. Chella. A possible approach to the development of robotic multiagent systems. In *IEEE/WIC Conference on Intelligent Agent Technology*, pp. 539-544, 2003.
5. M. Cossentino, S. Gaglio, A. Garro, V. Seidita. Method fragments for agent design methodologies: From standardisation to research. International Journal of Agent-Oriented Software Engineering, 1(1), pp. 91-121, 2007.
6. K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. IBM Systems Journal, 45, pp. 621-646, 2006.
7. K.H. Dam. Supporting software evolution in agent systems. PhD thesis, Computer Science and Information Technology, Science, Engineering, and Technology Portfolio, RMIT University (Australia), August 2008.

8. S. DeLoach, E. Matson, Y. Li. Applying agent oriented software engineering to cooperative robotics. In *The Fifteenth International Florida Artificial Intelligence Research Society Conference*, pp. 391-396, 2002.
9. A. Fernández-Caballero, J.C. Castillo, J. Martínez-Cantos, R. Martínez-Tomas. Optical flow or image subtraction in human detection from infrared camera on mobile robot. Robotics and Autonomous Systems, 58 (12), pp. 1273-1280, 2010.
10. A. Fernández-Caballero, J.M. Gascueña. Developing multi-agent systems through integrating Prometheus, INGENIAS and ICARO-T. Communications in Computer and Information Science (Agents and Artificial Intelligence), 67, pp. 219-232, 2010.
11. F.J. Garijo, F. Polo, D. Spina, C. Rodríguez. ICARO-T User Manual. Technical Report, Telefonica I+D, 2008.
12. J.M. Gascueña, A. Fernández-Caballero. Prometheus and INGENIAS agent methodologies: A complementary approach. 9th International Workshop on Agent-Oriented Software Engineering, AOSE 2008, Lecture Notes in Computer Science 5386, pp. 131-144, 2009.
13. J.M. Gascueña, A. Fernández-Caballero, E. Navarro. VigilAgent Methodology: Modeling Normal and Anomalous Situations. Highlights in Practical Applications of Agents and Multiagent Systems, Advances in Intelligent and Soft Computing, Vol 89, pp. 27-35, 2011.
14. J.M. Gascueña, A. Fernández-Caballero. Agent-oriented modeling and development of a person-following mobile robot. Expert Systems with Applications 38(4), pp. 4280-4290, 2011.
15. Medini QVT: IKV++ Technologies Home, Available at http://www.ikv.de/, last visited on March 2011.
16. INGENIAS Development Kit modernizada, Available at http://grasia.fdi.ucm.es/main/myfiles/ingenias-2_6-emf.zip, last visited on June 2011.
17. Object Management Group. Meta object facility (MOF) 2.0 query/view/transformation specification, version 1.0. OMG Document Number formal/2008-04-03, 2008. Available at http://www.omg.org/spec/QVT/1.0/PDF
18. L. Padgham, M. Winikoff. Developing Intelligent Agents Systems: A Practical Guide. John Wiley and Sons, 2004.
19. J. Pavón, J. Gómez-Sanz, R. Fuentes. The INGENIAS Methodology and Tools. Agent-Oriented Methodologies, Idea Group Publishing, pp. 236-276, 2005.
20. A. Kleppe, J. Warmer, W. Bast. MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise. Addison-Wesley, 2003.
21. A.S. Rao, M.P. Georgeff. Modeling rational agents within a BDI-architecture. In Proc. of the 2nd International Conference on Principles of Knowledge Representation and Reasoning, pp. 473-484, 1991.
22. J.A. Rogge, D. Aeyels. A strategy for exploration with a multi-robot system. Informatics in Control, Automation and Robotics, Lecture Notes in Electrical Engineering, 24, part II, pp. 195-206, 2009.
23. D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. Computer, 39(2), pp. 25-31, 2006.
24. G. Weiss. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. The MIT Press, 1999.
25. M. Wooldridge. Agent-based software engineering. IEE Proceedings - Software Engineering, 144(1), pp. 26-37, 1997.