ICARO-D

Infraestructura Ligera de Componentes Software Java basada Agentes y Recursos y Organizaciones para el desarrollo de Aplicaciones distribuidas (ICARO-D)

MANUAL DE USO DE LOS COMPONENTES AGENTE "COGNITIVO" DIRIGIDO POR OBJETIVOS

Autores	Francisco J Garijo

Distribución:	Público
EstatusDocumento :	Pre-release
Fecha Elaboración :	12 de Julio 2014

Revisiones

Version	Fecha	Autor	Descripción de Cambios

0.1		F Garijo	
0.7	19/03/2015	F Garijo	Adaptación curso DASI 2014-15
0.8	24/04/2015	F Garijo	Introducción de un Ej Formar un equipo de Agtes

INDICE

TABLA DE FIGURAS

1 Introducción

El presente documento es un complemento al manual de ICARO. Está dirigido a los desarrolladores que decidan utilizar el patrón de agente "cognitivo" incluido en la infraestructura, para la implementación de aplicaciones. En este apartado se introducen las características del patrón y sus diferencias con le modelo BDI. En el capítulo 2 se describe la arquitectura del patrón y su funcionamiento con un ejemplo sencillo– el sistema de acceso – ya utilizado para el patrón de agente reactivo. En el capítulo 3 se detallan las principales interfaces de los componentes. El capítulo 4 se dedica documentar la experiencias de los usuarios; se describen los problemas más frecuentes, y las recomendaciones para resolverlos. Este documento es una versión inicial que debe ir mejorando con las aportaciones de los usuarios.

1.1 Características del patrón de Agente "cognitivo" Dirigido por Objetivos (ADO)

El patrón tiene las mismas interfaces que el patrón de agente reactivo, pero se diferencia de él en la interpretación y el procesamiento de la información que llega a través de la percepción. El núcleo del patrón arquitectónico esta formado por **un Procesador de Objetivos y Tareas**. El patrón proporciona elementos computacionales (clases, threads, reglas) para representar:

- Los Objetivos de un agente. La representación simbólica de las metas que el agente pretende conseguir.
- Las Tareas que es necesario realizar para conseguir los objetivos. Las tareas son procedimientos o procesos que se seleccionan y se ejecutan para obtener la información que haga posible la consecución de los objetivos.
- La memoria interna del agente o el conjunto de informaciones que son consideradas para conseguir los objetivos.
- Las decisiones que definen el comportamiento del agente. Las decisiones se expresan con reglas situación-acción.
- El tratamiento de la información que se recibe por medio de mensajes y/o eventos externos, y su incorporación a la memoria interna del agente.

Parte de los conceptos utilizados en el patrón están inspirados en el comportamiento humano, sin embargo es necesario tener presente que **estamos tratando con una maquina de procesamiento de la información** cuyo funcionamiento tiene poco que ver con los modelos cognitivos y psicológicos.
En un principio este patrón se denominó *patrón de agente cognitivo* y se reclamó como un agente conforme al modelo BDI (Beliefs, Desires, Intentions) []. La utilización del patrón en el desarrollo de aplicaciones nos ha hecho reflexionar sobre el uso de conceptos psicológicos y antropomórficos en ingeniería, y sobre las consecuencias de crear expectativas exageradas en los desarrolladores. Como resultado hemos decidido llamar a las cosas por su nombre, y utilizar términos más modestos –Agente Dirigido por Objetivos- pero con un modelo computacional consistente y trazable.
Las diferencias respecto al modelo BDI son las siguientes:

- En lugar de creencia (que evoca connotaciones de tipo religioso) se ha preferido utilizar el término Modelo de Información del Agente. Este modelo esta formado por clases que pueden estar relacionadas o no. Podría llamarse Ontología (como se suele hacer en muchos casos) pero dado que en la mayoría de las aplicaciones no se han utilizado todos los elementos que caracterizan una ontología, consideramos más honesto llamarle como lo que es : un modelo de información.
- No hay Deseos en los agentes generados con el patrón ADO (a pesar de la D) (los sistemas que se reclaman BDI carecen de un soporte computacional para implementar los deseos de un agente).
- Tampoco hay un soporte explicito para modelar Intenciones. En su lugar se considera el conjunto de objetivos que un agente puede conseguir, y el ciclo de vida de cada objetivo que

debe definir el contexto y la acciones necesarias para que un objetivo sea creado, focalizado y que avance hacia su consecución hasta que finalmente se consiga.

- En lugar *de estado mental*, utilizamos el término *estado de la memoria de trabajo del procesador de objetivos.*

2 El patrón de Agente “Cognitivo” Dirigido por Objetivos (ADO)

En este apartado se introduce la arquitectura del patrón y a continuación su funcionamiento que se ilustrará con un ejemplo.

2.1 Arquitectura del patrón

En la **Figura 1** se presentan los principales componentes del patrón.

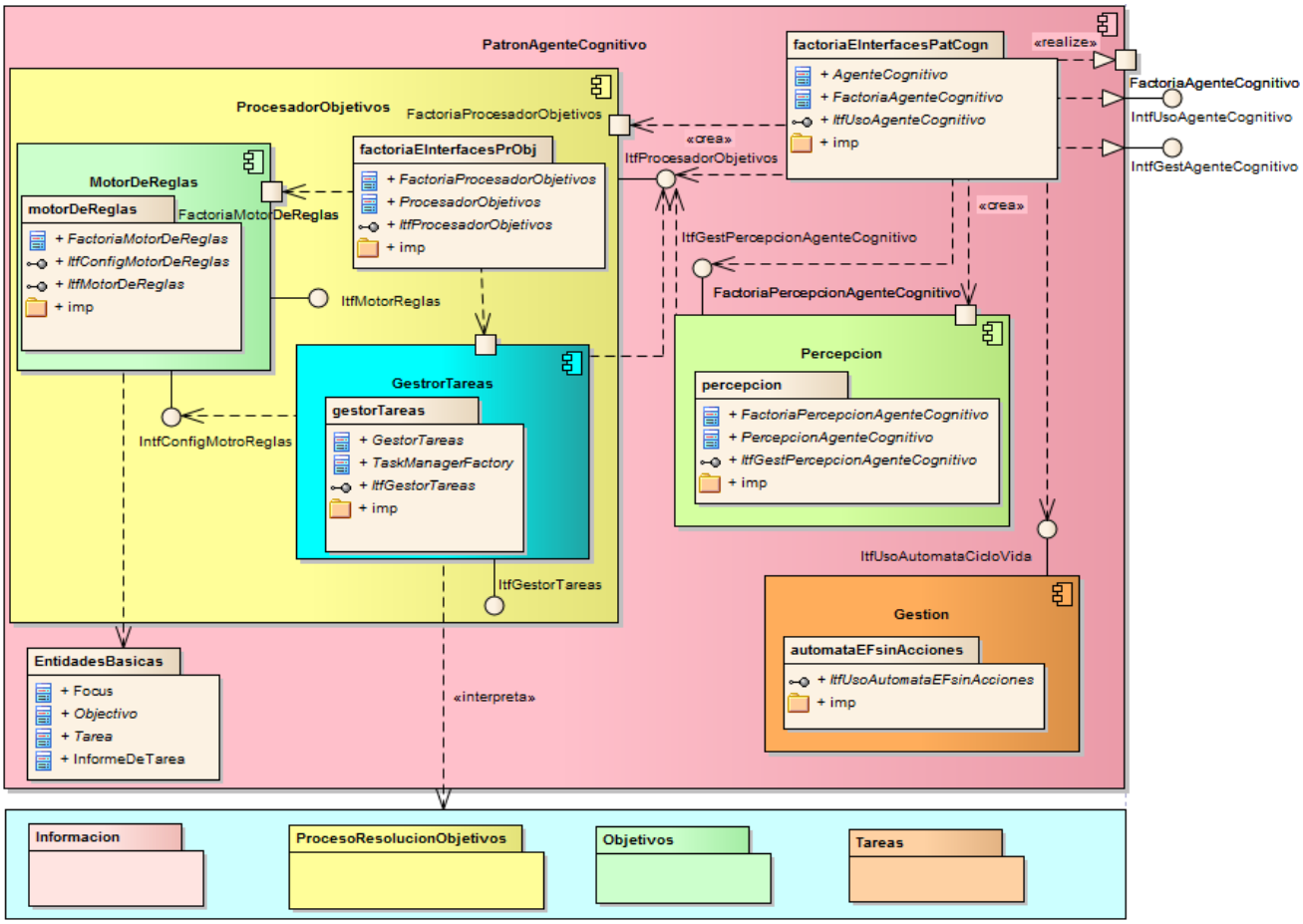


Figura 1: Arquitectura del patrón de Agente Dirigido por Objetivos (ADO)

La visión externa es similar a la del agente reactivo y a otros posibles modelos de agente en ICARO. Está formada por las interfaces de uso y de gestión

- La interfaz de gestión proporciona operaciones de arranque parada, monitorización y control
- La interfaz de uso es una interfaz de comunicación que permite recibir eventos o mensajes

Los componentes internos son los siguientes:

- **Gestión Agente.** Implementa las operaciones de la interfaz de gestión.
- **Percepción Agente.** Implementa las operaciones de la interfaz de comunicación. Se encarga de almacenar y de procesar los eventos y/o mensajes recibidos. El procesamiento básico consiste realizar filtrados, extraer la información contenida y enviarla al procesador de objetivos.

- **Procesador de objetivos.** Valida e interpreta el comportamiento del agente: objetivos, tareas y proceso de consecución de los objetivos. Se encarga de procesar la información procedente de la percepción y de utilizarla para resolver los objetivos. Está formado por dos componentes:
  1. **Motor de Reglas.** Interpreta el proceso de consecución de los objetivos e implementa el ciclo básico de procesador consistente en la generación de objetivos y el control de la consecución por medio de ejecución de tareas y la asimilación de la información externa.
  2. **Gestor de Tareas**. Ejecuta las tareas que indica el motor y controla su ciclo de vida.
  3. **Clases y tareas de infraestructura.** Implementan el ciclo de funcionamiento del procesador y sirven de base para las clases de dominio y para el control del proceso de consecución de los objetivos.

2.2 Ciclo de funcionamiento de un agente generado con el patrón.

El patrón permite crear agentes gestionables (son capaces de ejecutar operaciones de gestión en paralelo a su ciclo de funcionamiento normal). El ciclo básico de funcionamiento de un agente creado con el patrón ADO consiste en procesar los eventos y mensajes procedentes del exterior, extraer la información y asimilarla para conseguir los objetivos del agente. Este ciclo se implementa mediante la interacción entre **la percepción** y el **procesador de objetivos**.

La **Figura 2** describe con más detalle el flujo de información y las actividades implicadas.

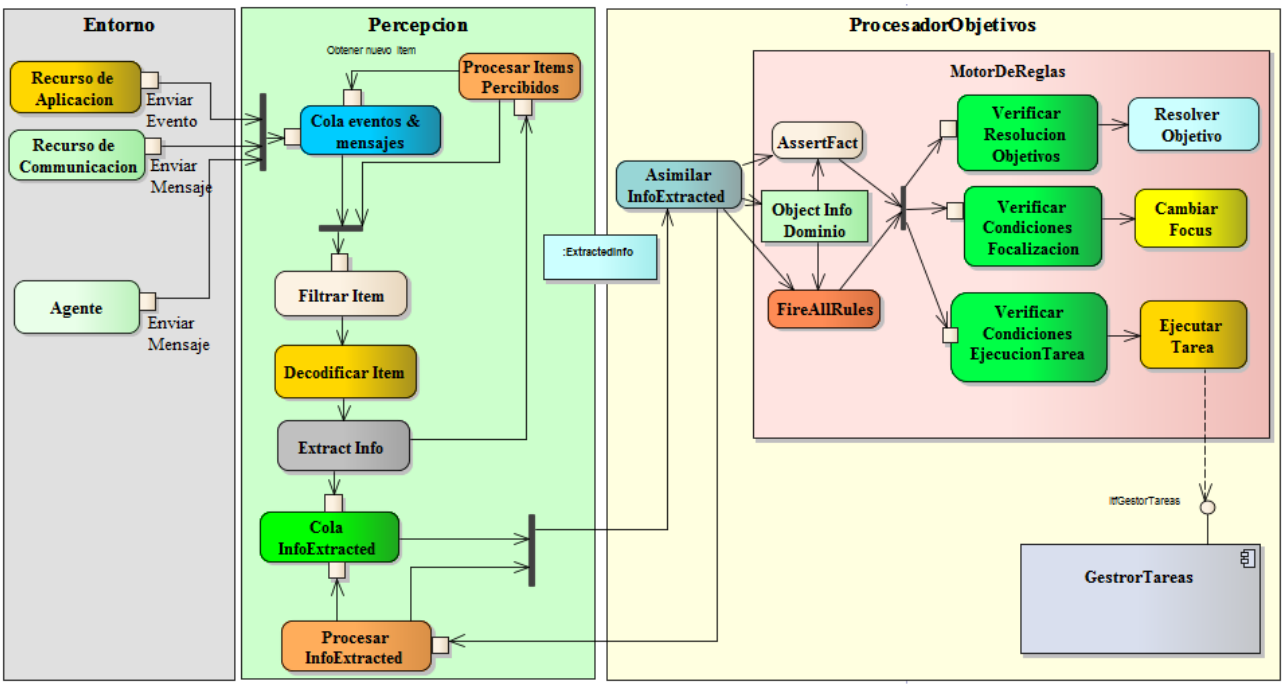


Figura 2: Ciclo de procesamiento de la información

La percepción realiza el siguiente ciclo:

- Encolar los eventos o mensajes recibidos través de la interfaz de uso del agente
- Procesar los ítems almacenados en la cola.
- Filtrar los ítems procesados. En general se eliminan los mensajes que no tienen el formato adecuado o que el identificador del receptor no coincide con el identificador del agente. Se pueden definir filtros específicos.
- Extraer la información contenida en los ítems que pasan el filtro, generando una información asimilable por el procesador de objetivos (*infoExtracted*). Para extraer la información se pueden

utilizar intérpretes específicos de los eventos o de los mensajes. Estos intérpretes producen objetos de la clase *infoExtracted.*

- Encolar la información extraída de los mensajes.
- Procesar la información almacenada en la cola. El procesamiento se limita a extraer un elemento de la cola, pero se pueden realizar operaciones más complejas como ordenación de los elementos con nuevos criterios, realizar correlaciones entre elementos extraídos, validaciones o transformaciones.
- Enviar la información obtenida (*objeto de la clase ExtractedInfo*), al procesador de objetivos para que pueda ser asimilada en el proceso de consecución de los objetivos.

El Procesador de objetivos se encarga de asimilar las informaciones enviadas por la percepción incorporándolas al proceso de consecución de los objetivos del agente.
La actividad de **"asimilar InfoExtracted"** adapta la información extraída de los eventos o de los mensajes a los requisitos del procesador de objetivos y envía el resultado al motor de reglas. La adaptación puede consistir en **verificar que la información recibida cumple determinadas condiciones**, **aceptarla tal cual** o en **transformarla en nuevas informaciones** teniendo en cuenta el contexto semántico, el contexto de ejecución u otros factores aplicativos. Esta situación se da por por ejemplo en la implementación de protocolos de comunicación entre agentes, donde se necesita traducir información entre diferentes dominios y establecer correlaciones con otras información existentes. Este tipo de ajustes se pueden implementar como extensiones de la actividad *asimilar InfoExtracted* de manera que los resultados lleguen elaborados al motor de reglas donde serán utilizados para la consecución de los objetivos.
En la **versión actual** del patrón cognitivo **se ha optado por lo más sencillo**: **no hacer ninguna transformación e introducir la información enviada por la percepción directamente en el motor.**
El motor de reglas. Esta concebido como una máquina abstracta con las siguientes características:

- **Proporciona interfaces para almacenar y gestionar información** en una memoria de trabajo que puede considerarse como la memoria (o estado global) del agente.
- **Interpreta la definición de los objetivos a resolver por el agente y los procesos de consecución de cada objetivo**.

Cada vez que el motor recibe nueva información a través de sus interfaces, se realiza un nuevo ciclo para comprobar el impacto de la información recibida en el proceso de consecución de los objetivos. Este ciclo consta de los siguientes pasos:(Figura 2)

- **Almacenamiento de la información.** Se añade la información a la memoria de trabajo del motor.
- **Verificación de condiciones.** Se comprueba si la nueva información hace que se cumplan:
  - **Los requisitos de consecución** de alguno de los objetivos.
  - **Las condiciones o el contexto para ejecutar acciones** que hagan progresar el proceso de consecución de algún objetivo.
  - **Las condiciones para seguir resolviendo** el objetivo actual (**condiciones de focalización)** o las de un nuevo objetivo.
- **Ejecución de Acciones.** En función de las condiciones satisfechas el motor ejecutará acciones de los siguientes tipos :
  - **Generación de objetivos.** Para que el agente funcione deben existir objetivos a conseguir. El patrón proporciona una clase Objetivo y varios mecanismos para generar las instancias que representan los objetivos a resolver: i) generación estática en la carga del fichero de reglas, ii) generación a partir de elementos guardados en sesiones anteriores (persistencia), iii) generación dinámica a partir de observaciones o condiciones del entorno. Se proporciona también un patrón de tarea que el usuario puede adaptar para definir las condiciones y los resultados del proceso de generación.
  - **Selección del objetivo a conseguir (Focalización).** Para controlar el proceso de consecución de los objetivos, se proporciona una primitiva de focalización que permite

seleccionar el objetivo a resolver de forma explicita o contextualizada. Un objetivo no avanza en el proceso de consecución si no esta focalizado.

- **Ejecutar tareas** –internas o externas- para intentar resolver un objetivo. Las tareas constituyen el mecanismo de actuación del agente. El motor ordena la ejecución de las tareas a un gestor de tareas que se encarga de crear la tarea, de iniciar su ejecución con los parámetros definidos por el motor. Las tareas pueden ejecutarse como hebras independientes y siempre producen resultados, que llegan al motor a través de la percepción del agente – tareas externas - , o directamente a través del procesador de objetivos – tareas internas- .
- **Cambio del estado de los objetivos.** Los objetivos tienen un ciclo de vida que evoluciona según el proceso definido para resolver el objetivo. Cuando se genera un objetivo su estado inicial es “pendiente de consecución” (pending) . El objetivo debe evolucionar a un estado “resuelto” (solved), donde se ha obtenido la evidencia o el estado computacional que justificaron su definición, pero puede que no llegue a este estado y se quede en otros posibles estados como son “resolviendose” (solving) o “fracaso” (failure) .

El motor esta implementado con sistema de reglas de producción basado en el algoritmo RETE (http://en.wikipedia.org/wiki/Production_system). Esto implica que el proceso de generación y de consecución de los objetivos se expresa mediante reglas <condición> → <acción>, y que el mecanismo computacional del motor es el de un sistema de producción. Actualmente se usa DROOLS (http://www.jboss.org/drools/) por tanto la sintaxis de las reglas es la que propone Drools que es compatible con el estándar JSR. En versiones anteriores del patrón se utilizaron otros motores como JESS (http://www.jessrules.com/) e ILOG Rules. La elección de Drools se debe a que es un sistema de código abierto java que se integra sin problemas con el resto de los componentes del patrón. Drools está bien documentado, tiene herramientas de ayuda al desarrollo y está soportado por una comunidad activa que mejora el motor y propone tecnologías alternativas al uso de reglas, que podrían ser consideradas para la consecución de los objetivos.

2.3 Ejemplo de uso del patrón: Sistema de acceso con un Agente controlador Dirigido por Objetivos.

En el manual de uso de ICARO se utiliza el sistema de acceso para ilustrar el uso y el funcionamiento del patrón de agente reactivo. En este ejemplo se utilizará el mismo problema y la misma arquitectura, pero se va a sustituir el agente reactivo por un agente cognitivo.

Recordamos la especificación funcional informal del sistema de acceso.

La funcionalidad consiste en pedir al usuario los datos de acceso (nombre de usuario y clave de acceso), y darle acceso si los datos son válidos. Se supone que los usuarios válidos han sido registrados previamente en el sistema por el administrador del mismo y están guardados en una base de datos.

La Figura 3 contiene la arquitectura del sistema de acceso con los componentes de ICARO.

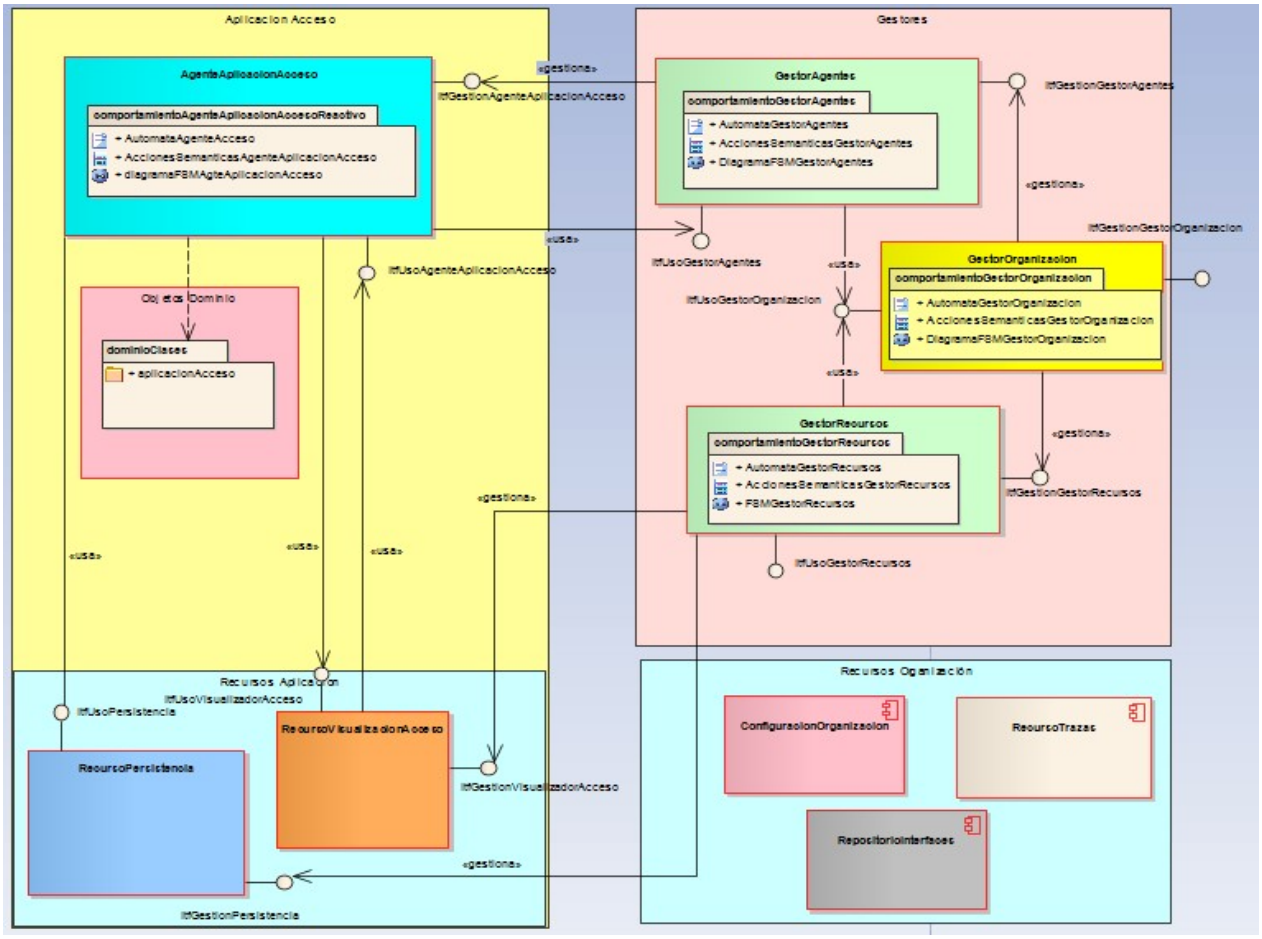


Figura 3: Arquitectura del Sistema de Acceso

- Los componentes gestores y de infraestructura en la parte derecha de la figura serán los mismos.
- Los recursos del sistema de acceso: Recurso de Persistencia y Recurso de Visualización permanecerán sin cambios.
- La visión externa del agente – interfaces de uso y de gestión - será la misma.
- El comportamiento del agente que controla el sistema de acceso (AgenteAplicacionAcceso) que esta basado en un modelo de Agente Reactivo se va a sustituir por un modelo de Agente Cognitivo ADO.

Para definir el comportamiento de un agente cognitivo se necesita especificar: Los **objetivos** del agente, las **tareas**, el **proceso de consecución de los objetivos** y las **clases de dominio**.

Los objetivos, las tareas y las clases de dominio son clases Java que dependen de la aplicación y que es necesario identificar e implementar. El **proceso de consecución de los objetivos se implementa con reglas de producción, <condicion> <accion>** donde en la parte condición se especifica las informaciones necesarias para que se ejecuten las tareas que harán avanzar los objetivos hacia su consecución. La interpretación y la ejecución de las reglas depende de la tecnología del motor de reglas. En nuestro caso Drools.

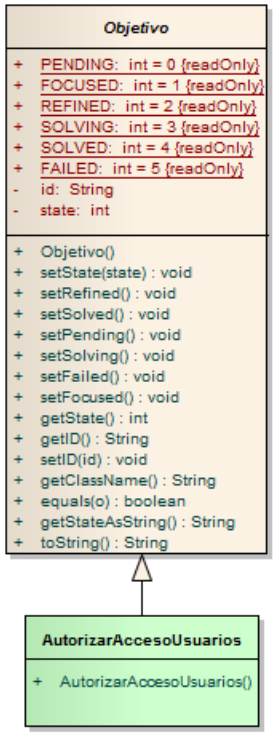
En los apartados siguientes se detalla la forma de identificar y de implementar Objetivos, Tareas, Clases de Dominio y Reglas para el ejemplo del sistema de acceso.

2.3.1 Identificación y definición de los objetivos del Agente de Acceso.

Para identificar los objetivos necesitamos responder a las preguntas: ¿Cual es el propósito del agente ? ¿Qué servicios debe proporcionar? ¿Qué es lo que tiene que conseguir el agente ? (una vez que empiece a funcionar).

La repuesta puede ser: *Autorizar a los usuarios a acceder al sistema* (si reúnen determinados requisitos como por ejemplo estar registrados). Esta respuesta puede ser suficiente para definir el primer objetivo del agente : ***AutorizarAcceso.***

> ***Para que el agente pueda utilizar el objetivo identificado lo definimos como una subclase de la clase objetivo y lo colocamos en la carpeta objetivos del comportamiento del agente.***



La clase Objetivo proporciona métodos para cambiar su estado que son heredados por los objetivos de la aplicación. Figura 4: Definición de un objetivo de Aplicación

La identificación del objetivo ***AutorizarAcceso*** nos debe llevar a plantearnos dos cuestiones:

- ¿Qué información debe conseguir o debe llegar al agente para que el objetivo se considere resuelto?
- ¿Cómo consigue el agente dicha información? ¿Qué tiene que hacer el agente para conseguirla ?

Para responder a estas preguntas es necesario ponerse en el lugar del agente, es decir asumir su comportamiento. Una vez creado, el agente debe realizar las siguientes acciones:

a) **Generar sus objetivos:** En el ejemplo debe crear un objeto instancia de la clase ***AutorizarAccesoUsuarios.*** Este objetivo se creará en el estado **pendig** (pendiente de consecución).

b) **Seleccionar el objetivo a conseguir**. En este caso ***AutorizarAccesoUsuarios***

c) **Comenzar a intentar conseguirlo actuando sobre los recursos o pidiéndoselo a otros agentes.** La capacidad de actuación del agente se modela mediante tareas. **Para conseguir un objetivo el agente ejecuta tareas**. Tras la ejecución de una tarea el agente debe:

d) **Verificar si con los resultados de las tareas ejecutadas se ha conseguido el objetivo.** En caso de que la información generada por la ejecución de las tareas **no permitan consiguir** el objetivo, se tendrá que seguir ejecutando otras tareas o dar el objetivo por fracasado.

En el ejemplo del sistema de acceso el agente puede comenzar con un único objetivo: ***AutorizarAccesoUsuarios.*** Cuando arranca, el agente genera el objetivo y lo selecciona para intentar conseguirlo dado que inicialmente no hay otros objetivos.

La cuestión siguiente es ¿**Qué tareas debe ejecutar el agente para que el objetivo *AutorizarAccesoUsuarios* avance hacia su consecución y al final se consiga?**

2.3.2 Identificación y definición de las tareas para conseguir los objetivos del Agente de Acceso.

La respuesta a la pregunta anterior es que **para autorizar el acceso al sistema** el agente debe :

1. Obtener el identificador y la clave de acceso introducidos por el usuario.
2. Validar los datos introducidos por el usuario.
3. Si los datos proporcionados son:
   a) **Válidos**:se informa al usuario de que puede acceder al sistema y el agente concede el acceso.
   b) **No válidos**: se informa al usuario de que sus datos no son correctos y se le invita a introducir nuevos datos para validarlos.

El agente es un componente controlador esto quiere decir que utiliza las capacidades de los recursos para conseguir sus metas.

- Para **obtener el identificador y la clave de acceso del usuario: El agente utilizará el recurso de visualización.** Accederá a su interfaz de uso para pedirle que muestre las ventana con la que el usuario pueda introducir los datos de autenticación. El recurso le enviará la información definida por el usuario mediante un evento.
  1. Definimos la tarea: **SolicitarDatosAcceso** que implementa las acciones anteriores: acceso a la interfaz de uso del recurso de visualización y ejecución de la operación *mostrarVisualizadorAcceso.* El resultado esperado de la ejecución de la tarea es la recepción de los datos de autenticación que introduzca el usuario, pero estos datos no llegarán hasta que el usuario decida teclearlos. El recurso de visualización de acceso enviará los datos en un evento, el agente los recibirá a través de la percepción y llegarán al procesador de objetivos en forma de creencias.
- Para **validar los datos introducidos por el usuario**. **El agente utilizará el recurso de persistencia**.
  - Definimos la tarea: **ValidarDatosAccesoUsuario.** Esta tarea obtiene la interfaz del recurso de persistencia y utiliza la operación *compruebaUsuario* para validar los datos introducidos por el usuario. La propia tarea obtiene los resultados de la validación y los envía al procesador de objetivos sin pasar por la percepción.
- Con los resultados producidos por la tarea **ValidarDatosAccesoUsuario,** el agente tiene que informar al usuario de que sus datos son correctos y permitirle el acceso o indicarle que sus datos son incorrectos y pedirle nuevos datos para validarlos. Definiremos dos tareas:
  1. **PermitirAcceso**. Se obtiene la la interfaz del recurso de visualización y se utiliza la operación *mostrarMensajeInformacion* para informar al usuario de que puede acceder al sistema.
  2. **DenegarAcceso**. Se obtiene la interfaz del recurso de visualización y se utiliza la operación *mostrarMensajeError* para informar al usuario de que los datos no son válidos. Se le invita a introducirlos de nuevo.

Las tareas se definen como subclases de la clase Tarea (Figura 5) y se incluyen en la carpeta Tareas.

Antes de entrar en detalles sobre la implementación de las tareas del sistema de acceso, resumiremos las ideas que hay que tener presentes para trabajar con Tareas.

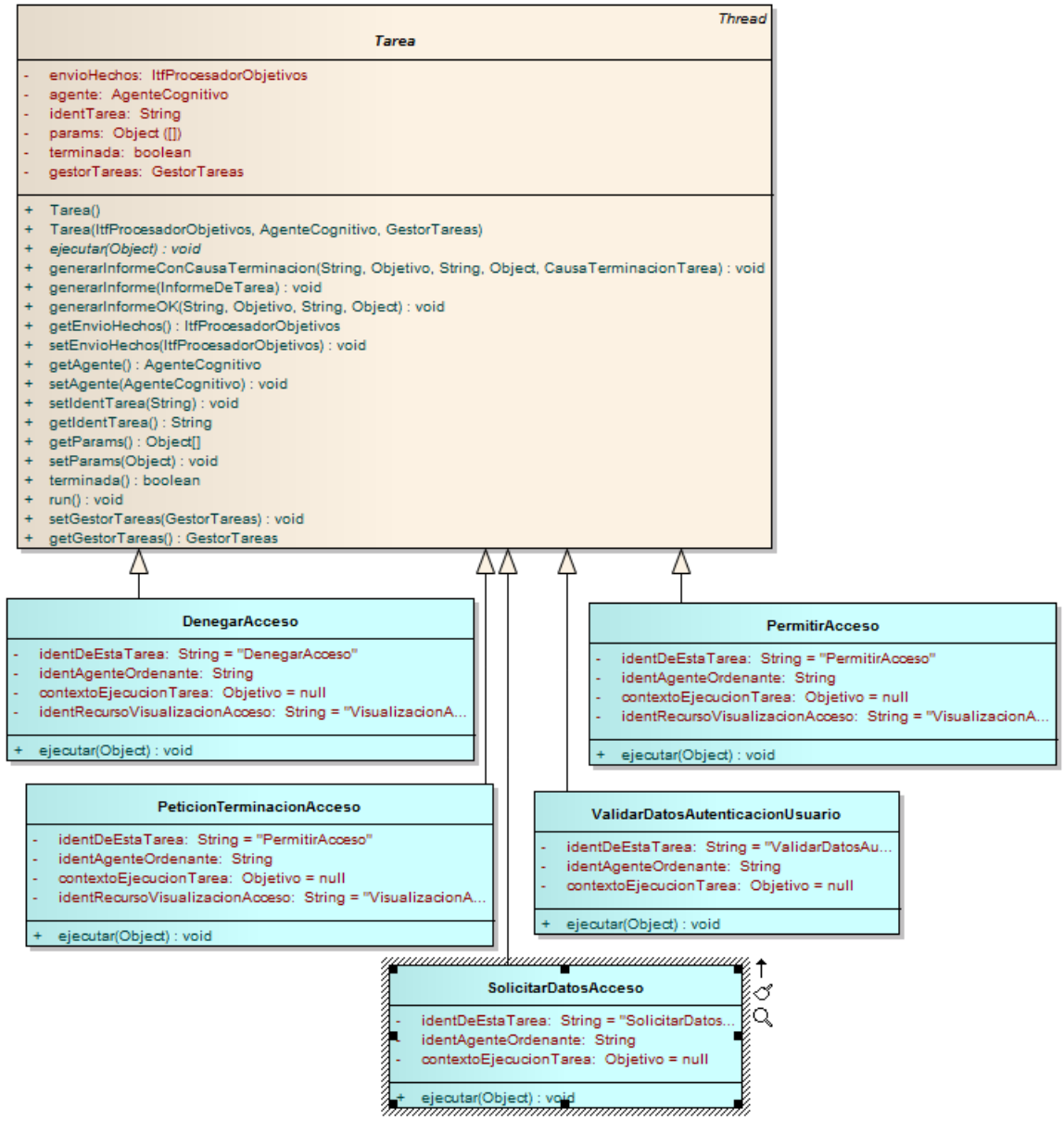


Figura 5: Definición de las tareas de la Aplicación

- Una tarea puede concebirse como un procedimiento o proceso que el agente ejecuta **para intentar conseguir información,** pero su ejecución no garantiza que la información se consiga.
- Las tareas se modelan como clases. Tienen parámetros y su ejecución genera resultados (objetos) que se envían al procesador de objetivos.
- Los resultados pueden ser objetos del dominio de aplicación, o informes donde se puede reportar las causas por las que la tarea no ha conseguido la información que se intentaba obtener.
- Las tareas pueden ejecutarse de forma síncrona o asíncrona como hebras. El Gestor de tareas es el encargado de su ejecución y de su gestión.
- El procesador de objetivos puede ordenar la ejecución concurrente de varias tareas para conseguir un objetivo.
- Las resultados producidos por las tareas llegan al procesador de objetivos de forma síncrona o asíncrona. Puede ocurrir:
  - Que los resultados lleguen cuando ya no se les espera.

- Que los resultados sean considerados en un contexto inadecuado o provoquen efectos colaterales que impidan conseguir otros objetivos.

Si los objetivos permiten representar explícitamente lo que el agente debe conseguir, las tareas representan las actividades o procesos necesarios para alcanzar el objetivo.

2.3.3 Definición del proceso de consecución de los objetivos

El proceso de consecución de los objetivos debe definir el ciclo de vida de cada objetivo, es decir las condiciones de creación, y su evolucionen desde su estado inicial hasta un estado en que se consideren alcanzados o resueltos. Para cada objetivo se deben especificar:

- Cuando debe crearse el objetivo (condiciones de generación)
- Qué información debe encontrarse en la memoria de trabajo para que se considere alcanzado o resuelto (condiciones de consecución).
- En qué circunstancias o contexto se debe iniciar el proceso de consecución del objetivo o interrumpirlo (condiciones de focalización).
- Qué acciones/ tareas deben ejecutarse para alcanzar o resolver el objetivo. (condiciones de intento de consecución).

Las condiciones de generación, de consecución, de focalización y de ejecución de tareas se expresan mediante reglas de producción <condición> → <acción>. En la parte condición se definen con expresiones lógicas el conjunto de informaciones que debe contener la memoria del agente (**estado de la memoria del agente)** para**:**

1. Generar objetivos.
2. Decidir qué objetivo realizar.
3. Seleccionar la tarea mas adecuada para intentar realizar un objetivo.
4. Determinar si se ha resuelto un objetivo o no se pueden realizar.

En la parte acción de la regla se utilizan primitivas para:

- Crear objetivos a realizar
- Especificar el objetivo que se desea realizar
- Cambiar el estado del objetivo que se esta realizando o que se pretende realizar
- Ejecutar tareas

El uso de reglas de producción viene determinado por la decisión de utilizar un motor de reglas de producción para implementar el proceso de consecución de los objetivos. La sintaxis y la interpretación de las reglas dependen de la tecnología del motor de reglas (en nuestro caso Drools), por tanto es necesario conocer la sintaxis del lenguaje de reglas de Drools y el mecanismo básico de interpretación de dichas reglas, para implementar el comportamiento del agente. (ver http://www.jboss.org/drools/drools-expert.html o directamente en : http://docs.jboss.org/drools/release/5.4.0.Final/drools-expert-docs/html_single/index.html).

Pero debemos subrayar que Drools es un componente al servicio del procesador de objetivos, por tanto las reglas deben limitarse a expresar las condiciones y las acciones para definir el proceso de consecución de los objetivos.

Es importante que las reglas esten estructuradas, sean legibles y que expresen situaciones y acciones sencillas y trazables.

Las reglas que implementan el proceso de consecución de los objetivos del agente de acceso están definidas en el fichero *reglasAgenteAcceso.drl* que se encuentra en la ruta:

icaro/agentes/ AgenteAplicacionAccesoADO/procesoConsecucionObjetivos

Se utilizará su contenido para explicar los tipos de reglas que modelan el comportamiento del agente.

Las 1 a 9 líneas del fichero indican la declaración de las clases de los objetos que puede gestionar el motor.

Las líneas 11,12 ,13 declaran variables globales de interfaces de la infraestructura a los que puede accederse desde el motor. Estas variables, dan acceso a las interfaces de las trazas y del gestor de tareas desde la parte derecha de las reglas .

```
1.  import icaro.infraestructura.recursosOrganizacion.recursoTrazas.ItfUsoRecursoTrazas;
2.  import icaro.infraestructura.recursosOrganizacion.recursoTrazas.imp.componentes.InfoTraza;
3.  import icaro.infraestructura.entidadesBasicas.procesadorCognitivo.*;
4.  import icaro.infraestructura.patronAgenteCognitivo.procesadorObjetivos.gestorTareas.ItfGestorTareas;
5.  import icaro.infraestructura.entidadesBasicas.PerformativaUsuario;
6.  import icaro.aplicaciones.agentes.AgenteAplicacionAccesoADO.objetivos.*;
7.  import icaro.aplicaciones.informacion.dominioClases.aplicacionAcceso.InfoAccesoSinValidar;
8.  import icaro.aplicaciones.informacion.dominioClases.aplicacionAcceso.VocabularioSistemaAcceso;
9.  import icaro.aplicaciones.agentes.AgenteAplicacionAccesoADO.tareas.*;
10.
11. global ItfGestorTareas gestorTareas;
12. global ItfUsoRecursoTrazas recursoTrazas;
13. global String agentId;
14.
15. rule "Creacion de los objectivos iniciales"
16. when
17. then
18.   TareaSincrona tarea = gestorTareas.crearTareaSincrona(InicializarInfoWorkMem.class);
19.   tarea.ejecutar();
20.   recursoTrazas.aceptaNuevaTrazaEjecReglas(agentId," EJECUTO LA REGLA: "+drools.getRule().getName());
21. end
```

Cuando se crea el agente su memoria de trabajo esta vacía. La tarea “InicializarInfoWorkMem “ permite crear objetos e introducirlos en la memoria de trabajo, Se puede además acceder a las operaciones de la interfaz de configuración del motor para definir el tipo de trazas que se desean obtener.

El código de la tarea es el siguiente.

```
public class InicializarInfoWorkMem extends TareaSincrona{
   @Override
   public void ejecutar(Object... params) {
     try {
        //     Objetivo objetivoEjecutantedeTarea = (Objetivo)params[0];
             String identTarea = this.getIdentTarea();
             String nombreAgenteEmisor = this.getIdentAgente();
             this.getItfConfigMotorDeReglas().setDepuracionActivationRulesDebugging(true);

this.getItfConfigMotorDeReglas().setfactHandlesMonitoring_afterActivationFired_DEBUGGING(true);
             this.getEnvioHechos().insertarHechoWithoutFireRules(new Focus());
             this.getEnvioHechos().insertarHechoWithoutFireRules(new AutorizarAccesoUsuarios());
      } catch (Exception e) {
                  e.printStackTrace();
                      trazas.aceptaNuevaTraza(new InfoTraza(this.getIdentAgente(), "Error al ejecutar
la tarea : "+this.getIdentTarea() + e, InfoTraza.NivelTraza.error));
      }
   }
}
```

La ejecución de esta tarea se expresa mediante la regla

```
rule "Creacion de los objetos iniciales"
when
then
    TareaSincrona tarea =
    gestorTareas.crearTareaSincrona(InicializarInfoWorkMem.class);
    tarea.ejecutar();
end
```

La regla expresa el hecho de que no es necesaria ninguna restricción ni condición a verificar para ejecutar la tarea InicializarInfoWorkMem. Cuando se crea el agente y arranca, el motor interpretará esta regla ejecutando la parte derecha (then), ya que las condiciones de su parte izquierda (when) son vacías y por tanto se satisfacen.

La ejecución de la tarea se realiza por medio del Gestor de Tareas. La ejecución de la tarea se implementa en dos pasos:

1. El gestor de tareas crea un objeto de la clase ***AutorizarAcceso*** *subclase de Tarea*
2. Se ejecuta el método ejecutar de la tarea creada

En su ejecución se accede a las interfaces del motor para :

1. Definir la información de depuracion :

```
this.getItfConfigMotorDeReglas().setDepuracionActivationRulesDebugging(true);
this.getItfConfigMotorDeReglas().setfactHandlesMonitoring_afterActivationFired_DEBUGGING(true);
```

2. Crear los objetos Focus y AutorizarAccesoUsuarios e introducirlos en la memoria de trabajo

```
this.getEnvioHechos().insertarHechoWithoutFireRules(new Focus());
this.getEnvioHechos().insertarHechoWithoutFireRules(new AutorizarAccesoUsuarios())
```

Una vez ejecutada la tarea la memoria de trabajo contiene los objetos de la clase *Focus* y de la clase objetivo *AutorizarAccesoUsuarios cuyo atributo* ***state*** *se ha inicializado a* ***pending***. En la *Figura 6*

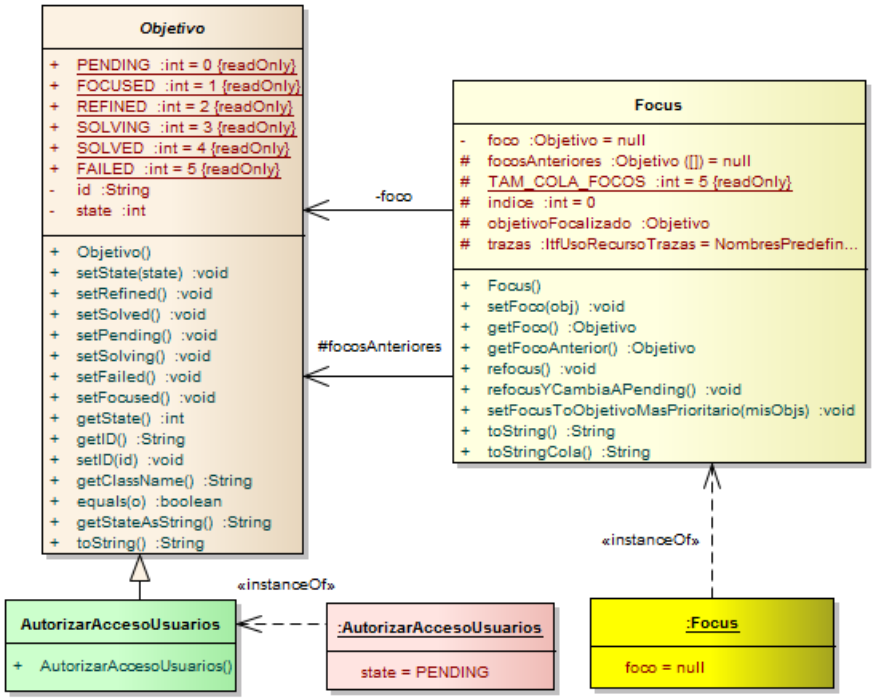


Figura *6: Objetos iniciales en la Memoria de Trabajo*

La clase Focus tiene un atributo foco que acepta como valores objetivos. Cuando se crea un objeto de la clase Focus el valor del atributo foco se inicializa a null. Esto puede interpretarse como que el agente no sabe qué objetivo va a comenzar a realizar.

- Si el atributo **foco** tiene como valor un objetivo X, (apunta al objetivo X), y el estado del objetivo es PENDING puede interpretarse como que **el agente ha decidido realizar el objetivo X** o que el agente **tiene la intención de realizar el objetivo X.**
- Si el agente está intentando realizar el objetivo X , es decir tiene en su memoria de trabajo un objeto f de la clase foco cuyo atributo foco apunta al objetivo X (f.foco == X) y decide realizar el objetivo Y, la decisión se implementa cambiando el foco del objetivo X al objetivo Y. Esto se realiza mediante la operación *setFoco. (f.setFoco(Y)).*

Podría decirse que **Focus es una clase que permite representar y gestionar las intenciones** del agente.

El hecho de que la tarea sea síncrona implica que el motor esperará a que la ejecución de la tarea termine para iniciar un nuevo ciclo consistente en : 1) obtener las reglas cuyas condiciones se satisfacen a partir de los cambios que se han producido en la memoria de trabajo; 2) actualizar la agenda de reglas ejecutables añadiendo las reglas cuyas condiciones se satisfacen y eliminado aquellas cuyas condiciones han dejado de satisfacerse; 3) seleccionar una regla y 4) ejecutar las acciones de su parte derecha. Para más detalles sobre el ciclo de funcionamiento del motor ver manual de usuario de Drools pp ()

Las siguientes reglas definen el proceso de consecución del objetivo *AutorizarAccesoUsuarios*.

a) ***reglas para expresar las condiciones para comenzar a conseguir el objetivo***

Para comenzar la consecución de un objetivo se necesita ***poner el foco en el objetivo*** (o que el agente decida intentar conseguir el objetivo) esto se expresa en la regla:

```
22. // ================================================================
23. //Reglas de focalizacion.
24. rule "Regla de focalizacion en objetivo AutorizarAccesoUsuarios"
```

```
25.  when
26.     f:Focus(foco == null )
27.     obj:AutorizarAccesoUsuarios()
28.  then
29.     f.setFoco(obj);
30.     recursoTrazas.aceptaNuevaTraza(new InfoTraza(agentId,"Foco: Focalizando el objetivo
    "+obj.getgoalId(),InfoTraza.NivelTraza.debug));
31.     update(f);
32. end
```

Esta regla puede también interpretarse de la forma siguiente :

Si el agente no tiene decidido ***qué objetivo va a realizar*** *(Focus(foco == null), y se ha generado el objetivo AutorizarAcceso() (no importa su estado)*
Entonces
el agente decide considerarlo para empezar a conseguirlo.
El agente ***cambia su foco (intención)*** *que pasa de* ***indefinida*** *a estar focalizada* ***en el objetivo*** *AutorizarAcceso().*

c) regla para expresar el inicio del proceso de realización del objetivo

Cuando se crea una instancia del objetivo Autorizar Acceso() su estado interno es pending (pendiente de consecución). Para que un objetivo comience a realizarse se necesitan dos condiciones:

- Que el agente tenga la intención de realizar el objetivo. El foco debe apuntar a ese objetivo o el objetivo debe estar focalizado.
- Que su estado sea solving.

En la regla anterior se han definido las condiciones de focalización. En esta regla se definen las condiciones para que el objetivo pase de pending a solving.

```
33. //=========================================================================
34. ///Reglas de Consecucion del Objetivo AutorizarAccesoUsuarios
35. rule "Inicio del proceso de consecucion del  objetivo AutorizarAccesoUsuarios"
36.   when
37.     obj:AutorizarAccesoUsuarios(state==Objetivo.PENDING)
38.     Focus(foco ==obj)
39.   then
40.     obj.setSolving();
41.     update(obj);
42. end
```

La regla expresa lo siguiente:

Si existe (en la memoria del agente)
un objetivo obj cuyo atributo state es PENDING y
un objeto de la clase Focus cuyo atributo foco tiene como valor el objeto obj
entonces
ejecutar el método setSolving de obj para cambiar su estado a SOLVING y a continuación actualizar el objeto en la memoria de trabajo.

d) reglas para intentar la realización del objetivo ejecutando una tarea. La tarea debe generar -directa o indirectamente- información para realizar el objetivo.

Las ejecución de tareas para intentar realizar un objetivo tienen como como pre-condición:

- **que exista la intención de realizar el objetivo** : debe existir un objeto de la clase Focus y su atributo foco debe apuntar al objetivo
- **que el objetivo se encuentre en estado solving** :el atributo state del agente debe ser SOLVING.

Ejemplo.

```
43. rule "Solicitud de Datos Inicial"
44.    when
45.     obj:AutorizarAccesoUsuarios(state==Objetivo.SOLVING)
46.     Focus(foco == obj)
47.    then
48.     recursoTrazas.aceptaNuevaTraza(new InfoTraza(agentId,"Se ejecuta la tarea :
           SolicitarDatosAcceso",InfoTraza.NivelTraza.debug));
49.     TareaSincrona tarea = gestorTareas.crearTareaSincrona(SolicitarDatosAcceso.class);
50.     tarea.ejecutar(VocabularioSistemaAcceso.IdentRecursoVisualizacionAccesoInicial);
51. end
```

La interpretación de la regla es la siguiente:

Si *existe un objetivo* ***AutorizarAcceso*** *en estado solving y el objetivo esta focalizado (el agente ha decidido resolverlo)* ***entonces*** *ejecutar la tarea* ***SolicitarDatosAcceso,*** *pasando como parámetro el identificador del recurso de visualización de acceso definido en VocabularioSistemaAcceso.IdentRecursoVisualizacionAccesoInicial*

Esta tarea tiene como cometido (ver sección 1.4.2) pedirle al Recurso de Visualización que presente la ventana para que el usuario pueda introducir los datos de acceso. La implementación del método ejecutar es el siguiente:

```
public class SolicitarDatosAcceso extends Tarea {
    private String identAgenteOrdenante;
    private Objetivo contextoEjecucionTarea = null;
  @Override
  public void ejecutar(Object... params) {
      //    String   identRecursoVisualizacionAcceso = "VisualizacionAcceso1";
          String identDeEstaTarea=this.getIdentTarea();
          String identRecursoVisualizacionAcceso = (String)params[0];
     try {
         // Se busca la interfaz del visualizador en el repositorio de interfaces
          ItfUsoVisualizadorAcceso visualizadorAcceso = (ItfUsoVisualizadorAcceso)
                  NombresPredefinidos.REPOSITORIO_INTERFACES_OBJ.obtenerInterfaz(
                                          NombresPredefinidos.ITF_USO +
identRecursoVisualizacionAcceso);
           if (visualizadorAcceso!=null)
                  visualizadorAcceso.mostrarVisualizadorAcceso(this.getAgente().getIdentAgente(),
                          NombresPredefinidos.TIPO_COGNITIVO);
            else {
                 identAgenteOrdenante = this.getAgente().getIdentAgente();
                 this.generarInformeConCausaTerminacion(identDeEstaTarea, contextoEjecucionTarea,
                          identAgenteOrdenante, "Error-
AlObtener:Interfaz:"+identRecursoVisualizacionAcceso,
  CausaTerminacionTarea.ERROR);
                  }
     } catch(Exception e) {
                       this.generarInformeConCausaTerminacion(identDeEstaTarea, contextoEjecucionTarea,
identAgenteOrdenante, "Error-Acceso:Interfaz:"+identRecursoVisualizacionAcceso,
CausaTerminacionTarea.ERROR);
                  e.printStackTrace();
           }
  }
}
```

El agente ejecutará la tarea para poder recibir los datos de autenticación que introduzca el usuario, pero ni la petición de los datos, ni el envío de los datos aparecen explícitamente en el código de la ejecución de la tarea. Es el recurso de **VisualizacionAcceso** quien debe enviarlos por medio de la interfaz de agente.

El siguiente diagrama de la Figura 7 ilustra el flujo de información entre el agente el recurso y el usuario cuando se ejecuta la tarea.

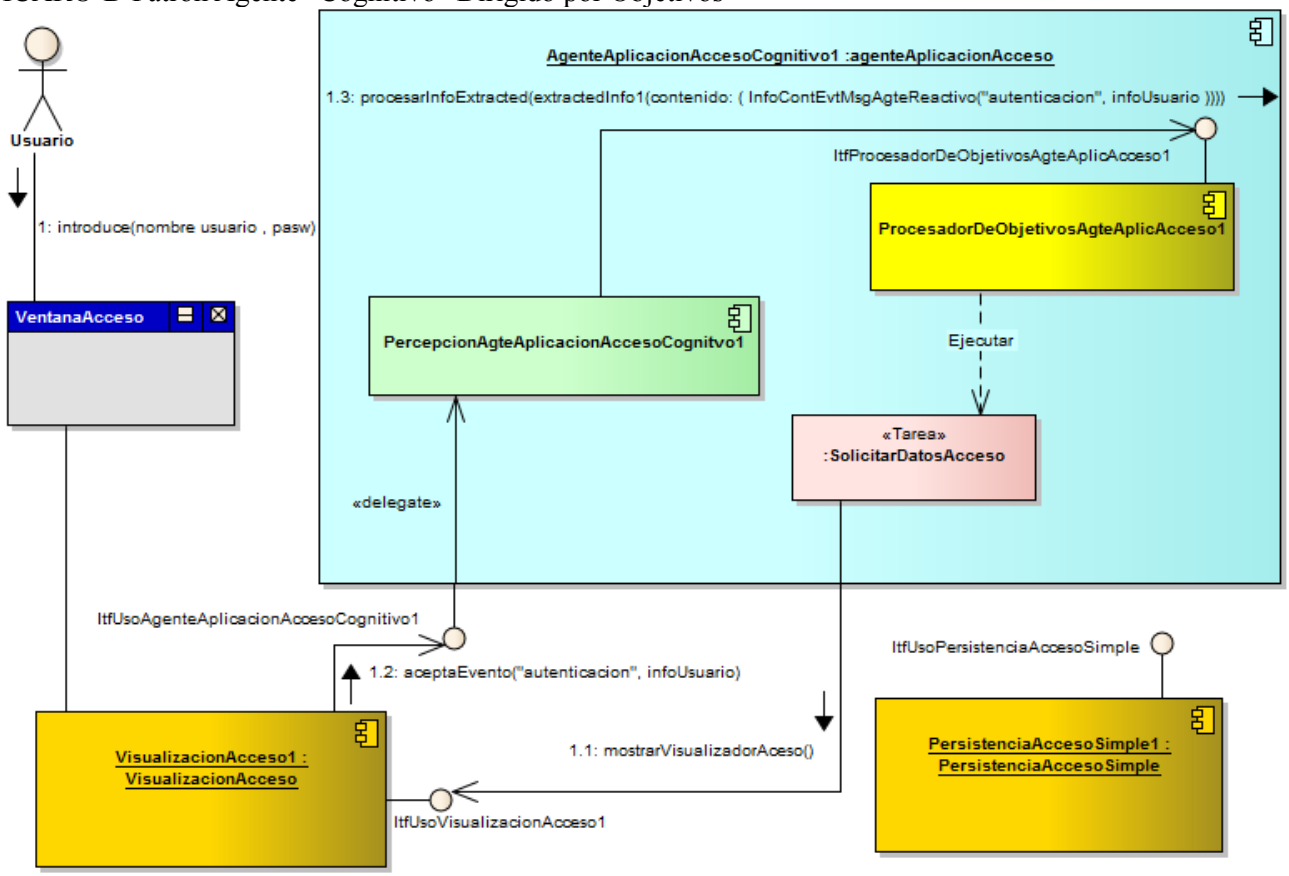


Figura 7: Flujo de información al ejecutar la tarea Solicitar datos de Acceso

El visualizador de Acceso: presenta la ventana de introducción de datos de acceso.

- Captura la información introducida por el usuario (nombre de usuario y clave de acceso)
- Crea un evento donde incluye la información
- Envía el evento al agente de acceso (mensaje 1.2 de la figura) .

El agente: recibe el evento a través de su interfaz de uso.

- La percepción es la encargada de procesar el ítem recibido que puede ser un evento o un mensaje.
  1. Encola el ítem
  2. El Procesador de ítems: extrae el ítem de la cola y puede descartarlo o considerarlo válido en cuyo caso puede extraer directamente su contenido o utilizar un **Interprete de eventos o mensajes que procesará el contenido, para alinearlo con el modelo de información del procesador de objetivos**. En ambos casos se genera un objeto de la clase **InfoExtracted** donde se indica la procedencia de la información, el creador de la misma y el contenido extraído. En el ejemplo se utiliza un intérprete de eventos.
  3. El **Interprete de eventos simples** procesa el evento recibido:
     - Extrae su contenido. El contenido es el contenido del evento enviado por el recurso: un objeto de la clase **InfoContEvtMsgAgteReactivo** que ha sido creado por el recurso de visualización.
     - Genera un objeto de la clase **InfoExtracted** y le asocia como contenido la información extraída del evento.
     - El **interprete de eventos simples puede ser extendido** con clases específicas asociadas al comportamiento de un agente. Estas clases pueden incorporar nuevos métodos para transformar la información contenida en los eventos en nuevos objetos relacionados con la semántica del dominio o con el contexto de funcionamiento del agente (por ejemplo en performativas FIPA)

4. El objeto obtenido se añade a la cola donde se almacena la información extraída (cola InfoExtrated). Esto permite procesar los elementos almacenados de forma independiente a la recepción y al procesamiento de los eventos y/o mensajes recibidos.
5. Los objetos almacenados en la cola de información extraída, se procesan enviándolos al Procesador de Objetivos .
6. El Procesador de Objetivos:
   - Recibe un objeto con la información extraída de un evento o mensaje
   - Examina la información extraída y puede descartar la información recibida, realizar transformaciones o enviar el objeto al motor. En el ejemplo la decisión consiste en introducir en el motor de reglas:
     - La información contenida en el objeto InfoExtrated
     - El propio objeto InfoExtrated

La Figura 8 ilustra este proceso.

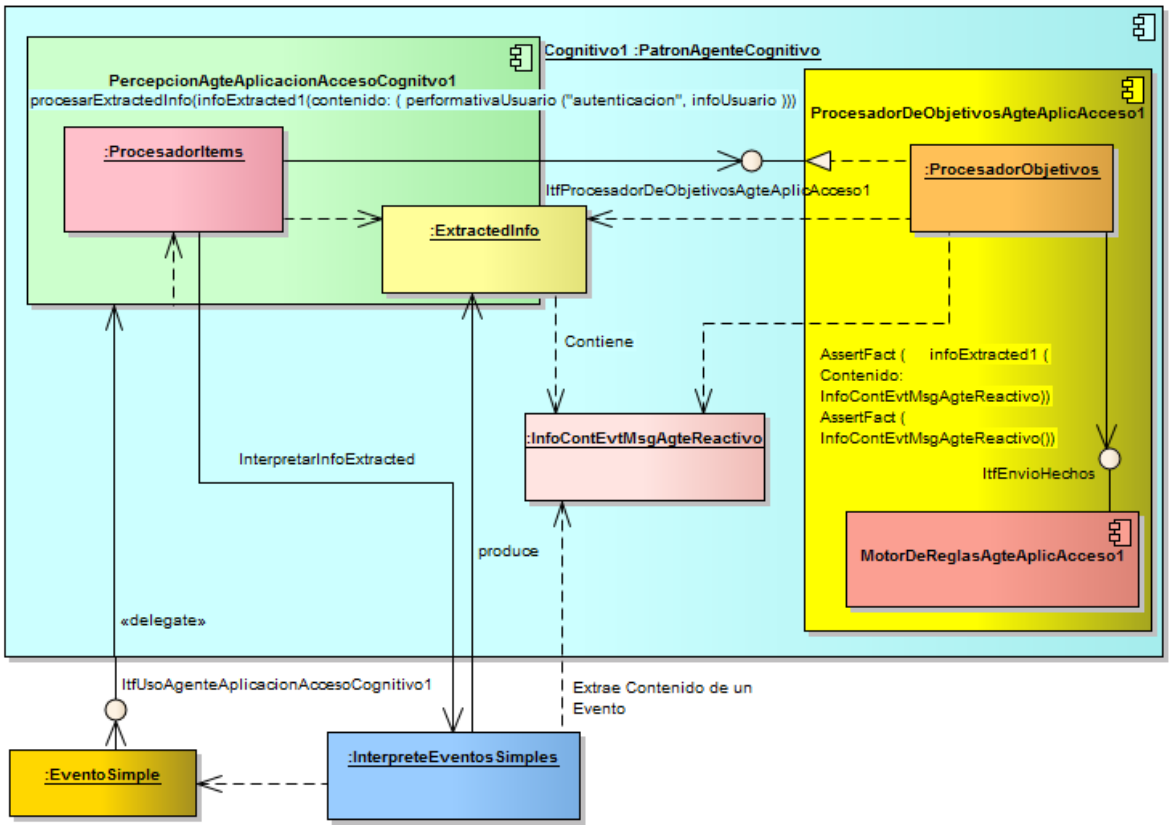


Figura 8: Procesamiento de Eventos

El tratamiento de la información extraída del evento recibido por el agente se realiza en la regla:

```
52. rule "Validacion de datos Iniciales de Acceso"
53.   when
54.     obj:AutorizarAccesoUsuarios(state == Objetivo.SOLVING)
55.     Focus(foco==obj)
56.     infoContEvt:InfoContEvtMsgAgteReactivo(msg soundslike "autenticacion" )
57.   then
58.     recursoTrazas.aceptaNuevaTraza(new InfoTraza(agentId,"Se ejecuta la tarea :
        ValidarDatosAutenticacionUsuarioIT",InfoTraza.NivelTraza.debug));
59.     Tarea tarea = gestorTareas.crearTarea(ValidarDatosAutenticacionUsuarioIT.class);
      60.     tarea.ejecutar(infoContEvt);
      61.     retract(infoContEvt);
62. end
```

La interpretación de la regla es la siguiente:

```
Si existe
un objetivo AutorizarAcceso con estado solving y el objetivo esta focalizado (
el agente ha decidido resolverlo ) y
```

```
(llega)un objeto InfoContEvtMsgAgteReactivo que contiene un atributo msg que contiene
la cadena  "autenticacion" y  la información de acceso introducida por el usuario
 entonces
Ejecutar la tarea  ValidarDatosdePerformativaUsuario pasando como parámetro el
objeto donde se encuentra la información introducida por el usuario.
Eliminar el  objeto de la memoria de trabajo
```

La eliminación del objeto de la memoria de trabajo esta motivada por el funcionamiento del motor de reglas que *sólo gira* (inicia un nuevo ciclo de verificación de reglas y de ejecución de acciones) , cuando recibe nueva información, por tanto para validar una nueva información enviada por el recurso de visualización se debe eliminar la antigua.

La ejecución de la tarea ValidarDatosdePerformativaUsuario realiza las siguientes acciones:

1. Obtiene la interfaz del Recurso de Persistencia
2. Utiliza el recurso para comprobar los datos introducidos por usuario
3. Con el resultado de la validación genera un informe de tarea con el resultado de la comprobación
4. Envía el informe al procesador de objetivos para que se utilice en el proceso de consecución de los objetivos en curso de consecución.
5. Si se producen excepciones al obtener la interfaz del recurso o al acceder a él, la tarea genera un informe indicando que ha terminado de forma errónea.

La implementación del método ejecutar de la tarea es el siguiente:

```
public class ValidarDatosAutenticacionUsuarioIT extends Tarea {
    private String identAgenteOrdenante;
    private Objetivo contextoEjecucionTarea = null;
@Override
public void ejecutar(Object... params) {
   String IdentRecursoVisualizacionAcceso
=VocabularioSistemaAcceso.IdentRecursoVisualizacionAccesoInicial;
   String   IdentRecursoPersistencia =VocabularioSistemaAcceso.IdentRecursoPersistenciaAcceso ;
//        Se extraen  los datos de los parametros
  InfoContEvtMsgAgteReactivo infoUsuario = (InfoContEvtMsgAgteReactivo) params[0];
        Object [] parametrosAccion = (Object [])infoUsuario.getParametrosAccion ();
        InfoAccesoSinValidar infoAcceso = (InfoAccesoSinValidar) parametrosAccion[0] ;
        try {
// Se obtienen las interfaces de los recursos. Si no se pueden obtener las interfaces se debe generar un
informe de tarea
        String identTarea = this.getIdentTarea();
        identAgenteOrdenante = this.getIdentAgente();
          ItfUsoVisualizadorAcceso visualizadorAcceso = (ItfUsoVisualizadorAcceso)

  NombresPredefinidos.REPOSITORIO_INTERFACES_OBJ.obtenerInterfaz(NombresPredefinidos.ITF_USO +
  IdentRecursoVisualizacionAcceso);
         if (visualizadorAcceso==null)
            this.generarInformeConCausaTerminacion(identTarea, contextoEjecucionTarea,
identAgenteOrdenante,                "Error-Al Obtener la interfaz de u so
de :"+IdentRecursoVisualizacionAcceso, C                                ausaTerminacionTarea.ERROR);
         else visualizadorAcceso.mostrarVisualizadorAcceso
                                    (identAgenteOrdenante,NombresPredefinidos.TIPO_COGNITIVO);
        ItfUsoPersistenciaAccesoSimple itfUsoPersistencia = (ItfUsoPersistenciaAccesoSimple)

  NombresPredefinidos.REPOSITORIO_INTERFACES_OBJ.obtenerInterfaz(NombresPredefinidos.ITF_USO +
  IdentRecursoPersistencia);
        if (itfUsoPersistencia == null) this.generarInformeConCausaTerminacion(identTarea,
                  contextoEjecucionTarea, identAgenteOrdenante, "Error-
                          AlObtener:Interfaz_"+IdentRecursoPersistencia, CausaTerminacionTarea.ERROR);
        else {
              boolean resultadoValidacion = itfUsoPersistencia.compruebaUsuario(infoAcceso.tomaUsuario()
                                          , infoAcceso.tomaPassword());
              String contenidoInformeTarea;
              if (resultadoValidacion){
                        contenidoInformeTarea
=VocabularioSistemaAcceso.ResultadoAutenticacion_DatosValidos;
              }else{
                    contenidoInformeTarea =
VocabularioSistemaAcceso.ResultadoAutenticacion_DatosNoValidos;
                  }

this.generarInformeOK(identTarea,contextoEjecucionTarea,identAgenteOrdenante,contenidoInformeTarea);
              }
       } catch (Exception e) {
              e.printStackTrace();
              this.generarInformeConCausaTerminacion(this.getIdentTarea(), contextoEjecucionTarea,

  identAgenteOrdenante,"ErrorAlObtener:Interfaz_Recurso_Persistencia",CausaTerminacionTarea.ERROR);
```

```
            }
        }
    }
```

Las dos reglas siguientes especifican qué hacer cuando el procesador de objetivos reciba los resultados de la validación.

```
63. rule "Permiso del Acceso con Informe Tarea"
64.  when
65.   obj:AutorizarAccesoUsuarios(state == Objetivo.SOLVING)
66.   Focus(foco== obj)
67.   informeTarea:InformeDeTarea(identTarea == "ValidarDatosAutenticacionUsuarioIT", contenidoInforme ==
    "usuarioValido")
68.  then
69.   Tarea tarea = gestorTareas.crearTarea(PermitirAcceso.class);
70.   tarea.ejecutar(VocabularioSistemaAcceso.IdentRecursoVisualizacionAccesoInicial);
71.   retract(informeTarea);
72. end
73.
74. rule "Denegacion del Acceso con Informe Tarea"
75.  when
76.   obj:AutorizarAccesoUsuarios(state == Objetivo.SOLVING)
77.   Focus(foco==obj)
78.   informeTarea:InformeDeTarea(identTarea == "ValidarDatosAutenticacionUsuarioIT", contenidoInforme ==
    "usuarioNoValido")
79.  then
80.     Tarea tarea = gestorTareas.crearTarea(DenegarAcceso.class);
81.     recursoTrazas.aceptaNuevaTraza(new InfoTraza(agentId,"Resolviendo el objetivo : "+obj.getgoalId()
    +"  Ejecutando la tarea : "+ tarea.getIdentTarea() ,InfoTraza.NivelTraza.debug));
82.     tarea.ejecutar();
83.     retract(informeTarea);
84. end
```

En ambas reglas se utilizan los resultados producidos por la tarea : ValidarDatosAutenticacionUsuarioIT para ejecutar las tareas: PermitirAcceso o DenegarAcceso.

La tarea PermitirAcceso realiza las siguientes acciones:

- Utiliza el recurso de visualización para informar al usuario de que sus datos son válidos
- Genera un informe que indica al procesador de objetivos que el usuario ha sido informado. El tratamiento de este informe puede permitir resolver el objetivo.
- En el caso de que no se puedan obtener las interfaces del recurso o que se produzcan excepciones, se genera un informe de tarea indicando terminación errónea.

La tarea DenegarAcceso:

- Utiliza el recurso de visualización para informar al usuario de que sus datos NO son válidos.
- En el caso de que no se puedan obtener las interfaces del recurso o que se produzcan excepciones, se genera un informe de tarea indicando terminación errónea.

La implementación de los métodos ejecutar de las tareas es el siguiente:

```
public class PermitirAcceso extends Tarea {
        private String identAgenteOrdenante;
        private Objetivo contextoEjecucionTarea = null;
  @Override
  public void ejecutar(Object... params) {
            String identDeEstaTarea=this.getIdentTarea();
            String identRecursoVisualizacionAcceso = (String)params[0];
  try {
       identAgenteOrdenante = this.getIdentAgente();
       ItfUsoVisualizadorAcceso visualizadorAcceso = (ItfUsoVisualizadorAcceso)

NombresPredefinidos.REPOSITORIO_INTERFACES_OBJ.obtenerInterfaz(NombresPredefinidos.ITF_USO +
          identRecursoVisualizacionAcceso);
       if (visualizadorAcceso==null) this.generarInformeConCausaTerminacion(identDeEstaTarea,
                  contextoEjecucionTarea,identAgenteOrdenante, "Error
                        AlObtener:Interfaz_Recurso:"+identRecursoVisualizacionAcceso,
CausaTerminacionTarea.ERROR);
        else {
              visualizadorAcceso.mostrarMensajeInformacion(identDeEstaTarea, "Acceso permitido. Termina
el                      servicio de Acceso ");
              visualizadorAcceso.cerrarVisualizadorAcceso();
              this.generarInformeOK(identDeEstaTarea, contextoEjecucionTarea, identAgenteOrdenante,
                       VocabularioSistemaAcceso.NotificacionAccesoAutorizado);
              }
      } catch (Exception e) {
               this.generarInformeConCausaTerminacion(identDeEstaTarea, contextoEjecucionTarea,
                identAgenteOrdenante, "Error
                    AlUtilizar:Interfaces_Recurso:"+identRecursoVisualizacionAcceso,
                        CausaTerminacionTarea.ERROR);
```

```
                    e.printStackTrace();
                }
    }

public class DenegarAcceso extends Tarea {
        private String identAgenteOrdenante;
        private Objetivo contextoEjecucionTarea = null;
        private String identRecursoVisualizacionAcceso
=VocabularioSistemaAcceso.IdentRecursoVisualizacionAccesoInicial;
  @Override
  public void ejecutar(Object... params) {
    String identDeEstaTarea=getClass().getSimpleName();
    try {
              identAgenteOrdenante = this.getAgente().getIdentAgente();
            ItfUsoVisualizadorAcceso visualizadorAcceso = (ItfUsoVisualizadorAcceso)

  NombresPredefinidos.REPOSITORIO_INTERFACES_OBJ.obtenerInterfaz(NombresPredefinidos.ITF_USO +
  identRecursoVisualizacionAcceso);
    if (visualizadorAcceso==null)this.generarInformeConCausaTerminacion(identDeEstaTarea,
      contextoEjecucionTarea,identAgenteOrdenante,"Error:AlObtener:Interfaz_Recurso:"+
      identRecursoVisualizacionAcceso, CausaTerminacionTarea.ERROR);
    else visualizadorAcceso.mostrarMensajeError("Acceso denegado", "Identificador de usuario o
Contraseña           incorrectas. Introduzcalas de nuevo");
  } catch (Exception e) {
            this.generarInformeConCausaTerminacion(identDeEstaTarea, contextoEjecucionTarea,
                    identAgenteOrdenante, "Error-AlObtener:Interfaces_Recursos",
CausaTerminacionTarea.ERROR);
                  e.printStackTrace();
            }
  }
}
```

e) reglas para definir las condiciones de consecución de un objetivo

Este tipo de reglas definen la información necesaria para que el objetivo se considere resuelto, (cambie su estado a solved).

Un el ejemplo de este tipo de reglas en el sistema de acceso es la siguiente.

```
85. rule " objetivo AutorizarAccesoUsuarios Resuelto"
86.   when
87.     obj:AutorizarAccesoUsuarios(state==Objetivo.SOLVING)
88.     Focus(foco ==obj)
89.     informeTarea:InformeDeTarea(identTarea == "PermitirAcceso", contenidoInforme ==
    "Autorizacion_Acceso_Notificado_Al_Usuario")
90.   then
91.     obj.setSolved();
92. recursoTrazas.aceptaNuevaTraza(new InfoTraza("AgenteAplicacionAcceso1","Se ha resuelto el objetivo :
    "+obj.getgoalId(),InfoTraza.NivelTraza.debug));
93.     update(obj);
94. end
```

En la regla se expresa que si el objetivo está focalizado y se está resolviendo, cuando llegue un informe de tarea indicando que se ha notificado al usuario la autorización de acceso, entonces se cambiará el estado del objetivo de pending a solved.

El conjunto de reglas definidas anteriormente definen el ciclo de vida del objetivo AutorizarAccesoUsuarios. Cada tipo de regla definido anteriormente expresa las condiciones para:

- Crear el objetivo
- Decidir resolverlo (focalización)
- Iniciar el proceso de consecución ejecutando tareas
- Determinar cuando se ha resuelto

Este proceso se puede definir de forma gráfica en UML utilizando diagramas de actividad

2.3.4 Definición del ciclo de vida de los objetivos mediante diagramas de actividad

El proceso de consecución del objetivo *AutorizarAccesoUsuarios* puede expresarse con un diagrama de actividad en UML Figura 9, donde las actividades son las tareas. En el diagrama se definen:

a) **los objetos necesarios para que se ejecuten las tareas** (condiciones de ejecución de las tareas)
b) el **orden de ejecución de las tareas** (flujo de trabajo o work-flow) **para conseguir el objetivo** (el objetivo pase de un estado pending a solved).

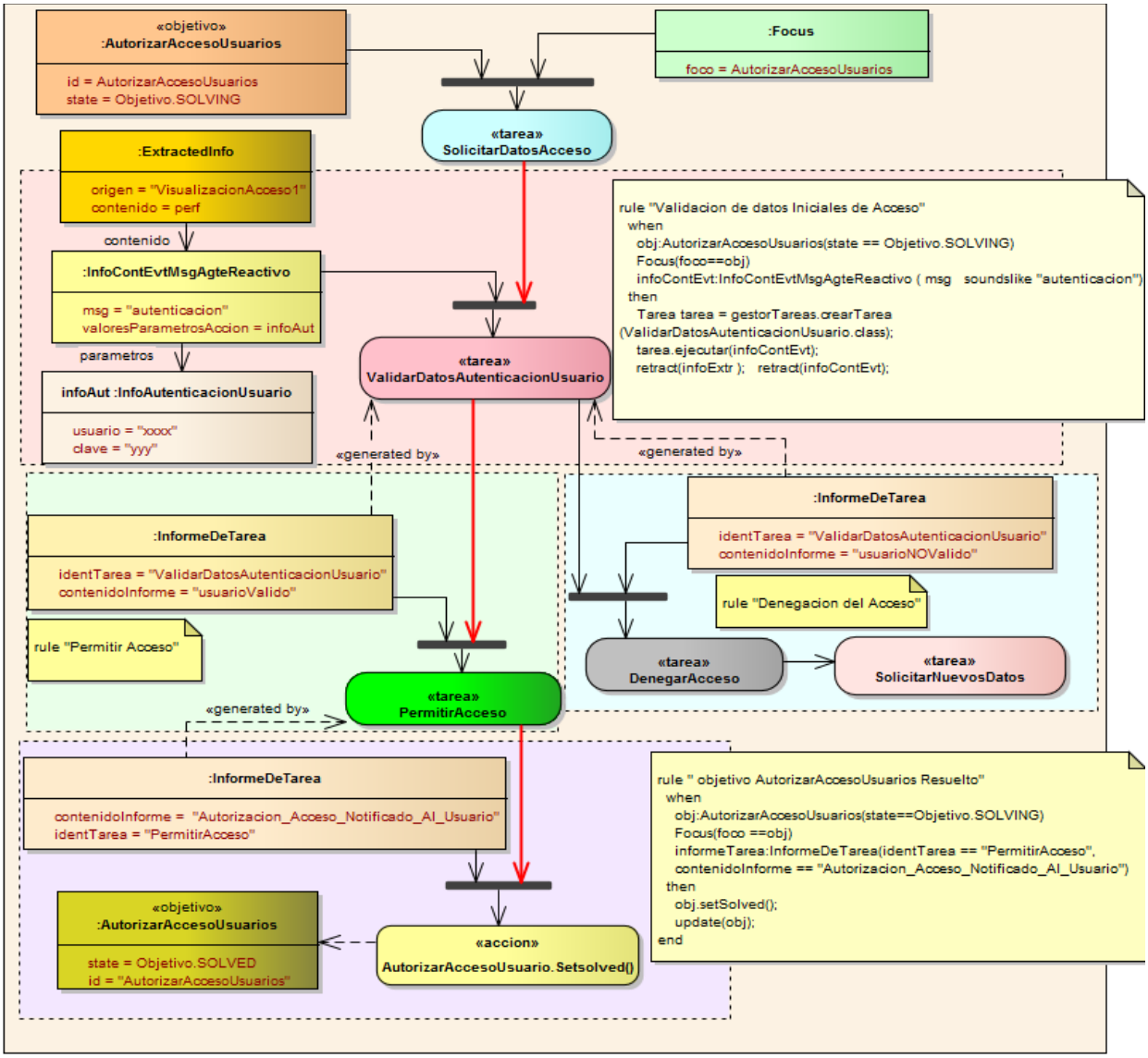


Figura 9: Ciclo de vida del objetivo AutorizarAccesoUsuarios

El la figura se expresa la equivalencia entre el formalismo gráfico del diagrama de actividad en UML (recuadros con líneas discontinuas) y el formalismo de las reglas en la parte derecha.

La tarea **SolicitarDatosAcceso** se ejecutara´ cuando exista : a) una instancia de objetivo con los atributos state=solving y su identificador sea AutorizarAccesoUsuarios y b) una instancia de la clase Focus cuyos atributo foco = objetivo AutorizarAccesoUsuarios. Como puede observarse esto es equivalente a lo expresado den la regla

```
rule "Solicitud de Datos Inicial"
   when
    obj:AutorizarAcceso(state==Objetivo.SOLVING)
    Focus(foco == obj)
   then
     Tarea tarea = gestorTareas.crearTarea(SolicitarDatosAcceso.class);
     tarea.ejecutar();
end
```

Toda tarea ***t*** que se ejecute para resolver un objetivo ***obj*** tiene como precondición:

- que exista el objetivo ***obj*** en estado solving: (obj: ObjetivoX(state == Objetivo.SOLVING))

- que el objetivo ***obj*** esté focalizado: Focus (foco==obj)

Por tanto para el resto de las tareas se definen únicamente los objetos necesarios para su ejecución. Ejemplo : La ejecución de la tarea ValidarDatosAcceso requiere:

a) una instancia de objetivo con los atributos state=solving y su identificador sea AutorizarAccesoUsuarios
b) una instancia de la clase Focus cuyos atributo foco = objetivo AutorizarAccesoUsuarios.
c) una instancia de creencia que contenga una performativaUsuario con los datos introducidos por el usuario

Esto es precisamente los que se expresa en la regla: rule *"Validación de datos Iniciales de Acceso"*

El diagrama de la Figura 9 proporciona una visión global del proceso de consecución del objetivo **AutorizarAccesoUsuarios.**

- Se definen las tareas que es necesario ejecutar para que el objetivo se considere realizado (o resuelto)
- Se definen las informaciones / objetos que debe recibir el procesador de objetivos (y el motor de reglas) para que el objetivo avance hacia su realización
- Se indica la entidad que genera las información (generated by)
- Se indica el orden de ejecución de las tareas (flujo de ejecución / work-flow , línea roja)
- A partir del diagrama pueden escribirse las reglas que serán interpretadas por el motor de reglas. Las reglas son en definitiva el formalismo que requiere el motor para **implementar cada paso** del ciclo de vida de un objetivo: generación, focalización y realización.

Es esencial **tener claro y bien definido el ciclo de vida completo de los objetivos**, porque en la mayoría de los casos las tareas deben ser ejecutadas ordenadamente. Por tanto es **altamente recomendable** definir explícitamente el flujo de ejecución de las tareas, así como los objetos necesarios para realizar cada paso del el proceso de consecución. Cómo se muestra en el ejemplo, este proceso se puede definir de forma clara y precisa con diagrama de actividad en UML. A partir de él pueden escribirse las reglas que interpretará el motor.

El diagrama de actividad puede considerarse un producto de diseño detallado, mientras que las reglas constituyen la implementación del proceso de consecución.

2.3.5 El modelo de información

El modelo de información esta formado por las clases que representan las entidades del dominio de aplicación. En el ejemplo hace falta una entidad para representar los datos de autenticación introducidos por el usuario. También es necesario representar el hecho de que los datos de autenticación sean válidos (cuando coincidan con los que están en la base de datos), o que no lo sean.

Se ha visto que los recursos generan eventos que envían a los agentes, los agentes deben extraer e interpretar el contenido, y/o para enviarlo a otros agentes por medio de mensajes.

En el ejemplo se utiliza las clases : ***InfoAccesoSinValidar y VocabularioSistemaAcceso.***

Pero puede observarse que en la carpeta: ***icaro.aplicaciones.informacion.dominioClases***. donde se encuentra esta clase, existen otras clases que han sido utilizadas en otras versiones del sistema de acceso.

Eventos, Mensajes y ExtractedInfo son contenedores de información. Los **contenidos** deben ser **objetos** correspondientes a **clases del dominio de aplicación**.

En el ejemplo el recurso de VisualizacionAcceso crea un evento donde introduce un objeto de la clase ***InfoAccesoSinValidar*** con el usn y pasw introducidos por el usuario y se lo manda al agenteAcceso. (Ver en la clase **NotificacionesEventosVisAcceso** el método *peticionAutentificacion,* las líneas siguientes.)

```
InfoAccesoSinValidar InfoAutenticacion = new
                InfoAccesoSinValidar(username,password);
enviarEventoaOtroAgente( new EventoRecAgte("autenticacion",
                            InfoAutenticacion,identificadordeEsteRecurso,

dentificadorAgenteaReportar),identificadorAgenteaReportar);
```

La clase ***VocabularioSistemaAcceso*** *centraliza las constantes utilizadas en la aplicación,* por ejemplo mensajes del sistema para el usuario, identificadores de agentes y de recursos, nombres de operaciones, mensajes de error, etc. La definición en una clase, simplifica la localización de las constante y la adaptación del sistema frente requisitos de interfaz de usuario , cambios idiomáticos e interacción entre componentes.

Compartir las clases de dominio permite que Agentes y Recursos colaboren para realizar los objetivos de la aplicación, es decir para implementar la funcionalidad del sistema .

2.3.6 Definición del comportamiento del agente en el fichero de descripción de la organización

El comportamiento del agente se define incluyendo en las carpetas objetivos y tareas las clases correspondientes a los objetivos y a las tareas de la aplicación y en la carpeta proceso de consecución de objetivos el fichero de reglas drools que expresa cómo generar los objetivos y como resolverlos.

Esto ficheros deben estar en las rutas:

icaro.aplicaciones.agentes.AgenteAplicacion<nombreAgente>.objetivos

icaro.aplicaciones.agentes.AgenteAplicacion<nombreAgente>.tareas

icaro.aplicaciones.agentes.AgenteAplicacion<nombreAgente>.procesoConsecucionObjetivos

La ruta donde se encuentra el fichero de reglas debe especificarse en el fichero de descripción de la organización que se encuentra en la ruta : (ver siguiente apartado)

config/icaro/aplicaciones/descripcionOrganizaciones/<nombreficheroDescripcioOrganizacio>.

El sistema valida la ruta donde se encuentra el fichero de reglas. No se creará el agente si la ruta no es la adecuada o si el nombrado del fichero de reglas no tiene el formato: **reglas**< *nombreAgente>.drl.*

Para completar la definición del modelo de agente es necesario especificar las clases de dominio que se pueden situarse

icaro.aplicaciones.informacion.dominioClases.aplicacionAcceso

o alternativamente en :

icaro.aplicaciones.agentes.AgenteAplicacion<nombreAgente>.informacion

Se recomienda estructurar los directorios de esta forma, pero no es obligatorio, excepto la ruta del fichero de reglas.

2.3.7 Ejecución/validación del sistema de acceso

Para ejecutar el sistema de acceso es necesario definir los agentes y los recursos que forman la organización mediante un fichero xml que se debe almacenar en la carpeta situada en la ruta:

config/icaro/aplicaciones/descripcionOrganizaciones

En la carpeta ***descripcionOrganizaciones*** *existen varias descripciones que pueden ser reutilizadas para definir la nueva organización. Lo más sencillo es modificar el fichero que describe el sistema de acceso* ***descripcionAplicacionAcceso.xml*** *basada en un agente reactivo. Se edita el fichero y se sustituye la definición del comportamiento del agente reactivo por un modelo dirigido por Objetivos (ADO) . Para ello:*

Se modifica la línea correspondiente al tipo de agente y a la localización del comportamiento

```
********* Descripción del comportamiento de los agentes de aplicación
*************************************************************************************************
-->
  <icaro:DescComportamientoAgentesAplicacion>
          <icaro:DescComportamientoAgente
                  nombreComportamiento="AgenteAplicacionAcceso" rol="AgenteAplicacion"
          tipo="ADO"
          localizacionComportamiento="icaro.aplicaciones.agentes.agenteAplicacionAccesoADO"
          localizacionFicheroReglas="icaro.aplicaciones.agentes.agenteAplicacionAccesoADO.
          procesoResolucionObjetivos.reglasAgenteAcceso.drl"/>
  </icaro:DescComportamientoAgentesAplicacion>
  </icaro:DescComportamientoAgentes>
```

Se puede tambien especificar opcionalmente la ruta donde se encuentra el fichero de reglas (linea siguiente en azul). La localización del comportamiento debe ser consistente con la localización del fichero de reglas. El fichero

de reglas debe situarse en la ruta de la localización del comportamiento en un directorio que se denomine procesoResolucionObjetivos. ***El nombre del fichero de reglas debe comenzar por "reglas".*** **Estas restricciones tienen como objetivos facilitar la estructuración de las aplicaciones y el nombrado de las entidades.**

Los cambios introducidos pueden salvarse en un nuevo fichero cuyo nombre será ahora : ***descripcionAplicacionAccesoADO.xml***

La ejecución de la aplicación se hace de la forma estándar: al main de Java se le pasa como parámetro el fichero de descripción de la organización que en este caso es : ***descripcionAplicacionAccesoADO***

Como resultado de la ejecución debe salir la ventana de trazas y la ventana de acceso donde el usuario puede introducir los datos de acceso Figura 10

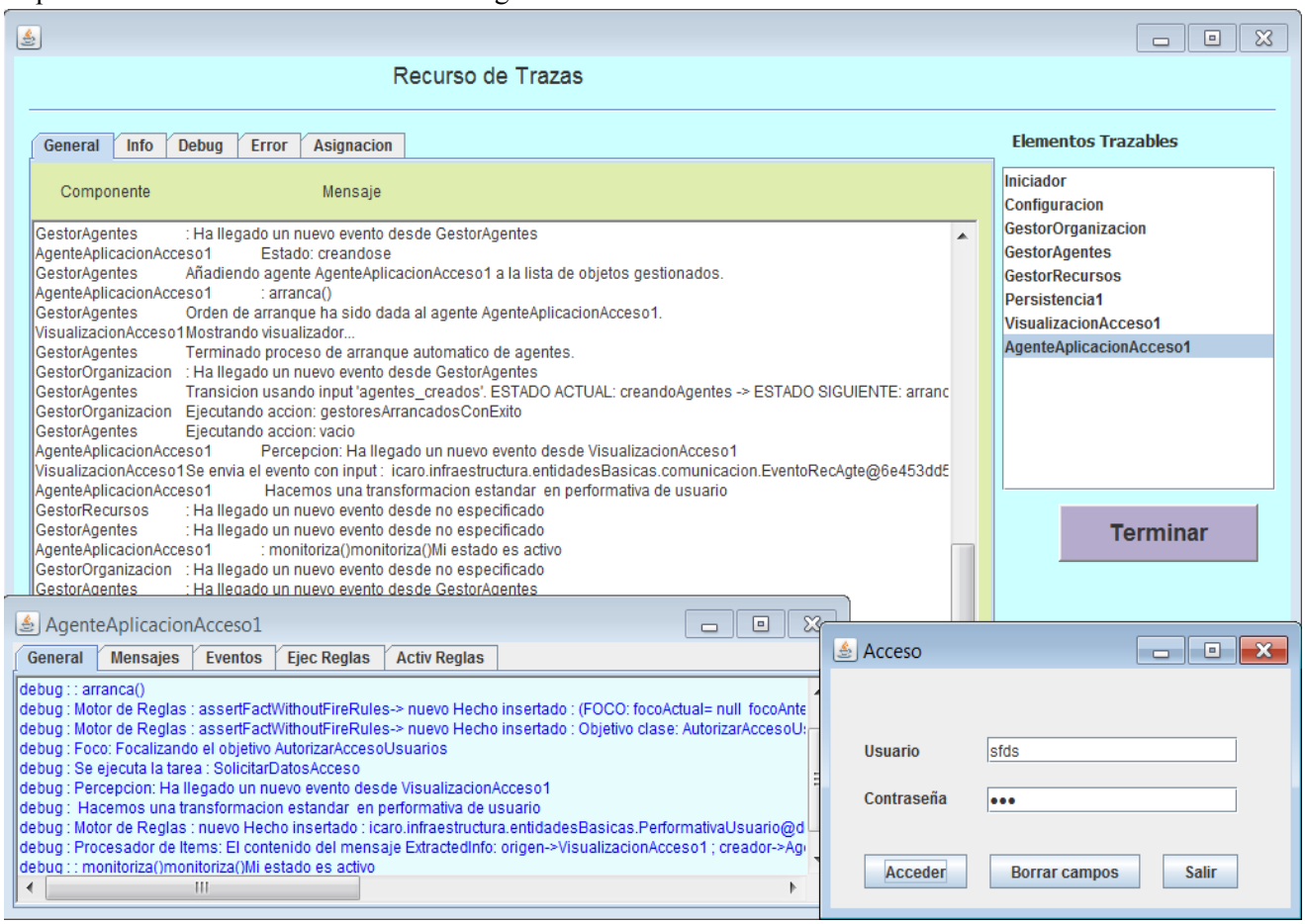


Figura 10: Ejecución sistema Acceso Cognitivo

La diferencia con el sistema de acceso implementado con un agente reactivo, puede verse en la ventana que traza el comportamiento del agente. En esta ventana **- AgenteAplicacionAcceso1 –** aparecen nuevas pestañas que permiten visualizar distintos tipos de trazas.

En la ventana de la pestaña General si visualizan las trazas que tengan como identificador en la traza el nombre del agente. En la ventana se muestran las siguientes trazas (en la parte izquierda de cada línea se indica el tipo de traza y en la parte derecha el mensaje de la traza :

- El agente recibe la orden de arranque (Esta orden la ha dado el gestor de agentes).
- El motor de reglas incorpora un nuevo hecho en la memoria de trabajo : el objeto FOCO
- El motor de reglas incorpora el Objetivo AutorizarAccesoUsuarios
- El Foco se cambia para que apunte al objetivo AutorizarAccesoUsuarios
- Se ejecuta la tarea SolicitarDatosAcceso. La tarea utiliza la interfaz del visualizador de acceso para ordenarle que visualice la ventana de acceso .

Cuando el usuario introduce su usuario y su contraseña:

El recurso envía un evento al agente de acceso con la información introducida.

- La información llega a la percepción que extrae el contenido del evento, transforma el contenido en una performativa de usuario y genera un objeto ExtractedInfo donde introduce la performativa de usuario con los datos de acceso.
- Cuando el motor recibe el objeto ExtractedInfo puede analizarse su contenido y transformarse o filtrarse de nuevo. Despues, se puede añadir al motor el objeto y su contenido o sólo su contenido. En este caso envía al motor sólo su contenido que es la performativa de usuario.
- El motor recibe la información y verifica que se satisfacen las condiciones de la regla: *"Validacion de datos Iniciales de Acceso"* por tanto se ejecutan las acciones de su parte derecha : la tarea *ValidarDatosAutenticacionUsuarioIT.class*.
- El resultado de la validación llega al motor mediante un informe de tarea
- La llegada del informe sirve para que se verifiquen las condiciones de la regla *"Denegacion del Acceso con Informe Tarea" que* ejecutará la tarea *Denegar acceso,* que ordenará al visualizador que saque un mensaje indicando que los datos no son válido.

En la ventana correspondiente a la pestaña Ejec Reglas se visualizan las reglas que se han utilizado en el proceso de consecución de los objetivos. Para ello debe utilizarse la operación de trazas aceptaTrazaEjecucionReglas (ej. *recursoTrazas.aceptaNuevaTrazaEjecReglas(agentId," EJECUTO LA REGLA: " + drools.getRule().getName())*;).

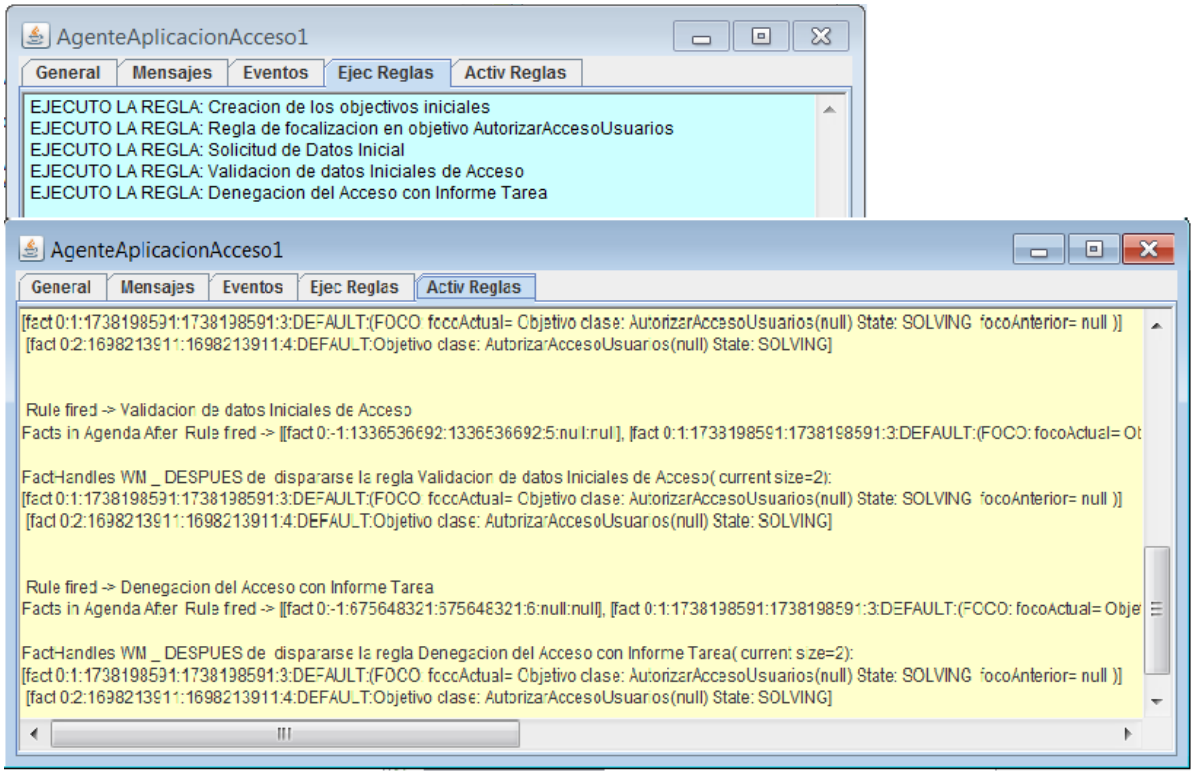


Figura 11: Ventanas de trazas de ejecución y de activación de reglas

En la ventana Activ Reglas puede obtenerse la justificación sobre la ejecución de las reglas generada por drools. En la traza se detallan las reglas que se aplican, los objetos que se han utilizado para validar las condiciones de la regla, y los objetos almacenados en la memoria de trabajo del motor antes y después de la ejecución de la regla.

En las pestaña de Eventos se muestran los eventos recibidos por el agente y en la de Mensajes los mensajes enviados y recibidos por el agente.

2.4 Extensión de la funcionalidad del agente de acceso: ejemplo acceso alta.

Una de las ventajas del modelo de agente dirigido por objetivos es su flexibilidad para realizar cambios en el comportamiento de los agentes, tanto para extender su funcionalidad como para modificarla o para limitarla. Objetivos, tareas y modelo de información, son elementos reutilizables. A la vista del comportamiento de la primera versión del sistema de acceso, se decide extender la funcionalidad con un nuevo requisito (informal) :

> *"Si el usuario realiza un acceso no válido, se le pedirán los datos de acceso –usn y pasw- para darle de alta en el sistema incluyéndolo en la base de datos".*

La siguiente figura ilustra los componentes del sistema y el intercambio de información para implementar el comportamiento requerido .

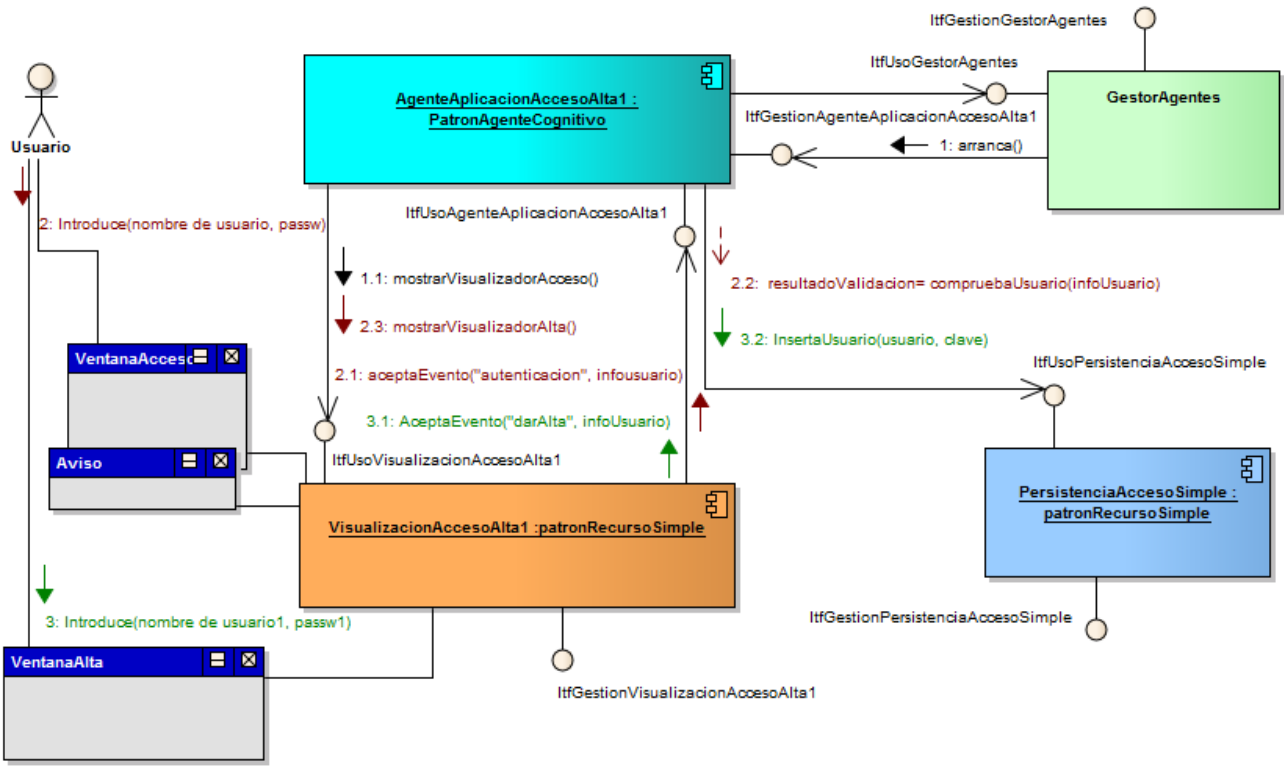


Figura 12: Diagrama de interacción con el comportamiento del sistema Acceso Alta

La arquitectura del nuevo sistema es básicamente la misma : un agente controlador y dos recursos uno de visualización y otro de persistencia. La implementación del nuevo requisito implica modificar la interfaz de usuario de forma que el usuario pueda introducir los datos para darle de alta. Esta funcionalidad se ha asignado a un nuevo recurso : **VisualizacionAccesoAlta** que implementará la funcionalidad del antiguo visualizador de acceso y la visualización de la nueva ventana para dar de alta al usuario. Este nuevo recurso :

- Ofrecerá una nueva operación en su interfaz de uso para que el agente pueda controlar la ventana de alta.
- Enviará al agente controlador la información introducida por el usuario en la ventana de alta por medio de un evento.

La implementación del recurso **visualizacionAccesoAlta** se encuentra en la ruta : *icaro/aplicaciones/recursos/visualizacionAccesoAlta.* Tiene la misma estructura que el recurso visualizacionacceso, pero se han nuevas clases para implementar las nuevas operaciones de su interfaz de uso (Figura ()) .

El nuevo agente **AgenteAplicacionAccesoAlta** se encargará de controlar la funcionalidad de la aplicación.

En el diagrama de comunicación de la figura () se detalla el intercambio de información entre componentes.

- La primera secuencia de mensajes etiquetada con 1, es la misma que en la versión anterior : El gestor de Agente arranca al agente **AgenteAplicacionAccesoAlta** :(1.Arranca()) , y este último le pide al recurso de visualización que muestre al usuario l ventana de acceso (1.1:mostrarVisualizadorAcceso).
- La secuencia de mensajes etiquetada con 2 ilustra la respuesta del sistema cuando el usuario introduce los datos de acceso (2.IntroduceNombre de usuario y pasw), los datos son enviados al agente (2.1 aceptaEvento (“autenticacion”, infoUsuario), el agente los envía al recurso de persistencia para su validación (2.2 resultadoValidacion=compruebaUsuario(infoUsuario)). Si los datos de acceso no son válidos el agente debe ordenarle al visualizador que presente al usuario la ventana de alta para recoger sus datos (2.3 mostrarVisualizadorAlta).
- La secuencia de mensajes etiquetada con 3 representa el tratamiento de los datos de alta introducidos por el usuario. Los datos los recoge el recurso de visualización y se los envía al agente por medio de un evento (3.1 AceptaEvento(“darAlta”, InfoUsuario). El agente recibe el evento, interpreta su contenido extrayendo la información de autenticación del usuario y se la envía al recurso de persistencia para su validación y posterior almacenamiento. (3.2 InsertaUsuario(Usuario, clave))

Para la implementación de este nuevo comportamiento basta con introducir un nuevo objetivo *DarAlta* que modelará **la obtención de los datos de alta** cuando el usuario no este registrado.

Este objetivo se sitúa en la ruta :

icaro/aplicaciones/ agentes/AgenteAplicacionAccesoADO/comportamientoAlta/objetivos

Para conseguir el nuevo objetivo se necesitan dos nuevas tareas :

- SolicitarDatosParaDardeAlta. Implementará la petición al recurso de visualización para que obtenga los datos de alta del usuario y los envíe al agente cuando los haya conseguido.
- ValidarDatosAlta. Utilizará el recurso de persistencia para validar los datos de alta definidos por el usuario y para almacenarlos con objeto de poder utilizarlos en nuevos accesos.

Las tareas del sistema de acceso que utilizan el recurso *visualizacionAcceso* ***deben ser modificadas*** *para que utilicen el nuevo recurso visualizacionAccesoAlta por ello se crearán dos nuevas tareas que incorporen estos cambios:*

- *SolicitarDatosAccesoAlta:practicamente igual a la tarea SolicitarDatosAcceso*
- *ValidarDatosAutenticacionUsuarioAlta: practicamente igual a la tarea ValidarDatosAutenticacionUsuario*

Las nuevas tareas se definen el la ruta :

icaro/aplicaciones/ agentes/AgenteAplicacionAccesoADO/comportamientoAlta/tareas

El comportamiento del agente es el siguiente:

Cuando recibe la orden de arranque genera los objetivos definidos en el sistema de acceso : *AutorizarAccesoUsuarios* y *TerminarServicioAcceso*

- Comienza la consecución del objetivo *AutorizarAccesoUsuarios* ordenando al visualizador presente las ventana de acceso para obtener los datos del usuario
- Valida los datos obtenidos, pero si no son válidos:
  - Genera un nuevo objetivo DarAlta para conseguir los datos de alta,
  - Abandona temporalmente el objetivo AutorizarAccesoUsuarios y se centra (focaliza) en conseguir el nuevo objetivo
- **Una vez conseguidos estos datos, los guarda y si no hay problema el objetivo** *DarAlta* **se puede considerar** *conseguido* **y continuar con la consecución del objetivo dar acceso.**

El ciclo de vida del nuevo objetivo *DarAlta* puede representarse con el diagrama de actividad de la Figura(). A partir del diagrama pueden obtenerse la reglas drools que implementarán el comportamiento del agente.

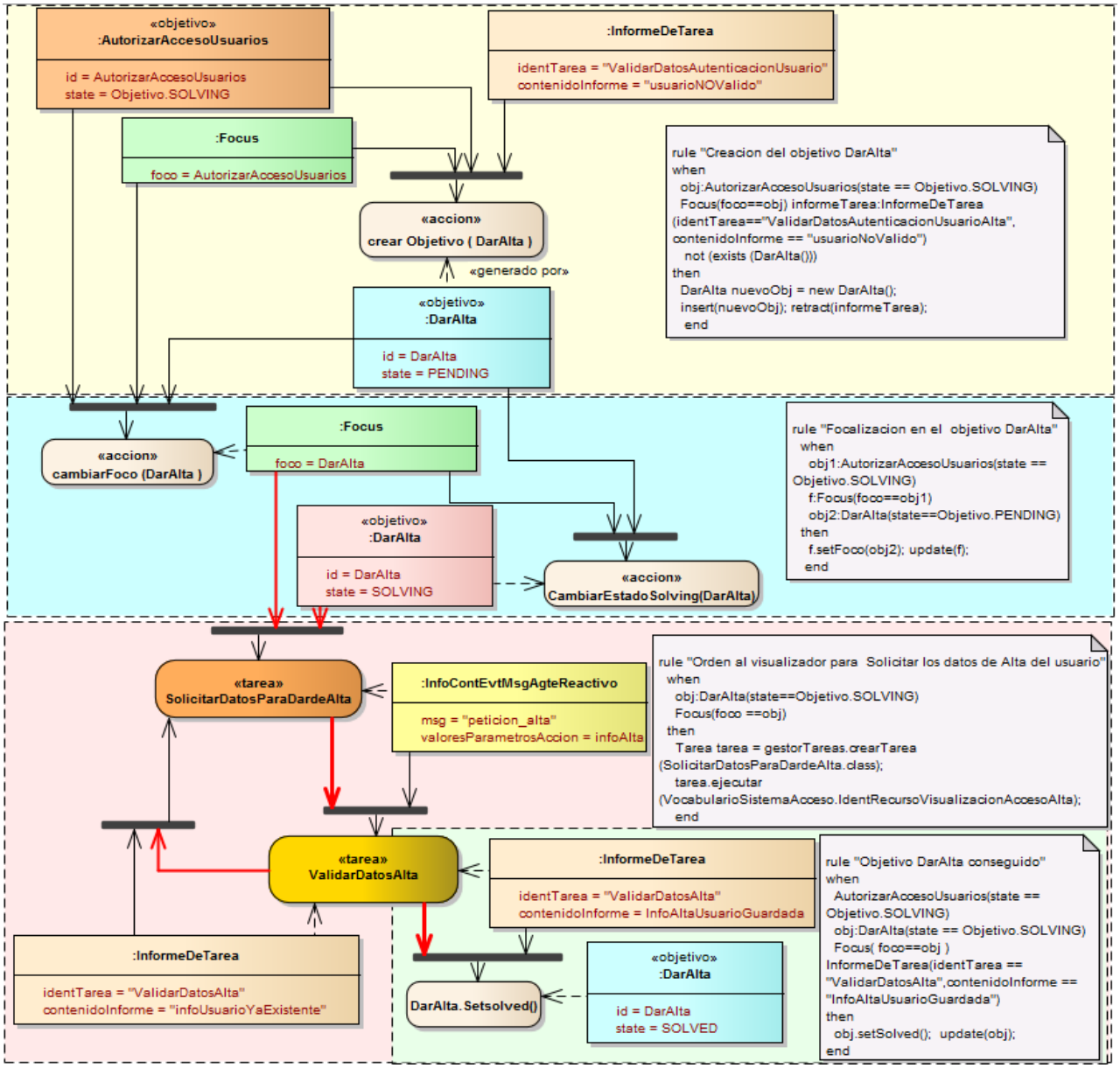


Figura 13: Diagrama de actividad del comportamiento del sistema Acceso Alta

En el diagrama se indican los identificadores de las reglas que se encuentran en el fichero : *reglasAgenteAccesoAlta.drl situado en la ruta icaro/aplicaciones/agentes/AgenteAplicacionAccesoADO/comportamientoAlta/procesoConsecucionObjetivos/**reglasAgenteAccesoAlta.drl***

El fichero de descripción de la nueva aplicación (organización) se encuentra en la ruta:

config/icaro/aplicaciones/descripcionOrganizaciones/sistemaAcceso/descripcionAplicacionAccesoSimpleADO.xml

El nuevo comportamiento del agente de acceso ha situado en la ruta:

icaro/aplicaciones/ agentes/AgenteAplicacionAccesoADO/comportamientoAlta/

donde se han situado los directorios *:*

***objetivos** : contiene el nuevo objetivo*

***procesoConsecucionObjetivos**:contiene el fichero de reglas*

***tareas**: contiene las nuevas tareas definidas*

En las reglas se utilizan tanto los objetivos del sistema de acceso inicial, y sus tareas como los objetivos y tareas nuevas.

En el desarrollo de esta nueva versión del sistema se han reutilizado:

- El recurso de persistencia
- Objetivos y tareas del sistema inicial
- Parte del proceso de consecución de los objetivos

Ha sido necesario extender:

- El recuro de visualización
- Definir un nuevo objetivo con su ciclo de vida
- Nuevas tareas para implementar la consecución del nuevo objetivo
- Una nueva descripción de la organización

2.5 Comparación de los modelos Reactivo y Dirigido por Objetivos

El ejemplo del sistema de acceso ilustra las facilidades que ofrece ICARO para reutilizar componentes y para realizar cambios sin que las modificaciones de componentes locales afecten al sistema completo. En particular:

- Se puede cambiar el modelo de un agente sin que esto afecte al resto de los componentes.
- El nuevo modelo de agente utiliza los recursos existentes, y el modelo de información común de la aplicación.
- El despliegue de los componentes, el arranque y la monitorización no varía.

ICARO facilita la elección del modelo de agente, y la evaluación experimental para determinar el que mejor se adapta a los requisitos de la aplicación.

Las implementaciones del comportamiento del agente de acceso realizadas con el patrón de agente reactivo y con el patrón dirigido por objetivos pueden servirnos para contestar las preguntas que surgen en un desarrollo ¿ Qué ventajas tiene el reactivo frente al cognitivo o viceversa ? ¿ En qué situaciones es más conveniente utilizar uno u otro ? ¿Existen criterios objetivos para seleccionar uno u otro ?

Como **ventajas del procesador de objetivos (PO)** frente a la máquina de estados finitos (MEF) pueden mencionarse:

- El PO proporciona conceptos computacionales de mayor nivel de abstracción
  - Objetivo, tarea, proceso de realización de un objetivo, son conceptos utilizados en el análisis de las aplicaciones. Estos conceptos pueden traducirse a entidades computacionales que se utilizan en la implementación .
  - El modelo reactivo esta basado en una máquina de estados. Los conceptos de alto nivel del análisis deben ser expresados / refinados en términos de estados y transiciones basadas en inputs y acciones.
- Facilita la reusabilidad
  - Objetivos y tareas son elementos reutilzables de un agente a otro y de un dominio a otro .
  - El concepto de estado es más restringido y menos reutilizable al estar ligado al modelo de control de una aplicación o a parte de ella.
- Mayor flexibilidad y expresividad para la ejecución de las tareas.
  - La ejecución de las acciones definidas en las transiciones de la máquina de estados dependen unicamente del estado actual y de los inputs recibidos.
  - Las condiciones de ejecución de las tareas se expresan mediante expresiones lógicas que pueden utilizar objetos existentes en la memoria de trabajo
- Facilidades de estructuración del modelo
  - Objetivos, tareas y reglas organizadas según el proceso de consecución de cada objetivo
  - Autómata difícilmente estructurable cuando el número de estados es muy grande
- Control de alto nivel

- Definido por las reglas de focalización que controlan la selección del objetivo a realizar
- Mecanismo inexistente en el modelo de agente reactivo

El concepto de objetivo proporciona mayor nivel de abstracción, permite hacer corresponder los objetivos identificados en el análisis con los objetivos computacionales que debe satisfacer el sistema

3 Nuevo ejemplo de uso del patrón : Formación de un equipo de agentes para la realización de tareas colectivas

Este ejemplo esta inspirado en el proyecto ROSACE [] . Un conjunto de robots están situados en un entorno de intervención para realizar diferentes tareas por ejemplo localización de posibles victimas y su rescate si fuera necesario. Un centro de control coordina la intervención de los recursos disponibles incluidos los robots desplegados en el entorno. Los robots son capaces de autoorganizarse, pueden formar equipos para realizar las tareas que encarga el centro de control y distribuirlas entre sus miembros de forma que se realicen de la forma más eficiente.

Con Icaro se puede simular el comportamiento de los robots de forma bastante realista considerando que son entidades físicamente distribuidas que tienen su propio ciclo de funcionamiento y que se comunican de forma asíncrona por medio de mensajes. Para trabajar en equipo los agentes deben disponer del concepto de “equipo” y utilizarlo para comunicarse y para cooperar con sus compañeros.

El modelo de organización del equipo es otro aspecto a considerar para definir la cooperación y la forma en que se consiguen los objetivos colectivos. Se utiliza el concepto de rol para definir posibles comportamientos de los agentes. El rol de un agente en un equipo define el tipo de objetivos que puede generar y conseguir así como la comunicación con agentes que tienen roles diferentes

3.1 Hipótesis de partida

Suponemos un conjunto de agentes/robots definidos en la organización inicial de ICARO. Para cada agente/robot se define el equipo al que pertenece. En el momento de su creación el agente/robot conoce el identificador de su equipo y su rol en el mismo pero desconoce los miembros del equipo y sus roles. Uno de los primeros objetivos para poder trabajar en el entorno de intervención es conocer a sus compañeros.

3.2 Definición del comportamiento de los agentes

Dado que los agentes van a trabajar en equipo el primer objetivo a conseguir una vez situados en el entorno sera “conocer los miembros de su equipo”. Este objetivo se modela con la clase **DefinirMiEquipo**. Una vez identificado el objetivo debemos especificar su ciclo de vida. Esto implica definir:

1. Las condiciones para se generación
2. Las condiciones para su consecución
3. Las acciones necesarias para que el objetivo se consiga

Se ha mencionado que el objetivo será lo primero que intenten conseguir los agentes definidos, por tanto la generación del objetivo e incluso su focalización puede incluirse en una tarea de inicialización de la memoria de trabajo de los agentes.

El objetivo se considerará conseguido (“solved”) cuando el agente tenga toda la información relativa a los miembros del equipo. Utilizaremos la clase **InfoEquipo** para modelar el concepto de equipo (Figura 14). Las variables o atributos de la clase indican los tipos de información que constituyen dicho concepto. Los agentes comienzan a funcionar inicializando su memoria de trabajo mediante la ejecución de una tarea “inicializarInfoWorkingMemory” : en esta tarea se crea un objeto de la clase **InfoEquipo** y se inicializan los atributos ***IndentEquipo*** con el identificador del equipo, **i*nicioContactoConEquipo*** = *false* , ***teamAgentIdsDefinidos***= *false*. El resto de los valores deben definirse durante el proceso de consecución del objetivo.

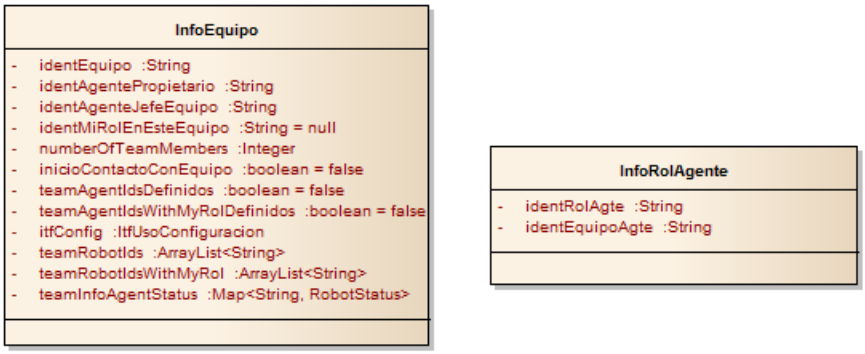


Figura **14**: Las **clase InfoEquipo e InfoRol**

Consideraremos que el objetivo se ha conseguido (pasará al estado *solved*) cuando el agente haya iniciado el contacto con los miembros del equipo y cuando haya conseguido sus identificadores. En términos computacionales se puede expresar de la forma siguiente:

> *cuando en la memoria de trabajo del agente el objeto que define la información del equipo (un objeto del a clase infoEquipo),los valores de los atributos* ***inicioContactoConEquipo*** *= true ,* ***teamAgentIdsDefinidos****= true . Entonces el objetivo podrá considerarse conseguido.*

Se podrían añadir condiciones adicionales que definirán la premisa de una regla de consecución de un objetivo por ejemplo que el número de robotIds definidos sea mayor que cero, o que sea un número determinado, pero eso también puede incluirse en el proceso para definir el valor true del atributo ***teamAgentIdsDefinidos.***

La cuestión ahora es definir las acciones y el proceso necesario para que los atributos tengan los valores que servirán para determinar que el objetivo se ha conseguido.

Los agentes pueden conseguir información ejecutando acciones propias, preguntando a otros agentes o utilizando las interfaces de los recursos de la organización. En el ejemplo, si un agente necesita conocer el equipo al que pertenece otro agente, parece que lo **más sencillo es preguntárselo,** si obtiene una respuesta afirmativa, actualiza su información acerca del equipo con el identificador del agente. Sin embargo el envío de preguntas por medio de mensajes asíncronos plantea como principal problema que la respuesta no llegue o que llegue cuando ya no se necesita. – esto suele ocurrir en la práctica diaria con el e-mail - . Las causas comunes a la falta de respuesta a un mensaje pueden ser :

- El mensaje no llega a su destinatario o el destinatario lo ignora.
- El destinatario recibe el mensaje y extrae la pregunta pero no responde, porque la pregunta llega en un momento inoportuno o porque no sepa o no quiera responder.

El agente que pregunta y que espera la respuesta para conseguir un objetivo, puede quedarse bloqueado si la respuesta no llega nunca. Una forma de evitarlo es **utilizar temporizadores** para indicar los tiempos de espera de la información; cuando pasa el tiempo definido, el agente puede: volver a enviar el mensaje e iniciar un nuevo temporizador, obtener la información por otros medios, intentar conseguir otros objetivos si los tiene, o esperar que llegue información exterior para generar objetivos y continuar funcionando.

Cuando un conjunto de agentes con el mismo objetivo – en el ejemplo conseguir información para conocer los miembros del equipo – intenta conseguirlo por medio de preguntas a todos los agentes, cada agente debe además de enviar las preguntas, procesar la respuestas, procesar la información de los temporizadores y reenviar las preguntas cuando las repuestas no llegan en el tiempo esperado.

Una forma de reducir la mensajería y el tiempo de proceso asociado a conseguir la información del equipo consiste en que los agentes "supongan" que hay otros posibles compañeros con el mismo objetivo y por tanto necesitarán conocer el equipo al que pertenece y su rol, por ello envían la información de su equipo y de su rol al resto de agentes sin que tengan que preguntarla y definen un intervalo de tiempo para procesar las informaciones que "suponen" que les van a enviar sus compañeros.

El diagrama de la Figura 15 describe un posible proceso para que los agentes de la organización consigan obtener la información sobre su equipo.

El proceso se inicia con la generación del objetivo **DefinirMiEquipo** nada más iniciarse la actividad del agente. Cada agente "*decide intentar conseguir"* para ello se cambia el atributo *state* del objetivo a *solving* y se focaliza para comenzar el proceso de consecución. Para ello:

Se envía la información particular sobre el equipo y el rol al resto de los agentes de la organización y se define un tiempo límite para obtener la información del equipo.

Se procesa la información acerca de su equipo procedente de otros agentes.

Cuando termina el tiempo fijado para la obtención de la información del equipo se comprueba que se ha recibido información de los agentes de la organización y se da por conseguido el objetivo.

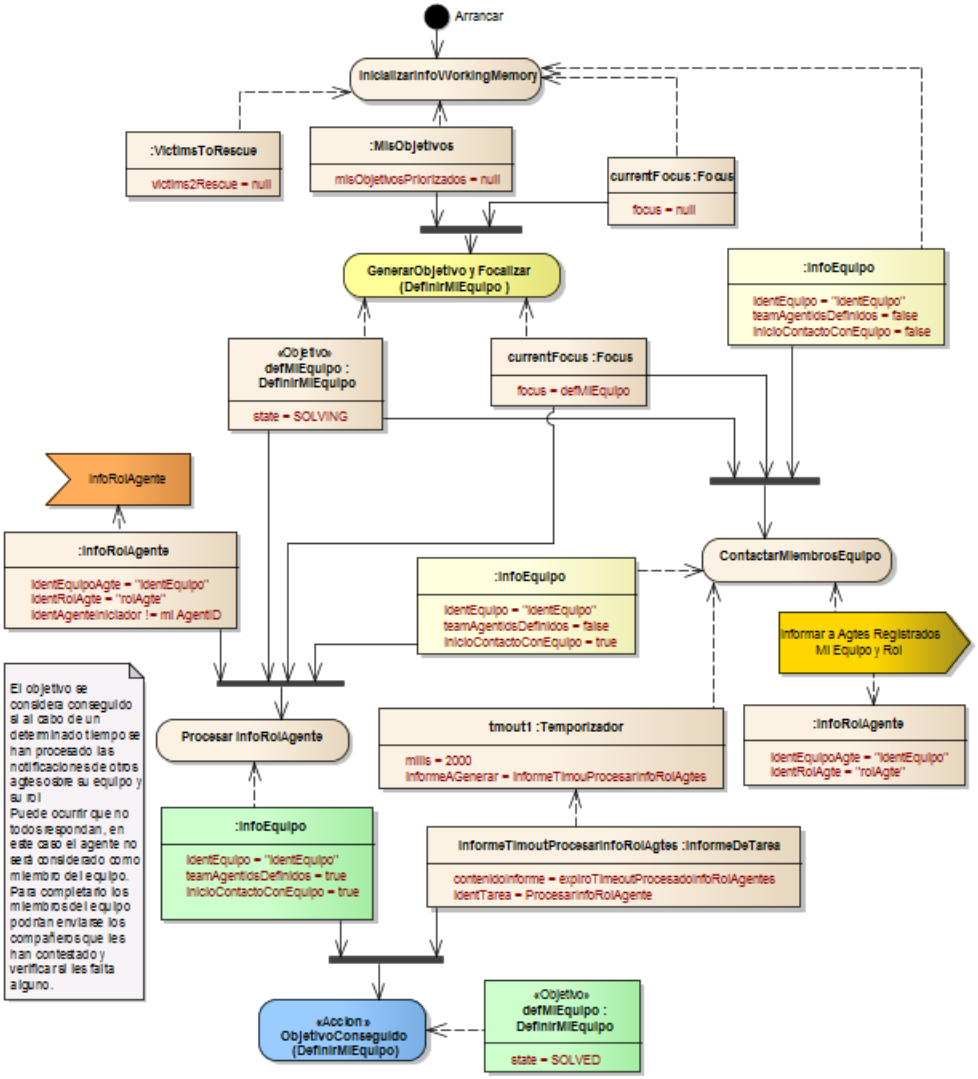


Figura 15: Proceso de consecución del objetivo DefinirMiEquipo

En el diagrama se utilizan las siguientes tareas:

- **InicializarInfoWorkingMemory** : crea los objetos iniciales (:InfoEquipo, :MisObjetivos, :Focus)
- **GenerarObjetivo y Focalizar** : crea un objeto de la clase DefinirMiEquipo y lo focaliza (el atributo focus del objeto creado toma como valor el objetivo creado .
- **ContactarMiembrosEquipo**: con esta tarea cada agente obtiene los agentes de la organización mediante el repositorio de interfaces y les envía un mensaje indicando el equipo y el rol del agente. La tarea inicia un temporizador donde se define el tiempo para obtener la información del equipo. Cuando el tiempo especificado haya pasado se genera un informe donde se indica la

tarea que generó el temporizador y un mensaje que indentifica la situación en que se produjo el final temporización

- **ProcesarInfoRolAgente:** procesa las informaciones recibidas de otros agentes indicando su equipo y su rol. El procesamiento consiste en actualizar los atributos de infoEquipo, por ejemplo, el número de miembros del equipo (numberOfTeamMembers), la lista de identificadores de los miembros del equipo (teamTobotsIds) y otros.

En el ejemplo se utilizan **restricciones temporales** que se implementan por medio de **temporizadores** y sirven para limitar el tiempo necesario para la consecución de determinada información o del proceso de consecución de un objetivo temporales. También puede utilizarse para evitar la inactividad de un agente “focalizado” en la consecución de un objetivo, o para cambiar de objetivo, o para continuar el proceso de consecución al cabo de un tiempo determinado.

Las tareas disponen de un método *generarInformeTemporizado* heredado de la clase **TareaSincrona** que permite crear un temporizador e iniciarlo definiendo la duración de la temporización y el mensaje que debe generarse cuando el tiempo haya transcurrido. En el diagrama de la figura la tarea : *contactarMiembrosEquipo* crea un temporizador con un tiempo de finalización de 2000 ms. Cuando los 2000 ms pasen, se generará un informe con dos atributos: *identTarea* que tiene como valor el identificador de la tarea que generó la temporización en este caso la tarea ProcesarInfoRolAgente; *informeDeTarea* contiene una cadena de caracteres “expiroTimeoutProcesarInfoRolAgtes” que indentifica la situación en que se produjo el final temporización.

El informe generado por el temporizador se envía al procesador de objetivos del agente que lo utilizará como una condición necesaria para dar por finalizado el proceso de consecución del objetivo.

Supongamos un grupo de cuatro agentes donde e ha definido individualmente el equipo y el rol al que pertenecen. Uno de los agentes tiene el rol “asignador de tareas” y el resto tienen el rol de “subordinado”.

Cuando se crean los agentes cada uno de ellos inicia el proceso definido en el diagrama de la Figura 15 Cada uno de ellos intenta en paralelo conseguir el mismo objetivo. En el diagrama de secuencia de la Figura 16 se ilustra el intercambio de mensajes entre los agentes. Se detallan las tareas y los objetos producidos en uno de los agentes cuando se recibe un objeto de la clase InfoRolAgente que contiene la información sobre equipo y rol de otro agente:

- El objeto recibido se le pasa como argumento a la tarea *ProcesarInfoRolAgente* que verifica si el agente pertenece a su equipo y si es así actualiza su lista de miembros.
- Se continua el procesamiento de otros objetos InfoRolAgente hasta recibir un informe generado por el temporizador indicando que se ha pasado el tiempo definido para la definición del equipo.
- Cuando se recibe en informe del temporizador se comprueba si se cumplen las condiciones para la consecución del objetivo.

El diagrama **sólo contempla el caso en que las condiciones para la consecución del objetivo se cumplen**, dado que cada agente tiene el mismo objetivo y que implementa el mismo proceso para conseguirlo, se podría concluir que al final todos tendrán un modelo de equipo idéntico y podrán iniciar proceso de cooperación, sin embargo si el tiempo especificado para conseguir el objetivo es demasiado pequeño o se producen problemas en el envío o en la recepción de la información sobre el equipo, es posible que todos los agentes consigan el objetivo **definirMiEquipo** pero su modelo de equipo sea diferente. Esta situación se deriva de las hipótesis de partida, aunque cada agente envía la información al resto no se ha verificado que la información haya sido recibida. Dado que cada agente conoce a quien ha enviado la información, cuando recibe el informe de temporización puede comprobar si alguno de los agentes no le ha enviado la información y pedírsela. El problema de nuevo es que a pesar de la pregunta la respuesta puede no llegar porque a) el mensaje no alcanza su destino; b) el agente ignora la pregunta; c) envía la respuesta pero no llega.

Se deja como ejercicio extender el ejemplo para implementar posibles mejoras.

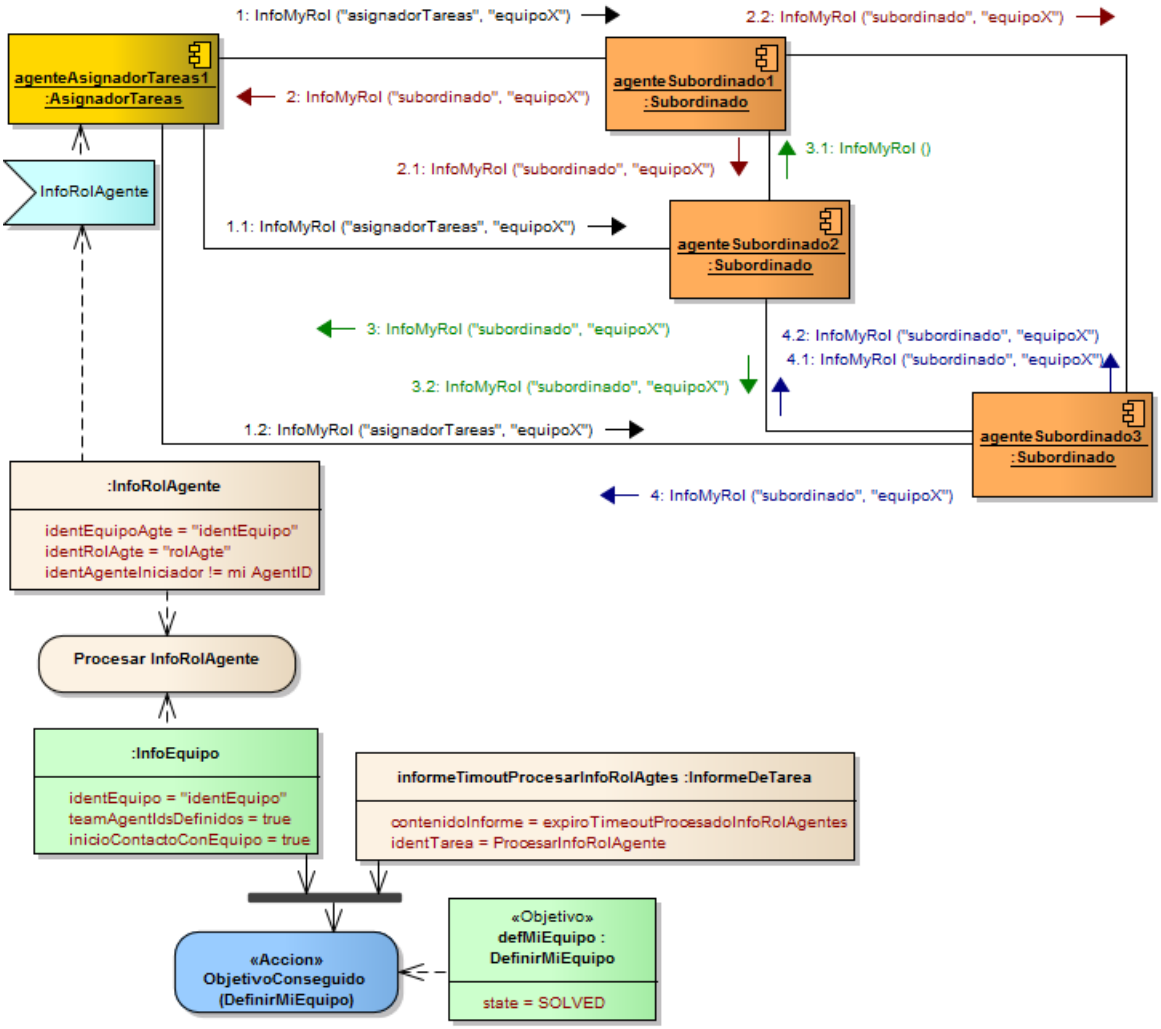


Figura 16: Mensajes y tareas en el proceso de consecución del objetivo DefinirMiEquipo

3.3 Detalles de la implementación

La implementación del ejemplo forma parte del material desarrollado en el proyecto ROSACE que esta accesible en https://github.com/fgarijo/rosaceSIM. La implementación del objetivo **definirMiEquipo** es común a todos los modelos de equipos de agentes. Detallaremos los aspectos más significativos concretamente parte de las reglas que implementan el proceso de consecución de los objetivos y alguna de las tareas. Tanto en el diagrama como en las reglas no se ha considerado la clase RobotStatus, por tanto los objetos de esta clase se han omitido.

Los diagramas de las Figura 15 y de la Figura 16 deben servir de base para la implementación con reglas.

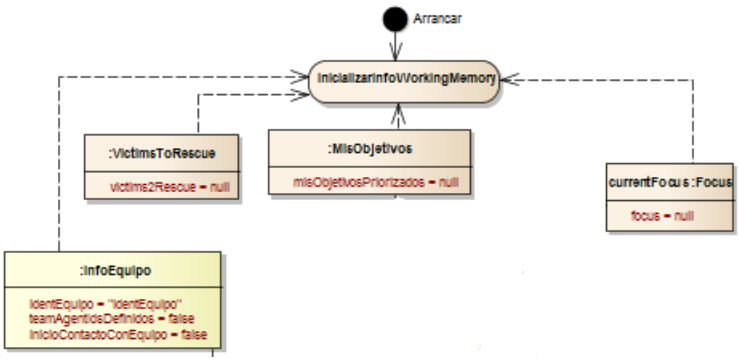


Figura 17: Inicialización de la memoria de trabajo del agente

Cada agente comienza inicializando su memoria de trabajo. El diagrama de la Figura 17 se implementa con la regla:

```
rule "Acciones Iniciales"
 when
    not ( exists(Focus()))
    not ( exists(MisObjetivos()))
 then
    TareaSincrona tarea1 =
gestorTareas.crearTareaSincrona(InicializarInfoWorkMem.class);
   tarea1.ejecutar( );
end
```

La aplicación de la regla tiene como consecuencia la ejecución de la tarea InicializarInfoWorkMem cuyo resultado es la creación los objetos de la figura y su inserción en la memoria de trabajo del agente.

El siguiente paso consiste en generar el objetivo : DefinirMiEquipo ponerlo a “solving” y focalizarlo para comenzar el proceso de consecución

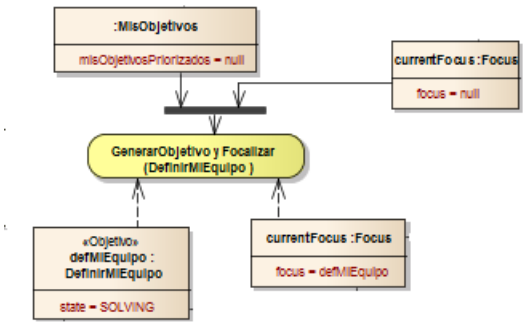


Figura 18: Generación y focalización del objetivo DefinirMiEquipo

Esta parte del diagrama se implementa con la regla

```
rule "Generacion Objetivo Conocer MiEquipo "
when
    focoActual : Focus(foco == null)
    misObjs: MisObjetivos()
 then
    DefinirMiEquipo definirMiequipoObj = new
DefinirMiEquipo(VocabularioRosace.IdentMisionEquipo);
    TareaSincrona tarea =
gestorTareas.crearTareaSincrona(GenerarObjetivoyFocalizarlo.class);
    tarea.ejecutar(definirMiequipoObj,misObjs,focoActual);
end
```

El proceso de consecución del objetivo comienza con el envío de la información del equipo y el rol del agente al resto de los agentes de la organización Figura 19.

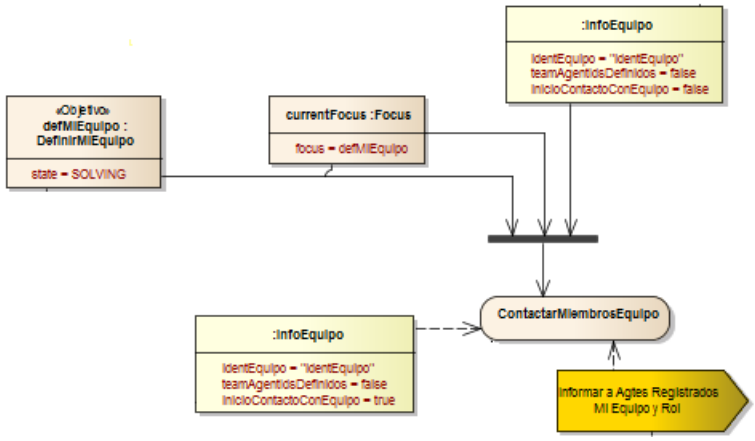


Figura 19: Contactar con miembros de la organización para informar del equipo y el rol

La tarea ContactarMiembrosEquipo se encarga de obtener los agentes de la organización y de enviarles la información. Para que se ejecute es necesario que en la memoria del trabajo del agente existan los objetos definidos en el diagrama con los valores de los atributos especificados.

La siguiente regla implementa el diagrama

```
rule "Inicio consecucion Objetivo Conocer MiEquipo "
when
    miEquipo: InfoEquipo(inicioContactoConEquipo== false)
    obj1 : DefinirMiEquipo(state == Objetivo.SOLVING)
    focoActual : Focus(foco == obj1)
then
    TareaSincrona tarea =
gestorTareas.crearTareaSincrona(ContactarMiembrosEquipo.class);
    tarea.ejecutar(miEquipo);
end
```

La implementación de la tarea **ContactarMiembrosEquipo** es la siguiente:

La tarea recibe como argumento un objeto de la clase **InfoEquipo** donde el valor del atributo inicioContactoConEquipo== false.

La implementación del método ejecutar de la tarea es el siguiente:

```
public void ejecutar(Object... params) {
try {
  InfoEquipo equipoInfo = (InfoEquipo)params[0];
// Se obtiene el identificador del agente que ejecuta la tarea
  nombreAgenteEmisor = this.getAgente().getIdentAgente();
// Se obtienen los identificadores de los posibles miembros del equipo
  agentesEquipo = equipoInfo.getTeamMemberIDs();
//Se crea un objeto con la información del equipo y el rol del agente para
//enviarla a los compañeros de equipo
  InfoRolAgente mirol = new InfoRolAgente(nombreAgenteEmisor,
  equipoInfo.getTeamId());
// se envia la información de equipo y rol a los posibles agentes del equipo
//utilizando la clase clase communicator accesible desde las tareas
  this.getComunicator().informaraGrupoAgentes(mirol, agentesEquipo);
 // Se actualiza la información del equipo anotando que se ha iniciado el
//contacto con el equipo , se pone true el atributo incioContactoConEquipo
// se actualiza el rol y el equipo del agente en la informacion del equipo
  equipoInfo.setinicioContactoConEquipo();
  equipoInfo.setidentMiRolEnEsteEquipo(miStatus.getIdRobotRol());
// Se envía la informacion del equipo actualizada  al motor
  this.getEnvioHechos().actualizarHechoWithoutFireRules(equipoInfo);
// Se genera el temporizador indicando el tiempo para conseguir la
// información que deben enviar el resto de compañeros
 this.generarInformeTemporizadoFromConfigProperty(
  VocabularioRosace.IdentTareaTimeOutContactarMiembrosEquipo,
  null,nombreAgenteEmisor, null);
        } catch (Exception e) {
              e.printStackTrace();
        }
   }
```

Una vez que se ha enviado la información InfoRolAgente al resto de agentes se pueden procesar cuando lleguen, las informaciones recibidas de otros agentes , según se especifica en el diagrama:

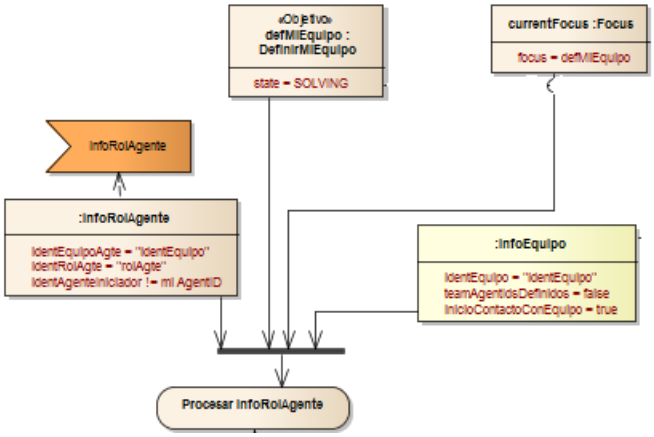


Figura 20: Procesamiento de las informaciones de equipo y rol enviadas por otros agentes

La información del diagrama se implementa con la siguiente regla

```
rule "Proceso InfoRol para conseguir Objetivo Conocer MiEquipo "
when
    miEquipo: InfoEquipo(idTeam:teamId,inicioContactoConEquipo== true)
    infoRolRecibido: InfoRolAgente(identEquipoAgte == idTeam)
    obj1 : DefinirMiEquipo(state == Objetivo.SOLVING)
    focoActual : Focus(foco == obj1)
then
    TareaSincrona tarea =
gestorTareas.crearTareaSincrona(ProcesarInfoRolAgente.class);
    tarea.ejecutar(miEquipo,infoRolRecibido);
end
```

La implementación de la tarea **ProcesarInfoRolAgente** es la siguiente:

La tarea recibe como argumentos un objeto de la clase **InfoEquipo** y otro objeto de la clase **InfoRolAgente** (el objeto enviado por alguno de los agentes del equipo) donde el valor del atributo

La implementación del método ejecutar de la tarea es como sigue:

```
public void ejecutar(Object... params) {
   try {
   InfoEquipo  miEquipo = (InfoEquipo)params[0];
   InfoRolAgente infoRolRecibido = (InfoRolAgente)params[1];
// El procesamiento de  infoRolRecibido se delega a un método de la clase
// InfoEquipo
   miEquipo.procesarInfoRolAgente(infoRolRecibido);
 // Si el equipo es jerarquico y el Rol es de agente asignador de tareas entonces
 // se obtiene su identificador y se le pone como  jefe en el equipo
 if (infoRolRecibido.getidentRolAgte().equals
        VocabularioRosace.IdentRolAgteDistribuidorTareas))
        miEquipo.setidentAgenteJefeEquipo(infoRolRecibido.getAgteIniciadorId());
 //Se elimina la información recibida para que la regla pueda aplicarse de nuevo
        this.getEnvioHechos().eliminarHechoWithoutFireRules(infoRolRecibido);
        this.getEnvioHechos().actualizarHecho(miEquipo);
            }
        catch(Exception e) {
              e.printStackTrace();
        }
    }
```

La regla precedente se aplicará cada vez que se añada un objeto de la clase InfoRolAgente , ejecutándose la tarea ProcesarInfoRolAgente que incorpora la información obtenida del procesamiento en el objeto que contiene la información del equipo del agente.

Este proceso continua hasta el vencimiento del temporizador iniciado por la tarea **ContactarMiembrosEquipo,** entonces se genera el informe de tarea definido en la última parte del diagrama de la Figura 15 y definido nuevamente en la figura Figura 21

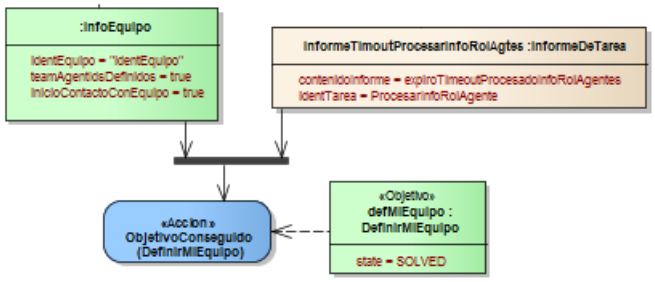


Figura 21: Consecución del objetivo DefinirMiEequipo

Esta parte del diagrama se implementa con la siguiente regla

```
rule "Proceso timeout para conseguir Objetivo Conocer MiEquipo "
when
    miEquipo: InfoEquipo(idTeam:teamId,inicioContactoConEquipo== true)
    misObjs: MisObjetivos()
    infoTarea: InformeDeTarea(identTarea ==
    VocabularioRosace.IdentTareaTimeOutContactarMiembrosEquipo)
    obj1 : DefinirMiEquipo(state == Objetivo.SOLVING)
    focoActual : Focus(foco == obj1)
then
// se da por conseguido el objetivo
    TareaSincrona tarea = gestorTareas.
                        crearTareaSincrona(ConseguirObjetivoActualizarFoco.class);
    tarea.ejecutar(misObjs,obj1,focoActual );
end
```

4 Utilización del patrón de Agente Dirigido por Objetivos para el Desarrollo de Aplicaciones

Este apartado esta dedicado a describir con más detalle la implementación del Patrón de Agente Dirigido por Objetivos así como los posibles problemas que se puedan encontrar en su utilización y las posibles soluciones. Se describe una nueva aplicación más típica de ingeniería de agentes donde se implementa un equipo de agentes que deben hacerse cargo de un conjunto de tareas especificadas por un operador en un escenario de crisis.

4.1 Estructura e interfaces del patrón Dirigido por Objetivos

En el apartado 1.2 se ha descrito brevemente el funcionamiento del patrón ADO. En esta sección se describirán con más detalle los componentes internos y las interfaces, subrayando aquellos aspectos que tienen mayor incidencia en el desarrollo de aplicaciones.

El comportamiento del patrón esta basado en la interacción entre dos componentes: la percepción y el procesador de objetivos.

La percepción tiene como cometido almacenar y procesar los mensajes y eventos recibidos a través de la interfaz de uso del agente. En la percepción se efectúa un primer tratamiento de la información que puede ser extendido según las necesidades de la aplicación. El tratamiento por defecto consiste en extraer el contenido de los mensajes y/o eventos almacenarlos en una clase denominada **ExtractedInfo** y enviarlos al procesador de objetivos para su tratamiento.

El Procesador de Objetivos controla el comportamiento global del agente por medio de la interpretación de las reglas que definen el ciclo de vida de los objetivos. La Figura 22 contiene la arquitectura del PO con las principales operaciones de las interfaces.

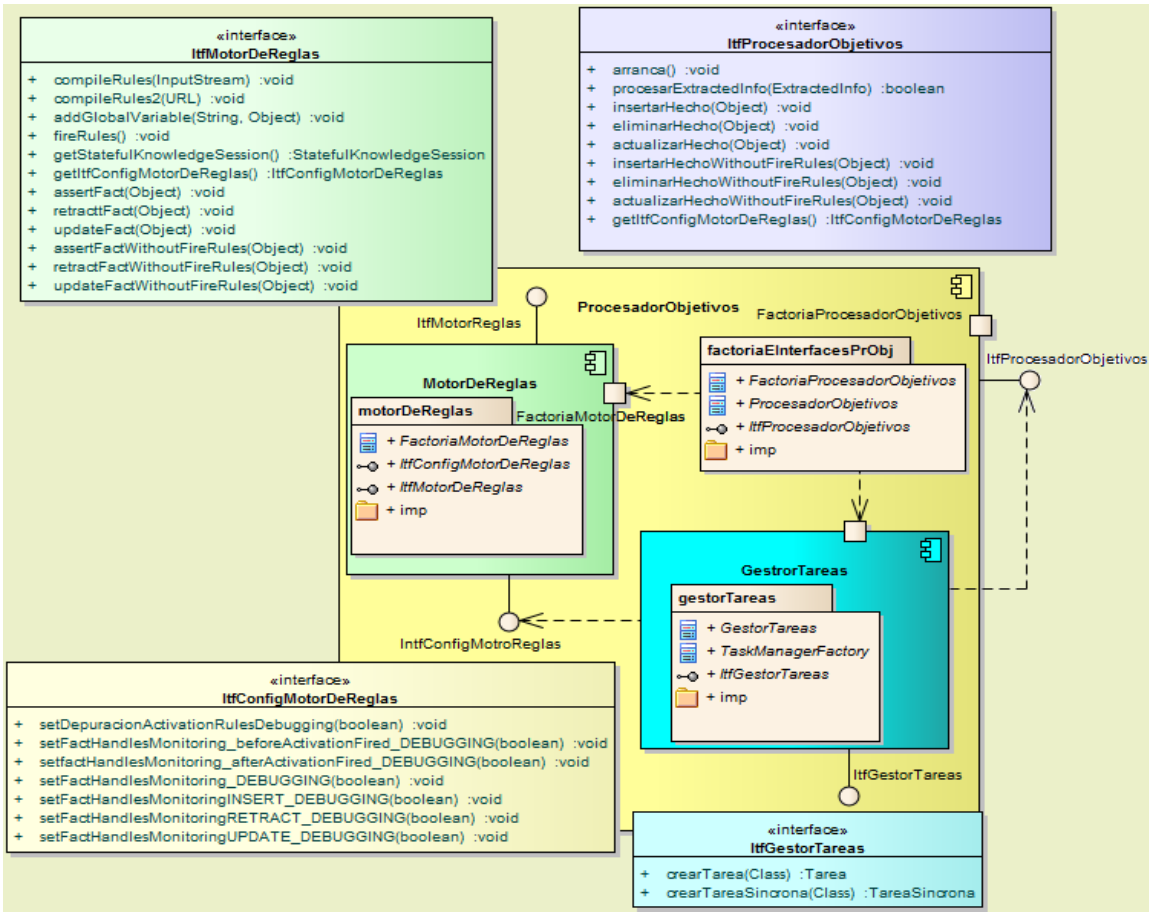


Figura 22: Arquitectura del Procesador de Objetivos

La interfaz del procesador de objetivos proporciona tres tipos de operaciones:

- Arranque del motor una vez creado por la factoría
- Obtención de la interfaz de configuración para poder definir las trazas que debe generar el motor
- Procesamiento de información: peticiones al motor de reglas para que añada, modifique o elimine objetos, y aplique las reglas cuyas condiciones se satisfagan.

Dado que el procesamiento de la información esta basado en un motor/interprete de reglas de producción, se hará un breve paréntesis para recordar el modelo computacional de este tipo de intérpretes y para destacar las particularidades de la implementación basada en Drools que son utilizadas en el patrón. En particular tanto la sintaxis de las reglas, como la validación y la interpretación en el contexto de la aplicación, los errores y las trazas generadas, los errores dependen de este componente.

4.1.1 Interfaces y operaciones del motor de reglas

El motor de reglas ha ido definido como un componente que ofrece dos tipos de interfaces (Figura 23),para crear y procesar información (ItfMotorReglas) y para configurar las trazas de ejecución (ItfConfigMotorDeReglas).

ItfMotorReglas proporciona dos tipos de operaciones para :

- Crear un motor/procesador que funcione de acuerdo con las reglas definidas, para ello es necesario:
  - Compilar las reglas que definen el proceso de consecución de los objetivos del agente (operaciones compileRules ();
  - Añadir variables globales al motor una vez compiladas con éxito las reglas (addGlobalVariable()).
  - Ejecutar las reglas para que el motor comience a funcionar (fireRules ())

- Procesar información. La información recibida consiste en objetos. El modelo de procesamiento es el de un sistema de reglas de producción basado en el algoritmo RETE (http://en.wikipedia.org/wiki/Production_system). El ciclo de funcionamiento de estos sistemas esta definido por el siguiente diagrama:

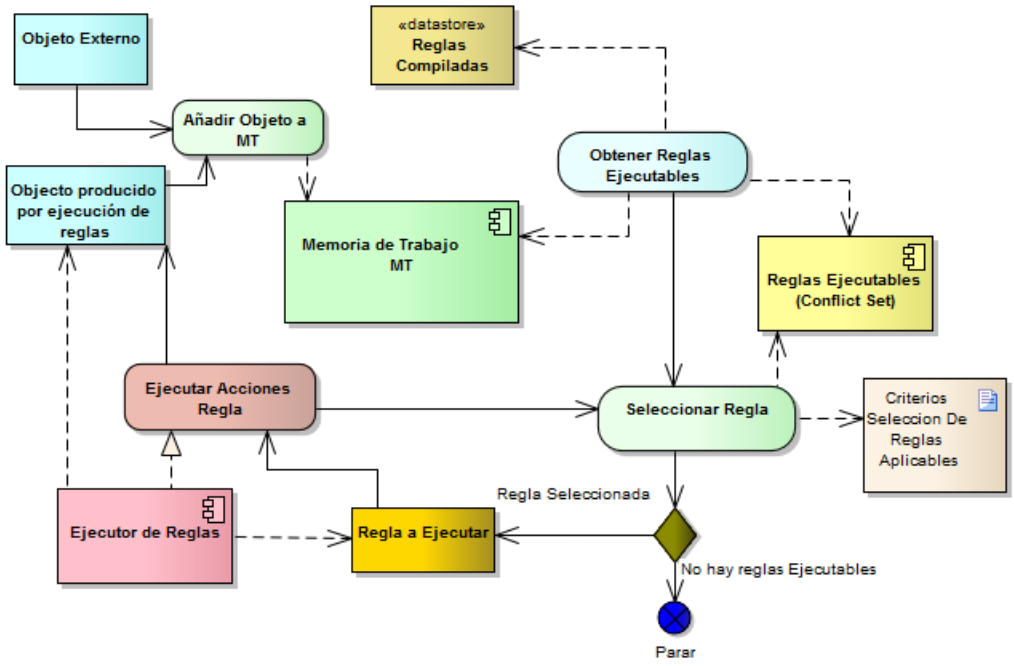


Figura 23: Ciclo de funcionamiento de un motor de reglas de producción

- La Base de Reglas (Rule Base) esta formada por un conjunto de reglas compiladas <condición> <acción>
- La Memoria de Trabajo (Working Memory), contiene items de información (objetos en el caso de Drools). Inicialmente la MT estará vacía, aunque también puede inicializarse con objetos específicos en el momento de la creación.
- Cuando el motor recibe una orden para *añadir información a la MT* se inicia el siguiente ciclo:

  Obtener el conjunto de reglas ejecutables. Una regla de la BR es ejecutable si existen objetos en la MT que hacen que las condiciones expresadas en parte izquierda de la regla se satisfaga. Con las reglas ejecutables se construye un Conjunto de Conflictos (Conflict Set), porque el conflicto para el motor es determinar qué regla debe ejecutar. A partir del CC se inicia un ciclo de selección y de ejecución de las reglas.

  Mientras haya reglas en el CC

  Seleccionar una regla del conjunto de reglas ejecutables CC . Para ello se pueden definir distintos criterios de selección, aunque el motor utiliza algunos por defecto. En general lo que hace el motor es **ordenar las reglas del CC** según diferentes criterios, por ejemplo la prioridad, la más reciente, la que tiene las condiciones más complejas, etc. y seleccionar la primera regla según el orden establecido.

  Ejecutar las acciones de la parte derecha de la regla teniendo en cuenta los objetos que hacen que la regla se satisfaga.

  Fin mientras;

Si las acciones de las reglas generan información y la añaden a la MT entonces el motor obtendrá las nuevas reglas aplicables y las añadirá al CC.

Si resulta que no hay reglas en el CC **el motor se para**. Pero podrá reactivarse cuando se añadan nuevas informaciones a la MT.

El algoritmo RETE permite construir de forma eficiente el CC a partir de las informaciones que se añaden a la MT. Para más detalles consultar el manual de Drools "Drools Exper User Guide.

La interfaz del motor proporciona un conjunto de operaciones, no sólo para añadir objetos a la MT sino también para modificar elementos y para eliminarlos.

public void assertFact(Object fact);

public void retracttFact(Object objeto);

public void updateFact(Object objeto);

Se puede indicar también si se activa o no el ciclo de comprobación y de ejecución de las reglas una vez ejecutada la operación sobre MT

public void assertFactWithoutFireRules(Object objeto);

public void retractFactWithoutFireRules(Object objeto);

public void updateFactWithoutFireRules(Object objeto);

La interfaz de configuración –ItfConfigMotorDeReglas-: Proporciona operaciones para configurar el tipo de trazas que puede generar el motor. Estas operaciones obtienen la información generada por Drools, y la redirigen hacia el recurso de trazas que permite clasificar las trazas y visualizarlas. La información puede sacarse por consola o guardarse en un fichero. Las operaciones en la interfaz son las siguientes

Trazado de las operaciones realizadas por el motor y de los objetos recibidos como parámetros .

- Trazado de las reglas que se activan cuando se ejecutan operaciones sobre el motor:
  public void setDepuracionActivationRulesDebugging (boolean boolValor);
- Trazado de los objetos que se insertan o se modifican en el motor
  public void setDepuracionHechosInsertados (boolean boolValor);
  public void setDepuracionHechosModificados (boolean boolValor);
- Trazado de los objetos en la MT (*FactHandles*) que se consideran para calcular el conjunto de reglas aplicables. Las operaciones permiten especificar el conjunto de *FactHandles:* a) antes de realizar la ejecución de la parte derecha de una regla almacenada en Conjunto de reglas cuyas premisas se satisfacen (Conflict Set); b) después de que la regla se haya ejecutado; c) después de realizar operaciones de inserción de objetos, de eliminación de objetos o de modificación de objetos en la MT.
  public void setFactHandlesMonitoring_beforeActivationFired_DEBUGGING (boolean boolValor);
  public void setfactHandlesMonitoring_afterActivationFired_DEBUGGING (boolean boolValor);
  public void setFactHandlesMonitoring_DEBUGGING (boolean boolValor);
  public void setFactHandlesMonitoringINSERT_DEBUGGING (boolean boolValor);
  public void setFactHandlesMonitoringRETRACT_DEBUGGING (boolean boolValor);
  public void setFactHandlesMonitoringUPDATE_DEBUGGING (boolean boolValor);

4.1.2 Interacción entre la percepción, el motor de reglas y las tareas

Cuando el agente recibe un mensaje extrae su contenido a través de la percepción y envía el contenido al motor de reglas. El motor actualiza el conjunto de reglas ejecutables, selecciona una regla y la ejecuta.

El diagrama Figura 16 ilustra la interacción entre los componentes que intervienen en el proceso de ejecución de una regla , cuando en su parte derecha se especifica la ejecución de una tarea (ver ejemplo de regla *rule "Validacion de datos Iniciales de Acceso"* en la sección precedente) .

El proceso puede iniciarse cuando la percepción recibe un evento o un mensaje cuyo contenido se almacena en el objeto extractedInfo . La percepción lo envía al procesador de objetivos y este lo envía al motor para que lo procese. Si la información recibida hace que se satisfaga y que se seleccione para ejecutar una regla donde se especifica la ejecución de una tarea, entonces el motor le pedirá al Gestor de Tareas que cree la tarea y una vez creada le pedirá a la tarea que se ejecute pasándole una colección de argumentos.

En la definición del método ejecutar de la tarea se comprueba el número y el tipo de los argumentos y si es correcto se ejecutan las acciones declaradas en el método.

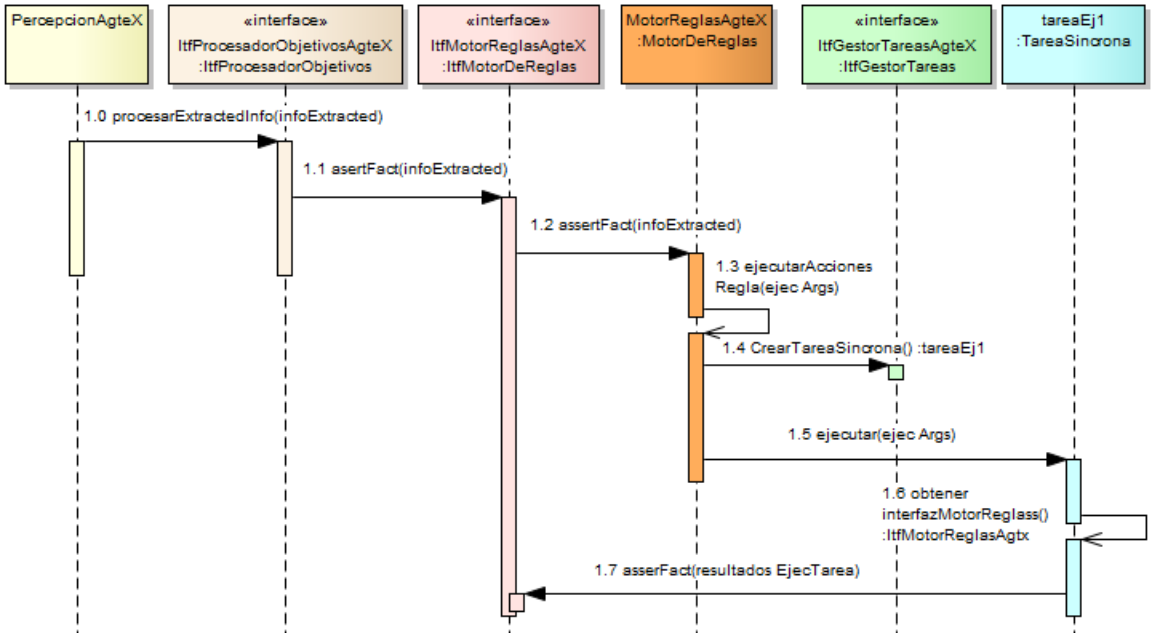


Figura 24: Interacción Percepción Motor Gestor Tareas

Las tareas envían sus resultados al motor para que continúe el ciclo de consecución de los objetivos. Para ello acceden a la interfaz del motor sin tener que utilizar la interfaz del procesador de objetivos. De esta forma al motor pueden llegar de forma concurrente informaciones procedentes de la percepción e informaciones generadas por las tareas, pero cuando las tareas son síncronas se prioriza la asimilación de resultados generados por las tareas a los resultados procedentes de la percepción.

A modo de resumen:

- Las tareas son creadas por el Gestor de Tareas a petición del motor de reglas cuando se ejecuta la parte derecha de una regla.
- El método ejecutar de las tareas admite un conjunto variable de argumentos de tipo Object (public void ejecutar(Object... params)).
- Una vez creada la tarea por el Gestor de Tareas se ordena su ejecución pasando como argumentos variables definidas en la parte izquierda de la regla , variables globales u otros objetos conocidos por el motor (ver ejemplo de regla rule "Validacion de datos Iniciales de Acceso" en la sección precedente)
- El tipo de los argumentos se define en la operación *ejecutar* de la tarea.
- La tarea envía los resultados obtenidos al motor recuperando la interfaz y utilizando las operaciones para insertar objetos en la memoria de trabajo, modificar objetos (updateFact), o eliminar objetos (modify).
- A partir de las operaciones realizadas el motor iniciará un nuevo ciclo: actualizará el conjunto de reglas aplicables, seleccionará la que proceda y la ejecutará, o se quedará inactivo si no hay reglas ejecutables.

5 Problemas y soluciones en la utilización del patrón ADO

Esta sección tiene como objetivo describir de forma ordenada los problemas encontrados por los desarrolladores y dar soluciones para resolverlos.

La utilización del patrón platea dos tipos de dificultades:

D**e tipo conceptual**. En primer lugar cómo determinar si se necesita un agente ADO u otro tipo de agente ? Cuantos agente de este tipo debe incluir en mi aplicación ? Cuantos objetivos deben tener los agentes ? Es mejor definir un agente con muchos objetivos o varios con menos objetivos . Cual es el proceso de consecución de los objetivos ? Es mejor tener tareas complejas o tareas más sencillas ?

De implementación. El comportamiento de los agentes se define mediante objetivos, reglas, tareas y clases que modelan el dominio de aplicación. La utilización de reglas como mecanismo de computacional para implementar el control del agente es uno de los primeros obstáculos a superar. El uso de reglas implica aprender el formalismo de definición y las características de su intérprete, funcionamiento, herramientas, requisitos de integración y otros. Por otro lado los agentes reciben eventos de los recursos y mensajes de otros agentes, obtienen información a través las interfaces de los recursos, y generan información propia mediante las tareas. El procesamiento de todo el flujo de información asíncrona puede ser una fuente importante de problemas.

Dejaremos de lado los problemas conceptuales que deben resolverse en la etapa de diseño y son muy dependientes del tipo de aplicación a realizar, para centrarnos en los problemas de implementación.

Definición del comportamiento de los agentes

Definición y ejecución de las reglas Drools

Se ha comentado que Drools Expert es un interprete de reglas que puede utilizarse con múltiples propósitos. En ICARO se recomienda utilizarlo de forma restringida fijándose en el proceso de generación y de consecución de los objetivos, pero nada impide que el programador haga lo que se le parezca más conveniente . Con esto se pretende obtener cierta estructuración de las reglas, limitar su complejidad y facilitar los procesos de detección y de corrección de errores.

En general los problemas con las reglas y con sus interpretes **que no siguen un ciclo computacional imperativo** son los siguientes:

- **La regla que debería ejecutarse no se ejecuta**. Parece que se han generado todos los objetos necesarios para que las premisas se satisfagan pero la regla que queremos que se ejecute No se ejecuta.
- **Se ejecutan reglas que no queremos que se ejecuten**. Se espera que tras generar o recibir una información se ejecute una determinada regla pero en su lugar se ejecutan otra o si inicia un proceso incontrolado de ejecución de otras reglas y de las acciones asociadas.
- **Se inserta en el motor recibe un objeto que debería activar una regla y ejecutarse, pero no ocurre nada** por varias posibles causas 1) el objeto generado no aparece en el motor aunque trazando la interfaz aparece como insertado; 2) el objeto generado llega pero no activa ninguna regla nueva; 3) el objeto llega y activa la regla pero no se ejecuta.

La solución común a este tipo de problemas consiste en

- Revisar el estado de la memoria de trabajo y del conjunto de reglas activadas
- Verificar y trazar los hechos que activan las reglas que se ejecutan
- Trazar las reglas ejecutadas
- Trazar las acciones que se ejecutan en la parte derecha de la regla y comprobar que la información generada se inserta en el motor y activa nuevas reglas.

Drools proporciona funciones de trazado para monitorizar y para trazar el comportamiento del motor. Alguna de estas funciones han sido incorporadas en las trazas de ICARO y agrupadas por cada agente definido.

Por el momento esta sección va permanecer abierta para poder incorporar dudas y problemas concretos que aparezcan en los proyectos.

6 Referencias