7 MEDIOS AMBIENTES

Una de las extensiones presentes en Jason consiste en la posibilidad de asociar explícitamente un **medio ambiente** a los SMA que definamos. Aunque esto no es mandatorio, la interacción entre los agentes y su medio ambiente puede ser importante, de ahí que Jason provea un conjunto de clases y métodos de Java con este fin.

Ambiente

En algunas ocasiones, el medio ambiente de los agentes es el **mundo real**, por ejemplo en robótica, control de plantas de manufactura, etc. En otros casos, el ambiente puede ser un sistema computacional como Internet, una aplicación o un sistema operativo. La conveniencia de usar el mundo real como ambiente, o no, ha sido ampliamente discutida por Brooks [27] y Etzioni [58] y otros. Aquí enfatizaremos el caso donde es necesario crear un **modelo computacional** del ambiente real y simular los aspectos dinámicos del mismo, por ejemplo, en simulaciones de sistemas complejos [128], incluyendo las simulaciones sociales [43, 180].

Ambientes reales y virtuales

Modelo del ambiente

Cuando se trabaja con cualquier sistema computacional distribuido, como es el caso de Internet, resulta conveniente adoptar una noción explícita del ambiente en el diseño del SMA. Aún en los casos donde el ambiente de los agentes sea el mundo real, un modelo del ambiente es de suma utilidad. Consideren que los SMA suelen ser grandes y complejos, de forma que su **verificación** y **validación** suele ser una tarea difícil. Si bien, se tienen avances en herramientas de verificación formal [19] derivadas del Model Checking [37] para $AgentSpeak(L)$, sigue siendo una práctica común evaluar el desempeño de un SMA bajo un ambiente específico mediante **simulaciones**. Esto no significa que los agentes tengan que situarse necesariamente en un ambiente simulado, normalmente la arquitectura de los agentes se puede redefinir para ajustarla a nuestras necesidades, de forma que se pueden tener métodos en la arquitectura que ligan al agente con el simulador y otros que lo ligan con su ambiente real.

Verificación y validación

Simulación

Si bien Java pareciera proveer una abstracción adecuada para modelar un medio ambiente, a veces resulta demasiado general. Es por ello que se han diseñado lenguajes de alto nivel, como ELMS [138], para **modelar** e implementar ambientes de simulaciones basadas en SMA. En este capítulo abordaremos las dos aproximaciones más usadas en Jason: Primero, la definición de ambiente tal y como Jason la provee a partir de la clase de Java `Environment` [15]; Segundo, una extensión al concepto de ambiente basado en las ideas del paradigma conocido como Agentes y Artefactos [139, 158].

Modelado

7.1 AMBIENTES JAVA PROVISTOS POR JASON

La mayoría de los SMA implementados en Jason están ligados a un medio ambiente escrito en **Java**. La arquitectura general de un agente Jason, escrita

Ambientes Java

también en Java, define los métodos necesarios para la interacción entre el agente y su ambiente. Esta interfaz solo necesita modificarse si el ambiente no está implementado en Java. La secuencia UML de la Figura 7.1 muestra esta interacción.

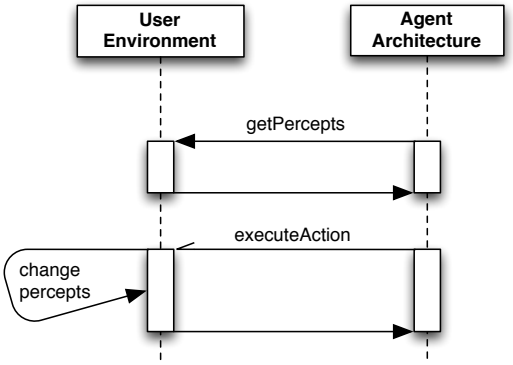

Figura 7.1: Interacción entre el ambiente y la arquitectura de un agente Jason. Adaptado de Bordini, Hübner y Wooldridge [15].

Clase Environment

Normalmente, un ambiente se define extendiendo la clase `Environment` y sobrecargando los métodos `executeAction` e `init`. La Figura 7.2 despliega los diagramas de clase pertinentes. El Cuadro 7.1 muestra algunos de los métodos disponibles para la clase `Environment`. Una clase definida por el usuario que implementa un ambiente en Jason suele tener la estructura del código mostrado en el Cuadro 7.2.

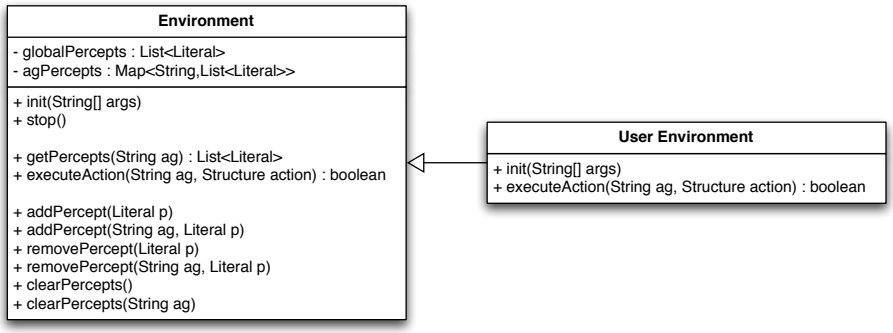

Figura 7.2: Diagramas de clase de `Environment` y `User Environment` Adaptado de Bordini, Hübner y Wooldridge [15].

Método	Descripción
`addPercept(L)`	Agrega la literal L a la lista de percepciones globales, todos los agentes perciben L.
`addPercept(A,L)`	Agrega la literal L a la lista de percepciones del agente A.
`removePercept(L)`	Elimina la literal L de la lista de percepciones globales.
`removePercept(A,L)`	Elimina la literal L de la lista de percepciones del agente A.
`clearPercepts()`	Borra todas las percepciones la lista global.
`clearPercepts(A)`	Borra todas las percepciones de la lista del agente A.

Cuadro 7.1: Algunos métodos de la clase `Environment`.

```java
import jason.asSyntax.*;
import jason.environment.*;

public class <EnvironmentName> extends Environment {
  // Otros los miembros de la clase

  @override
  public void init(String[] args) {
    // Estableciendo la creencia inicial p(a):
    addPercept(Literal.parseLiteral("p(a)"));
    // Si no usamos el Supuesto del Mundo Cerrado:
    addPercept(Literal.parseLiteral("~q(b)"));
    // Solo el agente "ag1" percibe p(a):
    addPercept("ag1", Literal.parseLiteral("p(a)"));
  }

  @override
  public void stop() {
    // Lo que se deba hacer cuando el sistema se detenga.
  }

  @override
  public boolean executeAction(String ag, Term act) {
    // Acciones
  }
}
```

Cuadro 7.2: Estructura general del código de un ambiente. `<EnvironmentName>` debe ser substituido por el nombre que se le ha dado al ambiente en el archivo de configuración del SMA en el archivo con extensión `mas2j`.

Percepciones

La arquitectura del agente usa el método `getPercepts` para obtener las **percepciones** de un ambiente simulado, a las cuales el agente tiene acceso. Es decir, aquellas propiedades del ambiente que son observables para un agente en particular. Las percepciones que el ambiente provee se ven reflejadas como creencias en los agentes, cuya fuente es el ambiente. Por ello, todas las creencias que los agentes forman con sus percepciones, van etiquetadas con la anotación `[source(percept)]`.

Literales

Es una práctica común usar el constructor de la clase `Environment` para implementar la lista de percepciones iniciales. Unicamente es posible agregar como percepciones objetos de la clase `Literal`, que es parte del paquete `jason`. Es común usar el método `parseLiteral` de la clase `Literal` con este fin. Muchos de los ambientes en el directorio de ejemplos de la distribución de Jason usan esta aproximación. Alternativamente, el método `init` permite recibir **parámetros** para la clase del ambiente definida por el usuario desde el archivo de configuración del Sistema Multi-Agentes.

Parámetros

Negación fuerte

Observen que es posible agregar percepciones positivas y negativas a los agentes, lo cual permite definir ambientes donde no se asume el Supuesto del Mundo Cerrado [157]. Las literales negativas usan el operador de **negación fuerte** tilde (∼). Además, la clase `Environment` soporta la implementación de percepciones individualizadas, de forma que el programador pueda asociar ciertas percepciones a ciertos agentes.

Sincronización

El acceso a las listas de percepciones se **sincroniza** automáticamente, pero

dependiendo de como hemos modelado el ambiente, quizás sea necesaria otra forma de sincronización al ejecutar el método `executeAction()`.

Acciones

En el caso de las acciones, el método `executeAction` se utiliza para ejecutar una **acción externa**. Recordemos que cuando una intención está siendo ejecutada y la fórmula que se ejecuta es una acción externa, se solicita la ejecución de la acción en cuestión y la intención queda **suspendida** hasta que el ambiente regresa una señal de retroalimentación, indicando si la acción se ejecutó o no. Como la intención que incluye la acción que está siendo ejecutada ha sido suspendida; y el ciclo de razonamiento del agente sigue adelante mientras la arquitectura interacciona con el ambiente, el efecto es que pareciera que el método `executeAction` es invocado asíncronamente. Si además, el ambiente está siendo ejecutado remotamente, en una computadora diferente a la que ejecuta al agente, la **duración** de esta suspensión puede ser significativo.

Intención suspendida

Tiempo de respuesta

Ejecución de una acción

Cuando un agente intenta **ejecutar** una acción externa, el nombre del agente y un término representando la acción elegida son enviados como parámetros al método `executeAction()` que hará los cambios pertinentes en el modelo del ambiente, lo cual normalmente implica cambiar las percepciones generadas: lo que es verdadero o falso del ambiente, se cambia de acuerdo a la acción ejecutada. Observen que la ejecución de una acción necesariamente regresa un valor booleano, que indica si el intento del agente por ejecutar la acción tuvo **éxito** o no. Un plan falla si cualquiera de sus acciones básicas no tienen éxito.

Retroalimentación

Accesibilidad

La posibilidad de definir percepciones individualizadas debe usarse para especificar los aspectos del ambiente que son **accesibles** a cada agente, si es el caso que el SMA está compuesto de agentes heterogéneos. Este efecto, al menos técnicamente, se puede implementar también modificando la arquitectura del agente, al especificar que creencias adquiere el agente a partir de sus percepciones. Sin embargo, esta última posibilidad debería usarse para modelar por ejemplo, **ruido** en la percepción y no las capacidades de percepción del agente.

Ruido

Persistencia

Nota mental

Hay dos errores de novato muy comunes, relacionados con el modelado y la implementación de los ambientes en Jason. El primero de ellos tiene que ver con la **persistencia** de las percepciones y consiste en esperar que un agente mantenga una creencia percibida en su estado mental, aun cuando la percepción de ésta solo dura un ciclo de razonamiento y es, en consecuencia, eliminada de la lista de percepciones del ambiente. Si se desea que el agente recuerde estas creencias que ya no son percibidas, el agente debería usar planes para crear lo que se conoce como una **nota mental** en respuesta a la percepción de interés. Las notas mentales permanecen en el estado mental del agente hasta ser eliminadas explícitamente. En contrapartida, las percepciones del agente se eliminan de la base de creencias del agente tan pronto como dejan de ser percibidas como verdaderas. Otra observación importante: si una percepción se añade como verdadera en el ambiente y posteriormente es eliminada por alguna acción, puede que no sea percibida por todos los agentes que tienen acceso a ella. En contra partida, si una propiedad es percibida, el proceso de actualización de creencias creará el evento correspondiente inmediatamente; lo mismo si la propiedad deja de ser percibida.

El segundo error típico está relacionado con la consistencia de tipos entre los nombres de las percepciones y acciones usados en el modelo del ambiente; y aquellos usados en el código $AgentSpeak(L)$ de los agentes. En principio, las metodologías de software orientadas a objetos deberían ayudar a evitar este tipo de errores. También es posible usar **ontologías** que incluyan los términos usados en los agentes, ambiente y comunicaciones, para evitar tales errores.

Ontologías

7.1.1 Modelo, Vista, Controlador

En esta sección veremos como podemos diseñar un ambiente en Jason siguiendo el **patrón de diseño** conocido como Modelo-Vista-Controlador, normalmente usado para el modelado e implementación de interfaces gráficas en los lenguajes orientados a objetos. Desde esta perspectiva el diseño del ambiente se conforma por tres elementos:

MVC

MODELO. Este elemento mantiene la información acerca del estado del ambiente y su dinámica. Por ejemplo, la posición de un agente robot y su nueva posición cuando éste se mueve.

VISTA. Este elemento especifica despliega de forma adecuada el ambiente, usando la información del modelo.

CONTROLADOR. Este elemento interactua con los agentes e invoca cambios en el ambiente y, algunas veces, en el modelo.

Para ejemplificar este tipo de ambiente utilizaremos un ejemplo adaptado del robot domestico [1], que acompaña a la distribución de Jason. Recordemos que en este ejemplo, un robot doméstico tiene como meta servir cerveza a su dueño. Su tarea es bastante simple: cuando recibe una solicitud de cerveza por parte de su dueño, va al refrigerador, toma una y se la lleva. El robot debe estar atento del cuantas cervezas hay el en refrigerador y eventualmente ordenar más cervezas al servicio de entregas del supermercado. Además, la Secretaría de Salud Pública ha alambrado en el robot un limite en el consumo diario de cervezas que también debe tomarse en cuenta.

El diagrama general Prometheus [142] de este proyecto luce como se muestra en la Figura 7.3. El robot percibe dos lugares con el predicado $at/2$, el sitio donde está el refrigerador y el sitio donde está su dueño. Tambien puede percibir el número de cervezas mediante el predicado $stock/2$ y si el dueño tiene cerveza con el predicado $has/2$. Las acciones del robot se explican solas. Quizás solo sea necesario mencionar que $move\_towards/1$ ha sido simplificada considerablemente en el proyecto.

La Figura 7.4 muestra el diagrama de clases para este ambiente. Para proveer las percepciones que se muestran la Figura 7.3, el modelo de este ejemplo debe maneter información sobre el número de cervezas disponibles en el refrigerador; si el dueño tiene una cerveza; y la posición del robot.

La percepción $has(owner, beer)$ se basa en el valor del contador $sipCount$. Cada que el dueño se hace de una cerveza, a este contador se le asigna un

1 /examples/domestic-robot/

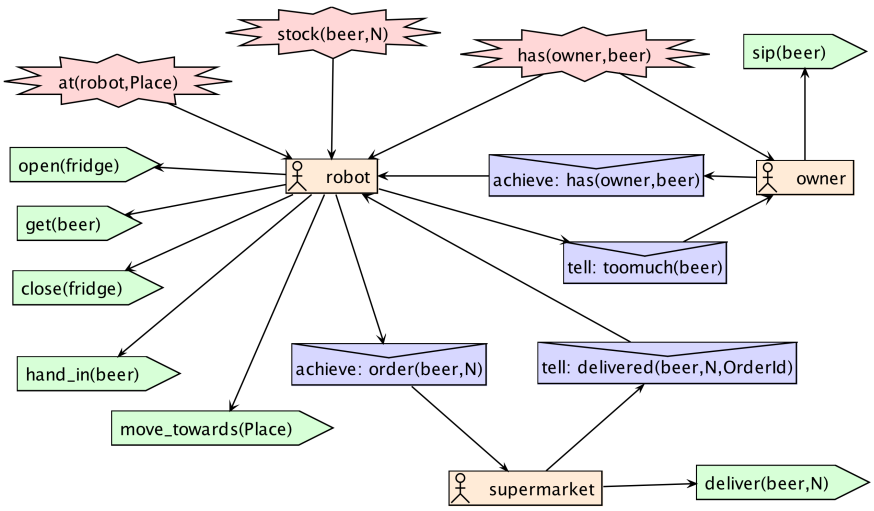

Figura 7.3: Vista general de la aplicación del robot domestico reportada por Bordini, Hübner y Wooldridge [15] en la página 58.

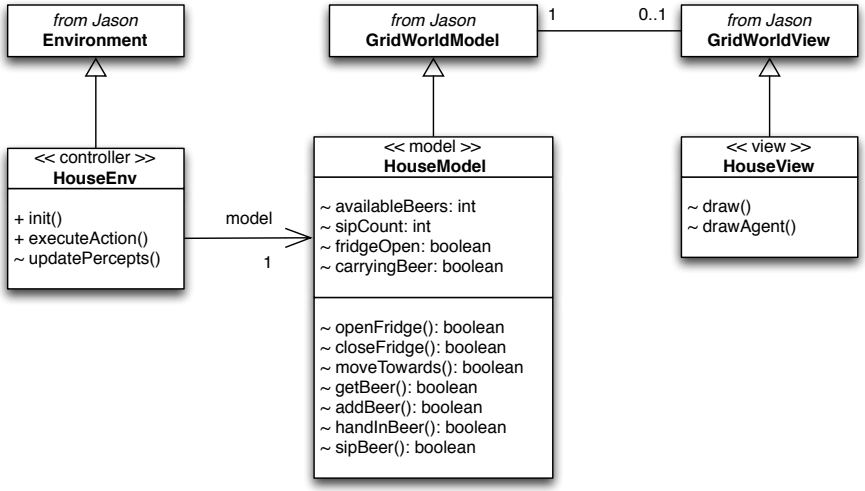

Figura 7.4: Diagrama de clases para el ambiente del robot doméstico, adaptado de Bordini, Hübner y Wooldridge [15].

valor de 10. Su valor decrece en 1 cada que le da un sorbo a su cerveza (la acción $sip(beer)$ es ejecutada). Esta percepción es accesible tanto para el robot como para el dueño mientras $sipCount > 0$. El modelo tiene otros dos atributos booleanos: $fridgeOpen$ relacionado con el estado de la puerta del refrigerador y usado para decidir si el robot puede percibir o no la cantidad de cervezas disponibles; $carryingBeer$ es verdadera si el robot lleva una cerveza en su mano. La acción $hand\_in(beer)$ solo tiene éxito si el robot efectivamente lleva una cerveza en su mano. En cualquier otro caso, la acción falla.

Aunque solo sería estrictamente necesario saber que el robot está en el refrigerador o junto al usuario, usaremos el modelo **GridWorldModel**, provisto por Jason, para mantener la posición del robot. Esta clase representa el ambiente como una rejilla de $n \times n$ posiciones. Cada posición debe contener objetos. La ventaja de usar este modelo es que la vista también es provista

por Jason (ver Figura 7.5), de forma que no necesitaremos implementar directamente una buena parte de la interfaz gráfica de esta aplicación. La clase `GridHouseView` implementa dos métodos: uno para dibujar objetos estáticos, como el refrigerador; y otra para dibujar el robot.

Figura 7.5: La interfaz gráfica del robot doméstico, tal y como la provee su vista.

Veamos como se ve esto en código. El archivo principal del SMA debe especificar el ambiente en que estarán situados sus agentes. Esto se hace mediante la palabra reservada `Environment`:

```
1  /* Jason Project
2
3     See Prometheus specification in doc folder
4
5  */
6
7  MAS domestic_robot {
8      infrastructure: Centralised
9      environment: HouseEnv(gui) // use "nogui" as parameter to not show the GUI
10     agents: robot;
11             owner;
12             supermarket;
13     aslSourcePath:
14       "src/asl";
15 }
```

El ambiente de este SMA se define como sigue,en el archivo `HouseEnv.java`. Primero se importan las clases necesarias y se definen algunas literales auxiliares. El `logger` es opcional, suele usarse para llevar un registro de la corrida detallada del sistema. Finalmente se define el modelo del ambiente:

```
1  import jason.asSyntax.*;
2  import jason.environment.Environment;
3  import jason.environment.grid.Location;
4  import java.util.logging.Logger;
5
6  public class HouseEnv extends Environment {
7
8    // common literals
9    public static final Literal of = Literal.parseLiteral("open(fridge)");
```

```
10  public static final Literal clf = Literal.parseLiteral("close(fridge)");
11  public static final Literal gb = Literal.parseLiteral("get(beer)");
12  public static final Literal hb = Literal.parseLiteral("hand_in(beer)");
13  public static final Literal sb = Literal.parseLiteral("sip(beer)");
14  public static final Literal hob = Literal.parseLiteral("has(owner,beer)");
15
16  public static final Literal af = Literal.parseLiteral("at(robot,fridge)");
17  public static final Literal ao = Literal.parseLiteral("at(robot,owner)");
18
19  static Logger logger = Logger.getLogger(HouseEnv.class.getName());
20
21  HouseModel model; // the model of the grid
```

El método inicial define la vista del modelo, en caso de que el parámetro del ambiente haya sido `"gui"`.

```
1   @Override
2   public void init(String[] args) {
3     model = new HouseModel();
4
5     if (args.length == 1 && args[0].equals("gui")) {
6       HouseView view = new HouseView(model);
7       model.setView(view);
8     }
9
10    updatePercepts();
11  }
```

Posteriormente actualiza la percepción de los agentes:

```
1   void updatePercepts() {
2     // clear the percepts of the agents
3     clearPercepts("robot");
4     clearPercepts("owner");
5
6     // get the robot location
7     Location lRobot = model.getAgPos(0);
8
9     // add agent location to its percepts
10    if (lRobot.equals(model.lFridge)) {
11      addPercept("robot", af);
12    }
13    if (lRobot.equals(model.lOwner)) {
14      addPercept("robot", ao);
15    }
16
17    // add beer "status" the percepts
18    if (model.fridgeOpen) {
19      addPercept("robot", Literal.parseLiteral("stock(beer," + model.availableBeers + ")"));
20    }
21    if (model.sipCount > 0) {
22      addPercept("robot", hob);
23      addPercept("owner", hob);
24    }
25  }
```

La ejecución de las acciones de los agentes de define como sigue, por casos:

```
1   public boolean executeAction(String ag, Structure action) {
2     System.out.println("[" + ag + "] doing: " + action);
3     boolean result = false;
```

```
4     if (action.equals(of)) { // of = open(fridge)
5       result = model.openFridge();
6
7     } else if (action.equals(clf)) { // clf = close(fridge)
8       result = model.closeFridge();
9
10    } else if (action.getFunctor().equals("move_towards")) {
11      String l = action.getTerm(0).toString();
12      Location dest = null;
13      if (l.equals("fridge")) {
14        dest = model.lFridge;
15      } else if (l.equals("owner")) {
16        dest = model.lOwner;
17      }
18
19      try {
20        result = model.moveTowards(dest);
21      } catch (Exception e) {
22        e.printStackTrace();
23      }
24
25    } else if (action.equals(gb)) {
26      result = model.getBeer();
27
28    } else if (action.equals(hb)) {
29      result = model.handInBeer();
30
31    } else if (action.equals(sb)) {
32      result = model.sipBeer();
33
34    } else if (action.getFunctor().equals("deliver")) {
35      // wait 4 seconds to finish "deliver"
36      try {
37        Thread.sleep(4000);
38        result = model.addBeer((int) ((NumberTerm) action.getTerm(1)).solve());
39      } catch (Exception e) {
40        logger.info("Failed to execute action deliver!" + e);
41      }
42
43    } else {
44      logger.info("Failed to execute action " + action);
45    }
46
47    if (result) {
48      updatePercepts();
49      try {
50        Thread.sleep(100);
51      } catch (Exception e) {
52      }
53    }
```

Observen que el método `executeAction` recibe el nombre de un agente y la acción que se va a ejecutar. Las acciones regresan el valor `true` si se ejecutan y `false` en otro caso. Esto no tiene que ver con la correctez de la acción, es decir con que el efecto pretendido se haya conseguido, sol se trata de saber si la acción se ha ejecutado o no. Observen que el hilo donde se ejecuta la acción puede dormirse, ya sea para dar tiempo a terminar los procesos; o bien para simular la duración de las acciones. Si la acción se ha completado, entonces se actualizan las percepciones de los agentes.

La ejecución de las acciones afecta el modelo del ambiente. Las propiedades de éste se definen como sigue, en el archivo `HouseModel.java`:

```
import jason.environment.grid.GridWorldModel;
import jason.environment.grid.Location;

/** class that implements the Model of Domestic Robot application */
public class HouseModel extends GridWorldModel {

  // constants for the grid objects
  public static final int FRIDGE = 16;
  public static final int OWNER = 32;

  // the grid size
  public static final int GSize = 7;

  boolean fridgeOpen = false; // whether the fridge is open
  boolean carryingBeer = false; // whether the robot is carrying beer
  int sipCount = 0; // how many sip the owner did
  int availableBeers = 2; // how many beers are available

  Location lFridge = new Location(0, 0);
  Location lOwner = new Location(GSize - 1, GSize - 1);

  public HouseModel() {
    // create a 7x7 grid with one mobile agent
    super(GSize, GSize, 1);

    // initial location of robot (column 3, line 3)
    // ag code 0 means the robot
    setAgPos(0, GSize / 2, GSize / 2);

    // initial location of fridge and owner
    add(FRIDGE, lFridge);
    add(OWNER, lOwner);
  }
```

Las acciones en si, se modelan de la siguiente manera:

```
  boolean openFridge() {
    if (!fridgeOpen) {
      fridgeOpen = true;
      return true;
    } else {
      return false;
    }
  }

  boolean closeFridge() {
    if (fridgeOpen) {
      fridgeOpen = false;
      return true;
    } else {
      return false;
    }
  }

  boolean moveTowards(Location dest) {
    Location r1 = getAgPos(0);
    if (r1.x < dest.x)
      r1.x++;
```

```
    else if (r1.x > dest.x)
      r1.x--;
    if (r1.y < dest.y)
      r1.y++;
    else if (r1.y > dest.y)
      r1.y--;
    setAgPos(0, r1); // move the robot in the grid

    // repaint the fridge and owner locations
    if (view != null) {
      view.update(lFridge.x, lFridge.y);
      view.update(lOwner.x, lOwner.y);
    }
    return true;
  }

  boolean getBeer() {
    if (fridgeOpen && availableBeers > 0 && !carryingBeer) {
      availableBeers--;
      carryingBeer = true;
      if (view != null)
        view.update(lFridge.x, lFridge.y);
      return true;
    } else {
      return false;
    }
  }

  boolean addBeer(int n) {
    availableBeers += n;
    if (view != null)
      view.update(lFridge.x, lFridge.y);
    return true;
  }

  boolean handInBeer() {
    if (carryingBeer) {
      sipCount = 10;
      carryingBeer = false;
      if (view != null)
        view.update(lOwner.x, lOwner.y);
      return true;
    } else {
      return false;
    }
  }

  boolean sipBeer() {
    if (sipCount > 0) {
      sipCount--;
      if (view != null)
        view.update(lOwner.x, lOwner.y);
      return true;
```

Ahora solo resta implementar la vista en `HouseView.java`. Primero definimos la vista y su modelo:

```
import jason.environment.grid.*;

import java.awt.Color;
import java.awt.Font;
```

```
import java.awt.Graphics;

/** class that implements the View of Domestic Robot application */
public class HouseView extends GridWorldView {

  private static final long serialVersionUID = 1L;

  HouseModel hmodel;

  public HouseView(HouseModel model) {
    super(model, "Domestic Robot", 700);
    hmodel = model;
    defaultFont = new Font("Arial", Font.BOLD, 16); // change default font
    setVisible(true);
    repaint();
```

Posteriormente definimos los métodos para dibujar a los objetos en el ambiente:

```
  /** draw application objects */
  @Override
  public void draw(Graphics g, int x, int y, int object) {
    Location lRobot = hmodel.getAgPos(0);
    super.drawAgent(g, x, y, Color.lightGray, -1);
    repaint();
    switch (object) {
    case HouseModel.FRIDGE:
      if (lRobot.equals(hmodel.lFridge)) {
        super.drawAgent(g, x, y, Color.yellow, -1);
      }
      g.setColor(Color.black);
      drawString(g, x, y, defaultFont, "Fridge (" + hmodel.availableBeers + ")");
      break;
    case HouseModel.OWNER:
      if (lRobot.equals(hmodel.lOwner)) {
        super.drawAgent(g, x, y, Color.yellow, -1);
      }
      String o = "Owner";
      if (hmodel.sipCount > 0) {
        o += " (" + hmodel.sipCount + ")";
      }
      g.setColor(Color.black);
      drawString(g, x, y, defaultFont, o);
      break;
```

Lo cual incluye a los agentes:

```
  }

  @Override
  public void drawAgent(Graphics g, int x, int y, Color c, int id) {
    Location lRobot = hmodel.getAgPos(0);
    if (!lRobot.equals(hmodel.lOwner) && !lRobot.equals(hmodel.lFridge)) {
      c = Color.yellow;
      if (hmodel.carryingBeer)
        c = Color.orange;
      super.drawAgent(g, x, y, c, -1);
      g.setColor(Color.black);
```

Hay tres agentes en este SMA, el central es el robot, definido como sigue:

```
/* Initial beliefs and rules */
```

```
2
3  // initially, I believe that there is some beer in the fridge
4  available(beer,fridge).
5
6  // my owner should not consume more than 10 beers a day :-)
7  limit(beer,10).
8
9  too_much(B) :-
10    .date(YY,MM,DD) &
11    .count(consumed(YY,MM,DD,_,_,_,B),QtdB) &
12    limit(B,Limit) &
13    QtdB > Limit.
14
15
16 /* Plans */
17
18 +!has(owner,beer)
19    :  available(beer,fridge) & not too_much(beer)
20    <- !at(robot,fridge);
21       open(fridge);
22       get(beer);
23       close(fridge);
24       !at(robot,owner);
25       hand_in(beer);
26       ?has(owner,beer);
27       // remember that another beer has been consumed
28       .date(YY,MM,DD); .time(HH,NN,SS);
29       +consumed(YY,MM,DD,HH,NN,SS,beer).
30
31 +!has(owner,beer)
32    :  not available(beer,fridge)
33    <- .send(supermarket, achieve, order(beer,5));
34       !at(robot,fridge). // go to fridge and wait there.
35
36 +!has(owner,beer)
37    :  too_much(beer) & limit(beer,L)
38    <- .concat("The Department of Health does not allow me to give you more than ", L,
39               " beers a day! I am very sorry about that!",M);
40       .send(owner,tell,msg(M)).
41
42
43 -!has(_,_)
44    :  true
45    <- .current_intention(I);
46       .print("Failed to achieve goal '!has(_,_)'. Current intention is: ",I).
47
48 +!at(robot,P) : at(robot,P) <- true.
49 +!at(robot,P) : not at(robot,P)
50   <- move_towards(P);
51      !at(robot,P).
52
53 // when the supermarket makes a delivery, try the 'has' goal again
54 +delivered(beer,_Qtd,_OrderId)[source(supermarket)]
55   :  true
56   <- +available(beer,fridge);
57      !has(owner,beer).
58
59 // when the fridge is opened, the beer stock is perceived
60 // and thus the available belief is updated
61 +stock(beer,0)
```

```
   :  available(beer,fridge)
   <- -available(beer,fridge).
+stock(beer,N)
   :  N > 0 & not available(beer,fridge)
   <- -+available(beer,fridge).

+?time(T) : true
  <-  time.check(T).
```

que interactua con el supermercado:

```
last_order_id(1). // initial belief

// plan to achieve the goal "order" for agent Ag
+!order(Product,Qtd)[source(Ag)] : true
  <- ?last_order_id(N);
     OrderId = N + 1;
     -+last_order_id(OrderId);
     deliver(Product,Qtd);
     .send(Ag, tell, delivered(Product,Qtd,OrderId)).
```

y por supuesto con su dueño:

```
/* Initial goals */

!get(beer).   // initial goal: get a beer
!check_bored. // initial goal: verify whether I am getting bored

+!get(beer) : true
   <- .send(robot, achieve, has(owner,beer)).

+has(owner,beer) : true
   <- !drink(beer).
-has(owner,beer) : true
   <- !get(beer).

// while I have beer, sip
+!drink(beer) : has(owner,beer)
   <- sip(beer);
     !drink(beer).
+!drink(beer) : not has(owner,beer)
   <- true.

+!check_bored : true
   <- .random(X); .wait(X*5000+2000);  // i get bored at random times
      .send(robot, askOne, time(_), R); // when bored, I ask the robot about the time
      .print(R);
      !!check_bored.

+msg(M)[source(Ag)] : true
   <- .print("Message from ",Ag,": ",M);
      -msg(M).
```

7.2 PROGRAMACIÓN DE AMBIENTES ENDÓGENOS

De alguna manera, la programación de un SMA debería resumirse en la ecuación:

$$SMA = Agentes + Ambiente$$

de forma que el ambiente ofreciese la funcionalidad para que los agentes lograsen sus metas, en al menos tres niveles:

CONTEXTO DE DESPLIEGUE. Esto es, acceso a los recursos externos de software y hardware con los que el SMA puede interactuar: sensores, actuadores, impresoras, redes, bases de datos, servicios web, etc.

CAPA DE ABSTRACCIÓN. Una interfaz entre la representación a nivel agente y los detalles de bajo nivel presentes en el contexto de despliegue, de forma que el programador no necesite acceso a estos últimos.

CAPA DE MEDIACIÓN E INTERACCIÓN. El ambiente es explotado para regular el acceso a recursos compartidos y para mediar la interacción entre agentes.

Para poder acceder plenamente a esta visión, es necesario alejarnos de la noción tradicional de ambiente como ese mundo exterior que los agentes perciben y modifican a través de sus acciones, mientras persiguen el cumplimiento de sus metas. Necesitamos abandonar concepción del ambiente como algo exógeno a los Sistemas MultiAgentes, por una concepción **endógena** donde el ambiente forma parte del sistema y se programa. Esta es la visión adoptada por la Ingeniería de Software Orientada a Agentes (*AOSE* por sus siglas en inglés), donde el ambiente es más una abstracción de primera clase que encapsula funcionalidad y servicios que soportan las actividades de los agentes. Bajo esta visión nos referirermos a Sistemas Multi-Agentes software con ambientes endógenos, donde el ambiente es una parte programable del sistema, ortogonal, pero fuertemente integrada, a la parte agente. Los agentes seguirán siendo la abstracción básica para diseñar y programar las partes **autónomas** del sistema software, en particular los componentes guiados por metas o tareas, individuales o sociales. Es decir, los agentes seguirán encapsulando la **lógica** y el **control** de las acciones del sistema. El ambiente se usará para diseñar y programar la parte computacional del sistema que ofrece **funcionalidad** a los agentes. Esto puede incluir desde el acceso al ambiente externo, hasta el diseño de estructuras computacionales para auxiliar a los agentes en su trabajo.

Ambientes endógenos

Consideren por ejemplo, que necesitamos implementar un pizarrón como medio de comunicación en un SMA. En la mayoría de los lenguajes de programación orientados a agentes, la separación que hemos mencionado no existe y por lo tanto, el pizarrón debe implementarse como un agente; lo cual es conceptualmente inadecuado, ya que por definición un pizarrón no es un agente. Si consideramos la separación recién introducida entre ambiente y agentes, el pizarrón puede implementarse como un recurso del ambiente accesible a los agentes a través de sus acciones y percepciones. De alguna forma esto se puede llevar a cabo en Jason, ya que, como hemos visto, el lenguaje provee el mecanismo para considerar explícitamente el ambiente de los agentes. En ese caso, el ambiente estará implementado en Java y el pizarrón se implementaría como un objeto manipulable por los agentes.

En realidad, necesitamos una aproximación diferente si queremos asumir la ecuación programación de agentes + programación de ambientes. Necesitamos un modelo computacional para la programación de ambientes que satisfaga la desiderata descrita a continuación:

ABSTRACTO. El modelo adoptado debe preservar el nivel de abstracción de los agentes. Esto es, los conceptos usados para programar la estructura y dinámica de los ambientes debe ser consistente con los conceptos usados para programar los agentes y su semántica. Ejemplos de esto incluyen las nociones de acción, percepción, evento, meta, tarea, etc.

ORTOGONAL. El modelo debe ser lo más ortogonal posible con respecto a los modelos, arquitecturas y lenguajes adoptados para la programación de agentes; de forma que soporte naturalmente la ingeniería de sistemas heterogéneos.

GENERAL. El modelo debe ser lo suficientemente expresivo y general como para desarrollar diferentes tipos de ambientes de acuerdo a diferentes dominios de aplicación y problemas, explotando el mismo conjunto básico de conceptos y constructores.

MODULAR. El modelo debe introducir concebir al ambiente como algo modular, evitando visiones de éste monolíticas y centralizadas.

EXTENSIÓN DINÁMICA. El modelo debe soportar la construcción dinámica, remplazo y extensión de las partes del ambiente, en una perspectiva de **sistema abierto**.

REUTILIZACIÓN. El modelo debe promover la reutilización de las partes de un ambiente en diferentes aplicaciones, contextos o dominios.

De lo anterior se puede intuir que la noción de objeto, tal y como se define en la Programación Orientada a Objetos (*OOP* por sus siglas en inglés) no debería ser usada como tal, en tanto que abstracción de primera clase en un ambiente. Por un lado, los objetos interactúan entre si mediante invocación de métodos sin que hagan referencia a acciones y percepciones; Por el otro, los lenguajes orientados a agentes no definen el concepto de invocación de métodos, de forma que no tiene mucho sentido hablar de interacción entre agentes y objetos en términos de invocación de métodos. Lo mismo sucede en marcos de trabajo donde los agentes se implementan como objetos, Jade de Bellifemine, Caire y Greenwood [9] por ejemplo. En este caso los objetos no son abstracciones de primera clase en el mundo de los agentes. De forma que en el resto de esta sección abordaremos diversos aspectos que deberían conformar un modelo de programación de ambientes.

7.2.1 Modelo de Acción

Este aspecto tiene que ver con la manera en que los agentes afectan el estado del ambiente y por tanto, con la noción de **acción externa** e involucra el tipo de semántica adoptada para definir el éxito y el fracaso de las acciones; el modelo de ejecución adoptado; y la manera de definir y programar el repertorio de acciones de los agentes.

Acción externa

La **semántica de éxito** adoptada comúnmente consiste en determinar el éxito de la acción de lado del agente, esto es, si la acción ha sido ejecutada con éxito por los efectores del agente y la acción ha sido aceptada por el ambiente, entonces la acción tuvo éxito. Sin embargo, esta semántica no dice

Exito

nada sobre los **efectos** y la **compleción** de las acciones. Si un agente quiere saber si la ejecución de una de sus acciones terminó con éxito, el agente deber revisar sus percepciones. Bajo la perspectiva de Programación de Ambientes es posible definir y explotar semánticas más ricas, en donde el éxito de la acción del lado del agente, no solo significa que la acción ha sido aceptada por el ambiente, sino que su ejecución ha sido completada y sus efectos esperados han sido producidos en el ambiente. De manera más general, en ambientes endógenos el conjunto de acciones se considera como parte de un **contrato** que el ambiente provee a los agentes que están logicamente situados en él.

Modelo de ejecución

Con respecto al **modelo de ejecución**, la semántica típicamente adoptada en los lenguajes de programación orientados a agentes consiste en modelar las acciones como **eventos**, es decir, como transiciones individuales atómicas (desde el punto de vista del agente) que cambian o inspeccionan el estado del medio ambiente. Esto es como decir que las acciones tienen una duración de cero y que la ejecución de dos acciones no puede traslaparse en el tiempo. También en este caso, la Programación de Ambientes hace posible introducir semánticas más ricas modelando la ejecución de las acciones como **procesos**, es decir, secuencias de dos o más eventos, incluído el evento representando el inicio de la ejecución de la acción y el evento que representa que la ejecución ha sido completada. Esto facilita la representación de acciones de largo término, posiblemente concurrentes; así como acciones que los agentes puedan usar para sincronizarse.

7.2.2 Modelo de Percepción

Este aspecto está relacionado con la manera como el ambiente es percibido por los agentes, la definición de los estímulos generados por el ambiente, y la correspondiente percepción resultado de este proceso. Junto con las acciones, las percepciones pueden considerarse también parte del contrato provisto por el ambiente.

Semánticas

Básicamente dos semánticas pueden adoptarse cuando definimos un modelo de percepción. En una semántica **basada en estados** el estímulo se compone de información sobre el estado actual del agente y es generado cuando el agente entra en la fase de percepción en su ciclo de ejecución. En las semánticas **basadas en eventos** el estímulo está compuesto por información sobre cambios ocurridos en el ambiente, enviados a los agentes cuando dichos cambios se producen, independientemente de la fase de ejecución en que se encuentre el agente. Por ejemplo, en la arquitectura abstracta de agente propuesta por Wooldridge [186], la función de percepción va del estado actual del ambiente E a un conjunto de percepciones P. En las arquitecturas BDI, una foto como ésta es usada para actualizar las creencias de los agentes, por ejemplo en Jason (aunque este aspecto es reconfigurable, como casi todos los componentes de este lenguaje). En cambio, en el lenguaje 2APL propuesto por Dastani [46], las percepciones son tratadas como eventos. El programador está obligado a definir explicitamente las reglas que especifican como cambian las creencias del agente cuando las percepciones son detectadas por el agente.

Como en el caso del modelo de acción, la elección de la semántica de la percepción puede tener un fuerte impacto en la dinámica de la ejecución del SMA. Por ejemplo, al adoptar una semántica basada en estados, si el ambiente cambia varias veces entre dos fases subsecuentes de percepción del agente, los cambios ocurridos serán imperceptibles para el agente.

7.2.3 Modelo del Ambiente Computacional

Este aspecto está relacionado con la funcionalidad del ambiente donde se ejecuta el programa, esto es las estructuras que definen el estado interno y su comportamiento computacional, incluyendo las computaciones que son causadas directamente por las acciones de los agentes y aquellas que representan procesos internos.

Un primer punto a considerar es el modelo adoptado para descomponer la computación del estado/comportamiento del ambiente. Lo más simple en este caso es el enfoque monolítico, donde un solo objeto computacional con un único estado representa la estructura computacional y el comportamiento del ambiente. En ese caso, tal objeto es el punto de entrada para definir los efectos de las acciones y el conjunto de estímulos generados por el ambiente. 2APL, Jason y GOAL [97] adoptan nativamente este enfoque, mediante una API basada en Java para programar el ambiente como una clase. Un enfoque más modular debería definir explícitamente estructuras de primera clase y, eventualmente, abstracciones para descomponer y modularizar la funcionalidad del ambiente. El modelo basado en **artefactos** que presentaremos en la siguiente sección es ejemplo de este enfoque. Otro ejemplo es la noción genérica de objeto como se define en GOLEM [25] y MadKit [93]. El modelo de acción puede incluir o no, dependiendo de la estructura adoptada, acciones para crear, disponer, remplazar componentes del ambiente en tiempo de ejecución.

Un aspecto relacionado al modelo de ejecución es el modelo de **concurrencia** adoptado. Esto es, cuantos hilos o flujos de control son explotados para ejecutar los procesos computacionales del ambiente; y en el caso de multiples hilos cual es el mapeo con respecto a las computaciones del ambiente: y cómo se resuelven los problemas típicos de la concurrencia como las interferencias. Es evidente que todo ello impacta en el desempeño global del sistema.

7.2.4 Modelo de Datos del Ambiente

Este aspecto tiene que ver con los tipos de datos intercambiados entre el ambiente y los agentes, cuestión relevante para codificar los parámetros de las acciones, la retroalimentación de las mismas, el contenido de los estímulos (percepciones) y su representación. Esto completa el contrato que el ambiente expone a los agentes. Las cuestiones propias de la **integración** en el mismo programa de partes desarrolladas en diferentes lenguajes de programación o marcos de trabajo, aplican aquí: en este caso, tenemos por un lado el lenguaje de programación orientado a agentes y el lenguaje de programación del ambiente.

Integración

Para atender estas cuestiones, es posible introducir un **modelo de datos** que defina explícitamente los posibles tipos de estructuras de datos presentes en las acciones y las percepciones. Para ello se puede adoptar un lenguaje de representación, basado en XML por ejemplo, e incluso el modelo de objetos de un lenguaje orientado a objetos. Del lado del lenguaje orientado a agentes es necesario especificar una forma de **vínculo** que defina como los tipos de datos del modelo de datos del ambiente se pueden traducir al modelo de datos adoptado por el lenguaje orientado a agentes y vice-versa. *Modelo de datos*

Un último aspecto debe afrontarse en el caso de los Sistemas Abiertos [68]: La definición de un modelo de datos del ambiente que permita la descripción de **ontologías** de forma que podamos definir explícitamente la semántica de los datos involucrados en la interacción agente-ambiente. A este respecto, el trabajo existente en el contexto del Web Semántico y los lenguajes y modelos adoptados para describir ontologías, como OWL, pueden explotarse apropiadamente.

7.2.5 Modelo de Distribución del Ambiente

Este aspecto tiene que ver con la distribución de un ambiente en una red de cómputo, ya sea porque esto es necesario o porque resulta oportuno. Un modelo de distribución del ambiente debería introducir una noción explicita de **localidad** para definir las porciones no distribuida del ambiente computacional; y entonces definir cómo y si las diferentes localidades que componen el ambiente interactúan entre si. Del lado del agente, el modelo de distribución del ambiente afecta su repertorio conductual, haciendo necesarias acciones para entrar y salir de las localidades. *Localidad*

De hecho, el modelo de distribución del ambiente, afecta al modelo del tiempo adoptado en el SMA. Para un SMA distribuido no es posible, ni teórica ni prácticamente, tener una sola noción de tiempo dentro del sistema para etiquetar los eventos y definir un orden total sobre ellos. Este es un tema importante, debido a que muchas formalizaciones de SMA, por ejemplo instituciones electrónicas, sistemas normativos, y organizaciones, típicamente se basan en una noción **global** de tiempo. Al subdividir el ambiente en localidades, es posible recuperar la noción de tiempo con respecto a una localidad.

7.3 AMBIENTES CARTAGO

CArtAgO [158] es un marco computacional para la programación de medios ambientes, basado en el meta-modelo de **Agentes y Artefactos** (Ver la Figura 7.6), que busca cumplir con la desiderata introducida en la sección anterior. El ambiente se concibe como un conjunto dinámico de entidades computacionales llamadas **artefactos**, que representan generalmente recursos y herramientas que los agentes trabajando en el mismo ambiente pueden compartir y explotar. Este conjunto de artefactos puede organizarse en uno o multiples **espacios de trabajo**, posiblemente distribuidos en diferentes nodos de una red de cómputo. Un espacio de trabajo representa una localidad. *Agentes y Artefactos* *Artefactos* *Espacios de trabajo*

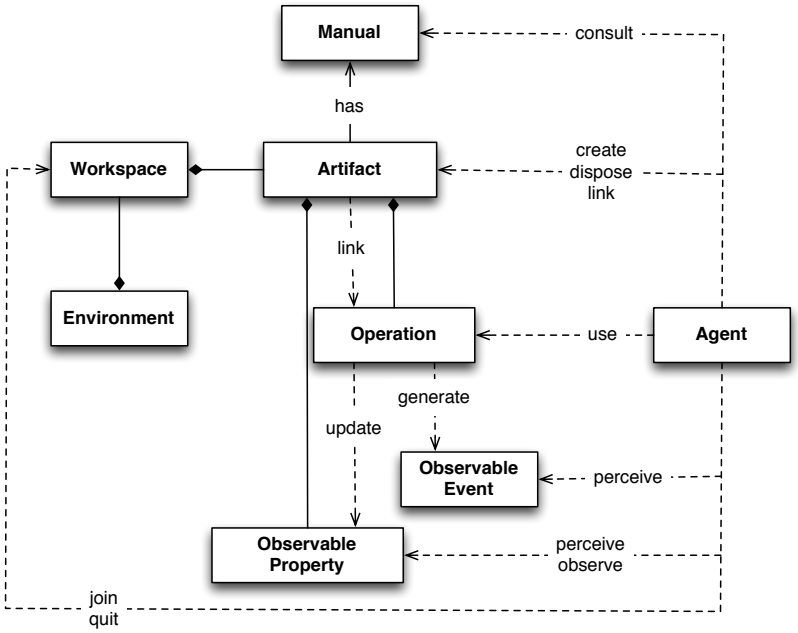

Figura 7.6: El meta-modelo A&A (Agentes y Arfefactos) en notación UML. Adaptado de Ricci, Piunti y Viroli [158], pág., 165.

Abstracción

Desde el punto de vista del diseñador y el programador del SMA, la noción de artefacto es una **abstracción** de primera clase, el módulo básico para estructurar y organizar el ambiente, proveyendo una programación de propósito general y un modelo computacional para dar forma a la funcionalidad disponible para nuestros agentes. De hecho, un programador de SMA define **tipos** de artefactos, análogos a las clases de la programación orientada a objetos, para definir la estructura y comportamiento de los artefactos concretos de cierto tipo. En principio, cada espacio de trabajo dispone de un conjunto dinámico de tipos de artefactos que puede ser usado para crear artefactos. Desde el punto de vista del agente, los artefactos son las entidades de primera clase que estructuran, desde un punto de vista funcional, el mundo computacional donde están situados. Los artefactos pueden ser creados, compartidos, usados y percibidos en tiempo de ejecución.

Tipos de artefacto

Operaciones

Interfaz de uso

Propiedades observables

Señales

Eventos

Para poner su funcionalidad a disposición de los agentes, un artefacto provee un conjunto de operaciones y otro de propiedades y eventos observables (Ver Figura 7.7). Las **operaciones** representan procesos computacionales, posiblemente de largo término, ejecutados dentro de los artefactos; que pueden ser disparados por agentes o por otros artefactos. El término **interfaz de uso** se refiere al conjunto de todas las operaciones de un artefacto que están disponibles para un agente. Las **propiedades observables** representan variables de estado cuyos valores pueden ser percibidos por los agentes que están observando el artefacto. El valor de una propiedad observable puede cambiar dinámicamente como resultado de la ejecución de una operación. La ejecución de una operación puede generar también **señales** o eventos que los agentes pueden percibir. A diferencia de las propiedades observables, las señales pueden usarse para representar **eventos** observables no persistentes ocurridos dentro del artefacto. Además de su estado observable, los

artefactos pueden tener también un estado oculto que quizás sea necesario para implementar la funcionalidad del artefacto.

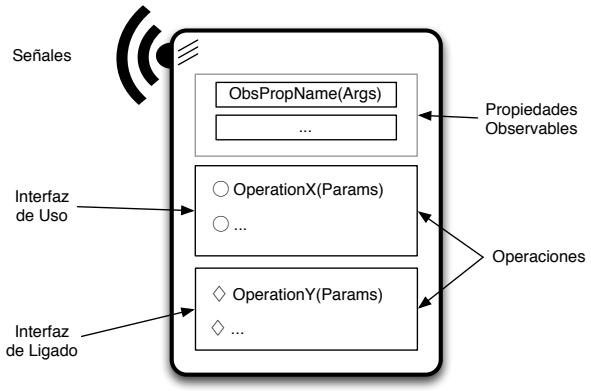

Figura 7.7: Representación abstracta de un artefacto. Adaptado de Ricci, Piunti y Viroli [158], pág., 166.

Desde la perspectiva de un agente, las operaciones de un artefacto representan las **acciones externas** provistas por el ambiente a los agentes. Este es un aspecto central del modelo. De forma que en los ambientes basados en artefactos el repertorio de acciones externas disponible para un agente, aparte de aquellas acciones relacionadas con la comunicación, se define por el conjunto de artefactos en su ambiente. Esto implica que el repertorio de acciones puede ser dinámico dado que el conjunto de artefactos disponible puede ser cambiado dinámicamente por los agentes. Las propiedades observables y los eventos constituyen la percepción de un agente. En los lenguajes BDI como Jason, 2APL o GOAL, las percepciones están relacionadas con el valor de las propiedades observables que pueden modelarse directamente dentro de un agente como **creencias** acerca del estado actual del ambiente. De hecho, para escalar con la complejidad de un ambiente basado en artefactos, un agente puede dinámicamente **focalizar** solo en los artefactos que le interesan.

Acciones externas

Creencias

Focus

Por principio de composición, los artefactos pueden **ligarse** entre si, de forma que un artefacto dispare la ejecución de operaciones de otro artefacto. Con este propósito, los artefactos exhiben una **interfaz de ligado** que análogamente a la interfaz de uso para los agentes, incluye el conjunto de operaciones que puede ser ejecutada por otros artefactos, una vez que dichos artefactos han sido ligados por un agente. La semántica de ejecución para las operaciones ligadas es la misma que la de las operaciones ejecutadas por los agentes: la solicitud de operación ejecutada por el artefacto que está ligando es suspendida hasta que la operación en el artefacto ligado ha sido ejecutada con éxito o fracaso (Como en la Figura 7.1). Las operaciones ligadas no son accesibles a los agentes, solo a artefactos ligados. El ligado permite la realización de ambientes distribuidos al facilitar el ligado de artefactos posiblemente situados en espacios de trabajo diferentes, situados en diferentes nodos de una red de cómputo. De esta manera los artefactos pueden ser tratados como componentes en el contexto de ingeniería del software orientada a componentes, asumiendo que las interfases adecuadas para conectar los artefactos son provistas tal y como son requeridas.

Ligas

Manual

Finalmente, un artefacto puede tener un **manual** en un formato legible por la computadora para ser consultado por los agentes. El manual contendría una descripción de las funcionalidad provista por el artefacto y la manera de explotar dicha funcionalidad. Esta característica ha sido concebida pensando en los Sistemas Abiertos compuestos por agentes inteligentes que dinámicamente deciden que artefactos usar de acuerdo a sus metas. De hecho, la noción de manual puede extenderse de los artefactos a los ambientes de trabajo. En este caso, el manual debe contener la descripción de los protocolos de uso que puede involucrar el uso de multiples tipos de artefactos.

7.4 ACCIONES PARA TRABAJAR CON ARTEFACTOS

Para trabajar en un espacio de trabajo, es necesario que el agente se una a él. Eventualmente, el agente debería salir del espacio de trabajo al completar sus actividades. Un agente puede trabajar simultáneamente en varios espacios de trabajo, posiblemente distribuidos en diferentes nodos de una red de cómputo. Una vez situado en un espacio de trabajo, las acciones disponibles para trabajar con artefactos pueden categorizarse en tres grupos:

- Acciones para crear, buscar y disponer de los artefactos en el espacio de trabajo.
- Acciones para usar artefactos, ejecutar operaciones y observar propiedades y señales.
- Acciones para ligar y desligar artefactos.

En lo que sigue, `Op(Params):Feedback` denota la operación `Op`; con parámetros `Params`; y retroalimentación `Feedback`. La retroalimentación de una operación es algún tipo de dato, que resulta de la ejecución de ésta última, llevando información sobre el éxito o fracaso de tal ejecución. A continuación describiremos las operaciones CArtAgO más comunes en detalle.

7.4.1 Creando y Descubriendo Artefactos

Extensión dinámica

Los artefactos pueden ser creados en tiempo de ejecución por los agentes y una vez creados pueden ser descubiertos por otros agentes. Los agentes también pueden destruir artefactos en su medio ambiente. Este es el mecanismo básico del modelo para dar soporte a la **extensión dinámica** de los ambientes. Tres clases de acciones se proveen con este fin:

- `makeArtifact(Nombre,Tipo,Params):ArtId` crea un nuevo artefacto en el espacio de trabajo actual. La retroalimentación de esta acción es el nombre lógico o **identificador** `ArtId` del artefacto creado.

Identificador

- `disposeArtifact(ArtId)` remueve al artefacto con identificador `ArtId` del espacio de trabajo actual. Se usa un modelo de **Control de Acceso basado en Roles** para evitar que cualquier agente pueda remover cualquier artefacto en el espacio de trabajo. La idea es agrupar a los

Acceso por roles

agentes por roles y para cada rol, especificar políticas que determinan que acciones pueden ejecutarse sobre determinados artefactos.

- `lookup(Nombre):ArtId` busca por `Nombre` un artefacto en el espacio de trabajo regresado como retroalimentación su identificador `ArtId`. Otra posibilidad es `lookupArtifactByType(Tipo):ArtId`, que busca artefactor por `Tipo`. Observen que esta acción regresa un conjunto (posiblemente vacío) de artefactos como retroalimentación.

Tipo

Los tipos de artefacto predefinidos por CArtAgO están en el paquete `cartago.tools`, de forma que si buscamos una consola por tipo de artefacto, deberíamos buscar `cartago.tools.Console`. Otros tipos de artefactos incluídos por default son listados en el Cuadro 7.3.

Barrier	Clock	CommonEventListener	CommunicatorArtifact
Console	EventOpInfo	GUIArtifact	Latch
SocketChannel	TupleSet	TupleSpace	

Cuadro 7.3: Tipos de Artefactos predefinidos en CArtAgO.

7.4.2 Usando y Observando los Artefactos

Usar un artefacto involucra dos aspectos. Primero, el agente debe ser capaz de ejecutar las operaciones listadas en la interfaz de uso del artefacto; Segundo, el agente debe ser capaz de percibir la información observable en términos de propiedades y eventos observables y eventos. Para ejecutar acciones, un agente puede recurrir a la operación `use`:

- `use(ArtId,NombreOp(Params)):ResOp` atiende al primer aspecto mencionado. Esta acción provee la identidad `ArtId` del artefacto que será usado; los detalles de la operación que será ejecutada, incluyendo su nombre `NombreOp` y sus parámetros `Params`. La acción tiene éxito si la operación correspondiente es terminada con éxito; y falla si la operación especificada no está incluída en la interfaz de uso del artefacto; o si un error ocurrió durante la ejecución de la operación, por ejemplo, que la operación misma haya fallado. La ejecución completa de la operación puede generar algún resultado que es devuelto al agente como la retroalimentación `ResOp`. Al ejecutar `use` la actividad o plan actual del agente se suspende hasta que un evento reportando que la operación se ha completado, con éxito o fracaso, es recibida. Al recibir este evento, la ejecución de la acción es completada y el plan o actividad relativo a ésta se reactiva. Sin embargo, aún cuando esta actividad ha sido suspendida, el agente no se bloquea, sigue su ciclo de razonamiento normal.

La semántica anterior resulta en considerar directamente a las operaciones de los artefactos como acciones externas de los agentes, o mejor aún, como una conjunto de operaciones provisto por los artefacto como una extensión al repertorio conductual del agente. Las operaciones ejecutadas por

los artefactos pueden ser procesos de larga duración, por lo que pueden no terminar inmediatamente en éxito o fracaso. Por lo tanto, el modelo de ejecución adoptado es el **orientado a procesos**. Veremos más adelante como esta semántica puede explotarse eficientemente para crear mecanismos eficientes de sincronización entre agentes.

Modelo Orientado a Procesos

El segundo aspecto del uso de artefacto, su observación, es atendido por `focus`:

- `focus(ArtId,Filtro)` especifica la identificación `ArtId` del artefacto a observar y opcionalmente un filtro, para seleccionar el subconjunto de eventos en los que el agente está interesado. En los lenguajes de programación orientados a agentes BDI, las propiedades observables se corresponden directamente con las **creencias** de los agentes. Un agente puede focalizar en diversos artefactos al mismo tiempo.

Creencias

- `stopFocus(ArtId)` es el dual de `focus`. Provee el mecanismo para dejar de observar al artefacto con identificador `ArtId`.

Observen que el modelo de percepción adoptado es **basado en eventos**. Cada vez que una propiedad observable de un artefacto cambia, o una señal es generada por dicho artefacto, eventos observables relacionados con estos cambios son enviados a todos los agentes que están atentos al artefacto en cuestión. En los lenguajes BDI, el primer tipo de eventos, cambios en las propiedades observables, hacen posible, del lado de la arquitectura del agente, actualizar automáticamente la actualización de creencias, manteniendo un registro de las propiedades observables. Las señales en cambio no están relacionadas con las propiedades observables, se les puede considerar como mensajes generados por los artefactos que son procesadas asíncronamente del lado de los agentes.

Percepción basada en Eventos

Es importante mencionar que un agente puede usar un artefacto sin necesidad de estar observándolo. De igual forma, si un agente ejecuta una operación de un artefacto que está siendo observado por otro agente, el agente observador tendrá eventualmente percepciones acerca de la ejecución de la operación (mientras la ejecución de `use` no se haya terminado). Una última instrucción en esta categoría es la siguiente:

- `observeProperty(Prop):ValorProp` lee el valor actual de la propiedad `Prop`. En este caso no hay eventos generados, el valor `ValorProp` regresa al agente como una señal de retroalimentación de la acción. Esto resulta útil cuando no es necesario que el agente este atento continuamente del estado observable de un artefacto; pero necesita saber el valor de la propiedad en un momento dado.

7.4.3 Ligando Artefactos

Ligar dos artefactos sirve para conectarlos de forma que el artefacto que liga puede ejecutar operaciones sobre el artefacto ligado. De manera más precisa, cuando ligamos dos artefactos, la ejecución de una operación en el artefacto que liga puede disparar la ejecución de operaciones en el instrumento ligado. Dos acciones son provistas con este fin:

- `linkArtifacts(Art1,Art2,Puerto)` Denota que el artefacto `Art1` liga `Art2`. El parámetro `Puerto` es necesario cuando se liga el mismo artefacto a multiples artefactos. En ese caso, el artefacto `Art1` debe proveer varios puertos etiquetados y adjuntar cada artefacto ligado a un puerto en particular.
- `unlinkArtifacts(Art1,Art2)` es el dual de la operación anterior.

Una nota final sobre la compatibilidad en el ligado de artefactos. Actualmente, se ha adoptado la solución más simple, permitiendo que cualquier artefacto pueda ser ligado por otros artefactos. Esto sin embargo, puede producir errores en tiempo de ejecución. Por ejemplo, en el caso en que el cuerpo de la operación de un artefacto una operación de ligado es ejecutada y 1) ningún artefacto está ligado en ese momento; ó 2) La operación ejecutada no es provista por el artefacto ligado. Es necesario introducir en algún momento una semántica formal de las operaciones de ligado.

7.5 CARTAGO Y JASON

Veamos como se integran Jason y CArtAgO a partir de una serie de ejemplos.

7.5.1 Hola Mundo

Comenzaremos con el ejemplo clásico de Hola Mundo. A continuación se muestra la definición del SMA para este caso:

```
MAS ej00 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    agente agentArchClass jaca.CAgentArch;

  aslSourcePath:
    "src/asl";
}
```

Aquí las novedades tienen que ver con que el ambiente en el que situaremos a nuestro agente será un ambiente CArtAgO, como se indica en el argumento de `environment: c4jason.CartagoEnvironment`. Observen que `agente` tiene como clase de arquitectura `c4jason.CAgentArch`, para explotar al ambiente. La definición de `agente` es muy simple:

```
// Agent agente in project cartagoHolaMundo.mas2j

/* Initial goals */

!inicio.

/* Plans */

+!inicio : true <- println("Hola mundo.").
```

Cuando un agente es creado, se une automáticamente al espacio de trabajo llamado `default` en el nodo actual donde se está ejecutando el ambiente. Esto se puede modificar, como veremos más adelante, mediante los parámetros de la declaración del ambiente. La operación `println` es provista por el artefacto `console` que está disponible automáticamente en el espacio de trabajo `default`, de forma que la acción externa `println` del `agente` es mapeada a la operación correspondiente de ese artefacto. Este es el caso de ejecución de operación se lleva a cabo sin especificar el artefacto específico que desea usarse. La salida en consola es la siguiente:

```
[agente] Hola mundo.
```

7.5.2 Definición, Creación y Uso de Artefactos

El siguiente ejemplo ilustra los aspectos básicos de la creación y uso de artefactos, incluída su observación. Dos agentes son creados de forma que observan y comparten un artefacto que implementa un `Contador`. El agente `usuario` realiza operaciones en el artefacto, mientras que el agente `observador` percibe el estado observable y las señales del mismo. La definición del SMA es como sigue:

```
MAS cartago01 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    usuario agentArchClass jaca.CAgentArch;
    observador agentArchClass jaca.CAgentArch;

  aslSourcePath:
    "src/asl";
}
```

Veamos primero al agente `usuario` cuya meta `inicio` comprende usar y crear un artefacto identificado como `contador00` de tipo `c4j.Contador` para usarlo dos veces ejecutando la operación `inc`:

```
// Agent usuario in project cartagoUsoArtefactos.mas2j

/* Initial goals */
!inicio.

/* Plans */
+!inicio
  <-  makeArtifact("contador00","tools.Contador",[],Id);
      inc;
      inc[artifact_id(Id)];
      inc[artifact_name("contador00")].
```

Los artefactos se crean con la acción externa `makeArtifact` que recibe como argumentos una cadena para el nombre el artefacto, `contador00`; otra que representa la clase que implementa el artefacto, `c4j.Contador`; una lista de parámetros, en este caso vacía; y una variable que unifica con el identificador lógico del artefacto, `Id`.

Hay tres formas de ejecutar una operación. En la primera, la operación `inc` se invoca sin especificar que artefacto deseamos usar. Si ningún artefacto provee esa operación, ésta falla y en consecuencia, la acción externa del agente falla también. Si se se encuentra más de un artefacto que provee la operación en cuestión, se consideran primero los artefactos creados por el agente y se selecciona uno no deterministicamente. En la segunda forma, especificamos que artefacto deseamos usar mediante su identificador lógico `Id`. En la tercera, usamos el nombre del artefacto `contador00` en lugar de su identificador lógico.

El `Contador` es un tipo de artefacto de una sola propiedad observable, denotada como `valor` inicializada como cero en el método `init`. Esta propiedad observable es actualizada por la operación `inc`. La ejecución de esta operación genera una señal `tick`. Su implementación en Java es como sigue:

```
1  package tools;
2
3  import cartago.*;
4
5  public class Contador extends Artifact {
6
7    void init(){
8      defineObsProperty("valor",0);
9    }
10
11   @OPERATION void inc(){
12     ObsProperty prop = getObsProperty("valor");
13     prop.updateValue(prop.intValue()+1);
14     signal("tick");
15   }
16 }
```

Todo tipo de artefacto extiende la clase predefinida `Artifact`. El método `init` representa el constructor del artefacto, que inicializa un objeto de la clase tan pronto como este es creado. Si `init` requiere parámetros, estos toman valores de la llamada a `makeArtifact` (en este caso, la lista de parámetros está vacía). Las operaciones se definen como métodos de tipo `void` anotados con `@OPERATION`. Los parámetros de estos métodos se corresponden con los parámetros de las operaciones.

Las propiedades observables de un artefacto, como `valor`, se definen mediante la primitiva `defineObsProp`. En su forma más general, una propiedad observable toma la forma de una tupla con un functor seguido de uno o más argumentos de cualquier tipo. Para acceder a una propiedad observable, se usa la primitiva `getObsProperty` especificando el nombre de la propiedad en cuestión. El método `updateValue` puede usarse para modificar el valor de una propiedad observable.

Al igual que las propiedades observables, las señales tienen forma de tupla, con un functor seguido de uno o multipes argumentos opcionales. En el caso de `tick`, la señal no tiene argumentos. La primitiva `signal` tiene dos variantes:

- `signal(String nombreSeñal, Object... params)` genera una señal que todos los agentes que observan al artefacto pueden percibir.

- `signal(AgentId id, Srting nombreSeñal, Object... params)` genera una señal que el agente *id* puede percibir si está folcalizado en el artefacto.

Transacciones atómicas

Las operaciones se ejecutan **transaccionalmente** con respecto al estado observable del artefacto. De forma que no pueden ocurrir interferencias cuando diversos agentes usan el artefacto de manera concurrente, ya que las operaciones son **atómicas**. Los cambios en las propiedades observables de un artefacto son observables solo cuando:

- La operación se completa con éxito;
- Una señal es generada;
- La operación es suspendida (por medio de la operación `await`, descrita más adelante).

Si la operación falla, los cambios en el estado observable del artefacto se revierten. Finalmente, el agente `observador` focaliza en el contador creado por `usuario` e imprime mensajes en la pantalla cada vez que la propiedad observable `valor` es modificada o una señal `tick` es percibida:

```
1  // Agent observador in project cartagoUsoArtefactos.mas2j
2
3  /* Initial goals */
4
5  !observa.
6
7  /* Plans */
8  +!observa : true
9    <-  focusWhenAvailable("contador00").
10
11 +valor(V)
12   <-  println("Nuevo valor observado: ",V).
13
14 +tick
15   <-  println("Percibiendo un tick.").
```

Los agentes pueden descubrir el identificador lógico de un artefacto mediante la acción `lookupArtifact` provista por el espacio de trabajo donde se encuentra el artefacto en cuestión; especificando ya sea el nombre del artefacto o su tipo. En el último caso, si existen varios artefactos del mismo tipo, uno de ellos es elegido no deterministicamente. En el ejemplo, si el agente ejecuta la acción `lookupArtifact` antes de que el artefacto haya sido creado, entonces la acción falla y la intención falla generando el evento Jason `-?miArtefacto(...)` invocando al plan de reparación correspondiente.

Los agentes pueden focalizar en diferentes artefactos del ambiente. La acción `focus` tiene dos variantes:

- `focus(ArtifactId id, IEventFilter filtro)` especifica un filtro para seleccionar las percepciones recibidas.
- `focusWhenAvailable(String nombreArtefacto)` focaliza en el artefacto especificado tan pronto como éste está disponible en el espacio de trabajo.

Al focalizar en un artefacto, las propiedades observables del artefacto se mapean en la base de creencias del agente. De esta forma, los cambios en las propiedades observables del artefacto son detectados como cambios en la base de creencias del agente. En este ejemplo, cada que la propiedad observable `valor` es modificada, el agente `observador` detecta un evento disparador `+valor(V)`. Las creencias relacionadas con propiedades observables de los artefactos están decoradas con anotaciones que pueden ser usadas al momento de seleccionar planes relevantes y aplicables. En particular:

- `source(percept), percept_type(obs_prop)` definen el tipo de percepción en cuestión.
- `artifact_id(Id), artifact_name(Id,Nombre), artifact_type(Id, Tipo)` y `workspace(Id,nombreWS)` proveen información sobre el artefacto fuente y su espacio de trabajo. Es importante observar que al tratarse de creencias, estos valores pueden accederse mediante consultas a la base de creencias, por ejemplo `?workspace(contador00,NombreWS)` debería unificar el nombre del espacio de trabajo donde esta situado el contador `contador00` con la variable `NombreWS`.
- Dado el modelo de creencias de Jason (recuerden la diferencia con Prolog, Jason solo guarda una vez hechos repetidos), las creencias y por tanto, las propiedades observables, con el mismo functor y los mismos argumentos, colpasan a una sola creencia con las anotaciones mezcladas.

Cuando un agente focaliza en un artefacto, las señales emitidas por este último se vuelven perceptibles para el agente. Al igual que con las propiedades observables, esto se ve reflejado en la base de creencias de los agentes. En el ejemplo un evento `+tick` se genera en el agente `observador` cada que `usuario` ejecuta la operación `inc`; aunque en este caso la base de creencias no cambia. Al igual que en el caso de las propiedades observables, las anotaciones pueden usarse en la deliberación del agente. Particularmente:

- `source(Id), percept_type(obs_ev)` definen este tipo de percepción en particular.
- `artifact_id(Id), artifact_name(Id,Nombre), artifact_type(Id, Tipo)` y `workspace(Id,NombreWS)` proveen información sobre el artefacto fuente de la señal y su espacio de trabajo.

La salida en consola de este ejemplo suele ser:

```
[observador] Nuevo valor observado: 3
```

Aunque en ocasiones puede incluir el reporte de `observador` indicando que un `tick` o una actualización de valor se han percibido. Eso depende de a partir de que momento este agente ha focalizado en el artefacto.

7.5.3 Fallo en las Acciones

Este ejemplo, una simple variación del anterior, ilustra el caso de fallo en las acciones de un agente. La definición del SMA es como sigue:

```
MAS cartago02 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    usuario agentArchClass jaca.CAgentArch;
    observador agentArchClass jaca.CAgentArch;

  aslSourcePath:
    "src/asl";
}
```

El SMA consta de dos agentes que crean, usan y observan un artefacto compartidod de tipo `ContadorAcotado`. Este tipo de artefacto se implementa como sigue:

```
package tools;

import cartago.*;

public class ContadorAcotado extends Artifact {
  private int max;

  void init(int max){
    defineObsProperty("valor",0);
    this.max = max;
  }

  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("valor");
    if (prop.intValue() < max) {
      prop.updateValue(prop.intValue()+1);
      signal("tic");
    } else {
      failed("La operación inc falló","inc_fallo","max_alcanzado",max);
    }
  }
}
```

A diferencia del ejemplo anterior, en este caso la operación `inc`, y por tanto la acción externa `inc` del agente, fallan si el contador ha alcanzado su máximo valor, especificado como un parámetro del constructor `init`. Para especificar el fallo de una operación empleamos la primitiva `failed`, que viene en dos sabores:

- `failed(String mensajeFallo)`
- `failed(String mensajeFallo, String descr, Object... args)`

En el ejemplo usamos el segundo caso de `failed`. Esto ocasiona que cuando un plan que invoca a `inc` falla por que la acción falló, el evento asociado incluya en sus anotaciones:

```
[error_msg(Msg),env_failure_reason(inc_fallo("max_alcanzado",Val))]
```

donde, para este caso, `Msg` unifica con "La operación inc falló" y `Val` con 50. El agente `usuario` hace uso de estas anotaciones y se define como sigue:

```
// Agent usuario in project cartagoErrorAccion.ma2j

/* Initial goals */
!creaUsaCont.

/* Plans */
+!creaUsaCont : true
  <-  !crea(C);
      !usa(C).

+!usa(C)
  <-  for (.range(I,1,100)) {
        inc[artifact_id(C)];
      }.

-!usa(C)[error_msg(Msg),env_failure_reason(inc_fallo("max_alcanzado",Val))]
  <-  println(Msg);
      println("Ultimo valor fue ",Val).

+!crea(C): true
  <- makeArtifact("contador00","tools.ContadorAcotado",[50],C).
```

Observen que el valor máximo del contador acotado es 50 por el parámetro que se le pasa a `makeArtifact`. Es importante también introducir un plan de recuperación para contender con los fallos de `+!usa`. El código del agente `observador` es como sigue:

```
// Agent observador in project cartagoErrorAccion.mas2j

/* Initial goals */
!observa.

/* Plans */
+!observa : true
  <- ?contador(C);
     focus(C).

+valor(V)[artifact_name(Id,"contador00")]
  <- println("Un nuevo valor observado para ", Id, ": ",V).

+tic[artifact_name(Id,"contador00")]
  <- println("Tic percibido").

+?contador(Id): true
  <- lookupArtifact("contador00",Id).

-?contador(Id): true
  <- .wait(10);
     ?contador(Id).
```

La salida en consola de este SMA es algo como:

```
...
[observador] Tic percibido
[observador] Un nuevo valor observado para contador00: 12
[usuario] La operación inc falló
...
[observador] Un nuevo valor observado para contador00: 19
[usuario] Ultimo valor fue 50
[observador] Tic percibido
[observador] Un nuevo valor observado para contador00: 20
```

```
10 |  ...
```

7.5.4 Operaciones con Salida y Acciones con Retroalimentación

Las operaciones pueden tener parámetros de salida, es decir valores que la ejecución de la operación computa. Del lado del agente estos parámetros se manejan como retroalimentación de las acciones. A nivel API, los parámetros de salida se representan con la clase `OpWithFeedbackParam<ParamType>`, donde `ParamType` debe ser el tipo específico del parámetro de salida. La clase provee un método `set` para dar valor al parámetro de salida. En el siguiente ejemplo, un agente crea un artefacto de tipo `Calculadora` para ejecutar operaciones con sus parámetros. El SMA es como sigue:

```
MAS cartago03 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    usuario agentArchClass jaca.CAgentArch;
  aslSourcePath:
    "src/asl";
}
```

El tipo de artefacto `Calculadora` se define como sigue:

```
package tools;

import cartago.*;

public class Calculadora extends Artifact {

  @OPERATION
  void suma(double a, double b, OpFeedbackParam<Double> suma) {
    suma.set(a+b);
  }

  @OPERATION
  void resta(double a, double b, OpFeedbackParam<Double> resta) {
    resta.set(a-b);
  }
}
```

y el agente `usuario` es:

```
// Agent usuario in project cartagoSalidaOps.mas2j

/* Initial goals */
!usaCalculadora.

/* Plans */
+!usaCalculadora
  <-  makeArtifact("miCalculadora","tools.Calculadora",[],_);
    suma(4,5,Suma);
    println("La suma de 4 y 5 es ",Suma);
    resta(4.0,5.0,Resta);
    println("La resta de 4 y 5 es ", Resta).
```

Observen que del lado del agente, los parámetros de salida de una acción se capturan en variables que están acotadas por el valor computado al ejecutar la operación correspondiente. Finalmente, una operación puede tener cualquier número de parámetros de salida. La salida de este SMA es como sigue:

```
[usuario] La suma de 4 y 5 es 9
[usuario] La resta de 4 y 5 es -1
```

7.5.5 Operaciones con Guardias

Cuando definimos una operación, es posible especificar las condiciones que deben verificarse para iniciar su ejecución. Estas condiciones se conocen como **guardias**. Si la guardia no se verifica, la ejecución de la operación se suspende. La forma de agregar una guardia a una operación es especificando el nombre de un método booleando (la guardia) anotado con `@GUARD`, en la anotación `@OPERATION`.

Los métodos de guardia son invocados con los mismos parámetros de la operación de guardan, por lo que deben declarar esos mismos parámetros. Normalmente, los métodos de guardia verifican los valores internos y el estado observable de los artefactos sin modificarlos.

Las operaciones con guardia son útiles para implementar artefactos de sincronización. En el siguiente ejemplo, las guardias se usan para implementar un artefato de tipo `BuferAcotado` en una arquitectura de productores-consumidores. El SMA es como sigue:

```
MAS cartago04 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    productor agentArchClass jaca.CAgentArch;
    consumidor agentArchClass jaca.CAgentArch;

  aslSourcePath:
    "src/asl";
}
```

El tipo de artefacto `BuferAcotado` se define como sigue:

```
package tools;

import cartago.*;
import java.util.*;

public class BuferAcotado extends Artifact {

  private LinkedList<Object> elems;
  private int nmax;

  void init(int nmax) {
    elems = new LinkedList<Object>();
    defineObsProperty("nElems",0);
    this.nmax = nmax;
```

```
  }

  @OPERATION(guard="buferNoLleno")
  void poner(Object obj) {
    elems.add(obj);
    getObsProperty("nElems").updateValue(elems.size());
  }

  @OPERATION(guard="elemDisponible")
  void obtener(OpFeedbackParam<Object> res) {
    Object elem = elems.removeFirst();
    res.set(elem);
    getObsProperty("nElems").updateValue(elems.size());
  }

  @GUARD
  boolean elemDisponible(OpFeedbackParam<Object> res){
    return elems.size() > 0;
  }

  @GUARD
  boolean buferNoLleno(Object obj){
    return elems.size() < nmax;
  }
}
```

El código del agente `productor` es:

```
// Agent productor in project cartago04.mas2j

/* Initial bels */
elemAProducir(0).

/* Initial goals */
!producir.

/* Plans */
+!producir
  <-  !crear(Bufer);
    !producirElems.

+!producirElems
  <-  ?proxElemAProducir(E);
    poner(E);
    !!producirElems.

+?proxElemAProducir(N)
  <-  -elemAProducir(N);
    +elemAProducir(N+1).

+!crear(B)
  <-  makeArtifact("miBufer","tools.BuferAcotado",[1],B).

-!crear(B)
  <-  lookupArtifact("miBufer",B).
```

Y el del agente `consumidor` es:

```
// Agent consumidor in project cartago04.mas2j


/* Initial goals */
```

```
!consumir.

/* Plans */
+!consumir
  <-  ?buferListo;
    !consumirElems.

+!consumirElems
  <-  obtener(Elem);
    !consumirElem(Elem);
    !!consumirElems.

+!consumirElem(E)
  <-  .my_name(Me);
    println(Me,": ",E).

+?buferListo
  <-  lookupArtifact("miBufer",_).

-?buferListo
  <-  .wait(50);
    ?buferListo.
```

La salida del SMA en consola es:

```
[consumidor] consumidor: 0
[consumidor] consumidor: 1
[consumidor] consumidor: 2
[consumidor] consumidor: 3
[consumidor] consumidor: 4
[consumidor] consumidor: 5
...
```

Observen que cuando un agente ejecuta una operación con guardia, cuya guardia es falsa, suspende su ejecución hasta que la guardia evalua a *true*. La exclusión mutua y la atomicidad se refuerzan: Una operación con guarda suspendida se reactiva y ejecuta únicamente si (cuando) no hay operaciones en ejecución.

7.5.6 Operaciones estructuradas

CArtAgO provee una familia de primitivas llamadas `await` para suspender la ejecución de una operación hasta que cierta condición se satisface. Esto es de utilidad al ejecutar operaciones complejas pues permite romper la ejecución de una operación en múltiples pasos transaccionales. Al suspender la ejecución de una operación, otras operaciones pueden ser invocadas antes de que la operación actual concluya. Cuando la condición especificada se satisface y no hay operaciones en ejecución, la operación suspendida es retomada. Las operaciones complejas que pueden emplearse usando este mecanismo, incluyen:

- Operaciones de largo término que necesitan bloquear el uso del artefacto;
- Operaciones concurrentes cuya ejecución se traslapan, que son escenciales para la realización de mecanismos de coordinación.

En el siguiente ejemplo, dos agentes comparten y usan concurrentemente un artefacto que provee una operación usando este mecanismo. El SMA es como sigue:

```
MAS cartago05 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    usuario1 agentArchClass jaca.CAgentArch;
    usuario2 agentArchClass jaca.CAgentArch;


  aslSourcePath:
    "src/asl";
}
```

El artefacto usado por estos agentes se define como sigue:

```
package tools;

import cartago.*;

public class ArtefactoComplejo extends Artifact {
  int ContadorInterno;

  void init() {
    ContadorInterno = 0;
  }

  @OPERATION void operacionCompleja(int nVeces){
    trabaja();
    signal("paso1Completado",ContadorInterno);
    await("miCondicion",nVeces);
    signal("paso2Completado",ContadorInterno);
  }

  @GUARD boolean miCondicion(int nVeces) {
    return ContadorInterno >= nVeces;
  }

  @OPERATION void actualiza(int delta) {
    ContadorInterno += delta;
  }

  private void trabaja(){}
}
```

En `operacionCompleja` se lleva a cabo una tarea `trabaja`, entonces se genera la señal `paso1Completado`; luego, mediante `await`, se suspende la ejecución de la operación hasta que la condición definida por la guardia `miCondicion` cuyo nombre y parámetros (de ser necesarios) se especifican como parámetros de `await`, se cumple.

Además de `operacionCompleja`, la operación `actualiza` provee un medio para incrementar el `ContadorInterno`. El agente `usuario1` ejecuta la `operacionCompleja`; y el agente `usuario2` ejecuta de manera repetida la operación `actualiza`. La acción, y por tanto el plan, del primer agente es sus-

pendido hasta que el segundo agente ha ejecutado `actualiza` el número de veces suficientes para que la ejecución de `operacionCompleja` sea retomada. El agente `usuario1` se implementa como sigue:

```
/* Initial goals */
!prueba.

/* Plans */

@prueba
+!prueba
  <-  println("Creando el artefacto...");
      makeArtifact("a0","tools.ArtefactoComplejo",[],Id);
    focus(Id);
    println("Ejecutando la acción compleja... ");
    operacionCompleja(5);
    println("Acción completada.").

+paso1Completado(C)
  <-  println("Primer paso completado. Contador = ",C).

+paso2Completado(C)
  <-  println("Segundo paso completado. Contador = ",C).
```

Observen que este agente reacciona a la señal `paso1Completado` generada por el artefacto, imprimiendo una mensaje en consola; Aún si la ejecución del plan `prueba` es suspendida a la espera de `operacionCompleja(5)`. El agente `usuario2` se implementa como sigue:

```
/* Initial goals */
!prueba.

/* Plans */

+!prueba
  <-  !descubreArtefacto("a0");
      !usaArtefacto(10).

+!usaArtefacto(N) : N>0
  <-  actualiza(1);
      println("Artefacto actualizado.");
      !usaArtefacto(N-1).

+!usaArtefacto(0)
  <-  println("Uso del artefacto completado").

+!descrubreArtefacto(NombreArt)
  <-  lookupArtifact(NombreArt,_).

-!descubreArtefacto(NombreArt)
  <-  .wait(10);
      !!descubreArtefacto(NombreArt).
```

Observen que aún cuando las operaciones de los agentes se traslapan, solo una operación está en ejecución a un tiempo dado, de forma que no hay interferencias al acceder y modificar el estado del artefacto. Por otra parte, al ejecutar `await`, todos los cambios en las propiedades observables hechos hasta ese momento están comprometidos. La salida del SMA es como sigue:

```
  [usuario1] Creando el artefacto...
```

```
  [usuario1] Ejecutando la acción compleja...
  [usuario1] Primer paso completado. Contador = 0
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario1] Segundo paso completado. Contador = 6
  [usuario1] Acción completada.
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario2] Artefacto actualizado.
  [usuario2] Uso del artefacto completado
```

7.5.7 Artefactos de coordinación

A continuación incluiremos un ejemplo sobre como explotar las operaciones estructuradas para implementar artefactos de coordinación. La idea es resolver un problema de coordinación conocido como la **cena de los filósofos**, propuesto en su forma actual por Hoare [100], mediante un **pizarrón** (*blackboard*). Las operaciones `in` y `rd` son facilmente implementadas usando la primitiva `await`. Recuerden que el artefacto `blackboard` es provisto por el espacio de trabajo `default`. El SMA se compone de un mesero y cinco filósofos. El mesero es el responsable de preparar el ambiente, creando las tuplas que representan los tenedores. Esto es, 5 tuplas `tenedor(F)` y boletos, 5 tuplas `boleto`, que permiten evitar los puntos muertos. El SMA se define como sigue:

```
MAS cartago06 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    mesero agentArchClass jaca.CAgentArch;
    filosofo agentArchClass jaca.CAgentArch #5;

  aslSourcePath:
    "src/asl";
}
```

El código del mesero es como sigue:

```
// Agent mesero in project cartago06.mas2j

/* Initial beliefs and rules */

filosofo(0,"filosofo1",0,1).
filosofo(1,"filosofo2",1,2).
filosofo(2,"filosofo3",2,3).
filosofo(3,"filosofo4",3,4).
filosofo(4,"filosofo5",4,0).

/* Initial goals */
```

```
12
13 !prepararMesa.
14
15 /* Plans */
16
17 +!prepararMesa
18   <-  for(.range(I,0,4)) {
19       out("tenedor",I);
20       ?filosofo(I,Nombre,Izq,Der);
21       out("filosofoIni",Nombre,Izq,Der);
22     };
23     for(.range(I,1,4)) {
24       out("boleto");
25     };
26     println("Hecho.").
```

Los filósofos repetidamente se hacen de un par de tenedores, los usan para comer y los liberan. Antes de tomar los tenedores, deben hacerse de un boleto, que es liberado después de liberar los tenedores. He aquí su código:

```
1  // Agent filosofo in project cartago06.mas2j
2
3  /* Initial beliefs and rules */
4
5  /* Initial goals */
6
7  !start.
8
9  /* Plans */
10
11 +!start <-
12   .my_name(Yo);
13   in("filosofoIni",Yo,Izq,Der);
14   +miTenedorIzq(Izq);
15   +miTenedorDer(Der);
16   println(Yo, " listo.");
17   !!viviendo.
18
19 +!viviendo <-
20   !pensando;
21   !comiendo;
22     !viviendo.
23
24 +!comiendo <-
25   !adquirirRecursos;
26   !comer;
27     !liberarRecursos.
28
29 +!adquirirRecursos: miTenedorIzq(TIzq) & miTenedorDer(TDer) <-
30   in("boleto");
31   in("tenedor",TIzq);
32     in("tenedor",TDer).
33
34 +!liberarRecursos: miTenedorIzq(TIzq) & miTenedorDer(TDer) <-
35   out("tenedor",TIzq);
36   out("tenedor",TDer);
37     out("boleto").
38
39 +!pensando <-
40   println("Pensando.").
```

```
41
42 +!comer <-
43   println("Comiendo.").
```

Observen que no es necesario crear un artefacto `blackboard` porque este ya es provisto por el espacio de trabajo `default`. La salida del SMA es algo como:

```
1   [filosofo1] filosofo1 listo.
2   [filosofo2] filosofo2 listo.
3   [filosofo1] filosofo1 pensando.
4   [filosofo3] filosofo3 listo.
5   [filosofo4] filosofo4 listo.
6   [filosofo2] filosofo2 pensando.
7   [filosofo5] filosofo5 listo.
8   [filosofo1] filosofo1 comiendo.
9   [mesero] Hecho.
10  [filosofo3] filosofo3 pensando.
11  [filosofo4] filosofo4 pensando.
12  [filosofo5] filosofo5 pensando.
13  [filosofo3] filosofo3 comiendo.
14  [filosofo2] filosofo2 comiendo.
15  ...
```

7.5.8 Una Interfaz gráfica como artefacto

Un ejemplo interesante de artefactos que encapsulan operaciones de entrada y salida son los artefactos GUI (*Graphic User Interface*), esto es, artefactos funcionando como componentes de una Interfaz Gráfica de Usuario. Esto permite la interacción entre humanos usuarios y sus agentes. Estos artefactos usan `Swing` para definir la estructura de la interfaz gráfica; y entonces permiten, por una parte, definir operaciones que se corresponden con las acciones de la GUI, de tal forma que pueden manejar eventos especificos de la interfaz. Tales operaciones generan señales o cambian los eventos observables para disparar a los agentes que observan la GUI. Por otra parte, proveen operaciones que pueden ser usadas posiblemente por los agentes para cambiar la GUI. En el siguiente ejemplo, el agente `agente_gui` crea y usa un artefacto como los descritos para interactuar con su usuario. El SMA es como sigue:

```
1  MAS cartago07 {
2
3    infrastructure: Centralised
4
5    environment: jaca.CartagoEnvironment
6
7    agents:
8      usuario agentArchClass jaca.CAgentArch;
9
10   aslSourcePath:
11     "src/asl";
12 }
```

Para facilitar el desarrollo de artefactos GUI, CArtAgO provee la implementación de un artefacto GUI de base `cartago.tools.GUIArtifact`. Este artefacto explota la operación `await` y otros comandos de bloqueo. La idea

es que proveea al programador de facilidades básicas para ligar eventos GUI y operaciones del artefacto. Crearemos un artefacto `MiGUI` que crea una intefaz con campo de texto y un botón. Algunos eventos GUI –presionar el botón, teclear en el campo de texto, cerrar la ventana– están ligados a algunas de las operaciones internas del artefacto, que a su vez, generan eventos observables para los agentes en el SMA. El artefacto se implementa como sigue:

```
1  package tools;
2
3  import cartago.*;
4  import cartago.tools.*;
5  import javax.swing.*;
6  import java.awt.event.*;
7
8  public class Gui extends GUIArtifact {
9
10   private MiMarco marco;
11
12   public void setup() {
13     marco = new MiMarco();
14     linkActionEventToOp(marco.botonOk,"ok");
15     linkKeyStrokeToOp(marco.texto,"ENTER","textoModificado");
16     linkWindowClosingEventToOp(marco,"cerrado");
17
18     defineObsProperty("valor",obtenerValor());
19     marco.setVisible(true);
20   }
21
22   @INTERNAL_OPERATION void ok(ActionEvent ev) {
23     signal("ok");
24   }
25
26   @INTERNAL_OPERATION void cerrado(WindowEvent ev) {
27     signal("cerrado");
28   }
29
30   @INTERNAL_OPERATION void textoModificado(ActionEvent ev){
31     getObsProperty("valor").updateValue(obtenerValor());
32   }
33
34   @INTERNAL_OPERATION void asignarValor(int valor){
35     marco.asignarTexto(""+valor);
36     getObsProperty("valor").updateValue(obtenerValor());
37   }
38
39   private int obtenerValor(){
40     return Integer.parseInt(marco.obtenerTexto());
41   }
42
43   class MiMarco extends JFrame {
44
45   private static final long serialVersionUID = 1L;
46
47   private JButton botonOk;
48     private JTextField texto;
49
50     public MiMarco(){
51       setTitle("Artefacto como GUI");
```

```
52        setSize(200,100);
53        JPanel panel = new JPanel();
54        setContentPane(panel);
55        botonOk = new JButton("incrementa");
56        botonOk.setSize(80,50);
57        texto = new JTextField(10);
58        texto.setText("0");
59        texto.setEditable(true);
60        panel.add(texto);
61        panel.add(botonOk);
62      }
63
64      public String obtenerTexto() {
65        return texto.getText();
66      }
67
68      public void asignarTexto(String s) {
69        texto.setText(s);
70      }
71    }
72  }
```

El agente que hace uso de este artefacto se define como sigue:

```
1  // Agent gui in project cartago07.mas2j
2
3  /* Initial goals */
4  !prueba.
5
6  /* Plans */
7
8  +!prueba2 <- // Test with the REPL agent
9    asignarValor(1000).
10
11 +!prueba <-
12   makeArtifact("gui","tools.Gui",[],Id);
13     focus(Id).
14
15 +valor(V) <-
16   println("Valor actualizado: ",V).
17
18 +ok : valor(V) <-
19   asignarValor(V+1).
20
21 +cerrado <-
22   .my_name(Yo);
23      println("Adios");
24      .kill_agent(Yo).
```

La meta `!prueba2` (comentada el código) ilustra la comunicación en el otro sentido, el agente ejecuta `asignaValor(1000)` y esto se ve reflejando en el campo de texto de la interfaz gráfica. La interacción entre este agente y su artefacto puede verse en la figura 7.8.

7.5.9 Operaciones de Ligado

El mecanismo de ligado permite crear interacciones entre artefactos mediante la ejecución de sus operaciones. Además de su intefaz de usuario, un artefacto puede exponer operaciones etiquetadas como `@LINK`. Estas opera-

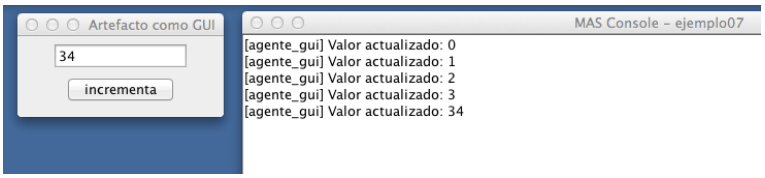

Figura 7.8: El agente interfaz gráfica y su artefacto.

ciones están concebidas para ser llamadas por otros artefactos. Para permitir que un artefacto A ejecutar operaciones de otro artefacto B se proveen dos opciones:

- El artefacto A debe ser ligado explícitamente al artefacto B por un agente, ejecutando a acción `linkArtifacts`. Un argumento de esta acción debe especificar el **puerto** de salida `output` que el artefacto A expone. Entonces, las operaciones del artefacto A pueden ejecutar operaciones de ligado del artefacto B usando la primitiva `execLinkedOp` con la especificación del puerto de salida `output` donde el artefacto ligado ha sido agregado.
- Son ligar dos artefactos, un artefacto A puede ejecutar operaciones sobre un artefacto B especificando en `execLinkedOp` que B es el artefacto afectado, mediante su identificador.

En el siguiente ejemplo, un agente crea y liga dos artefactos. Luego ejecuta una operación en el artefacto que liga, para observar como se ejecutan operaciones en el aparato ligado. El SMA es como sigue:

```
1  MAS cartago08 {
2
3    infrastructure: Centralised
4
5    environment: jaca.CartagoEnvironment
6
7    agents:
8      agente agentArchClass jaca.CAgentArch;
9
10   aslSourcePath:
11     "src/asl";
12 }
```

El artefacto a ligar es un contador que se define así:

```
1  package tools;
2  import cartago.*;
3
4  public class ArtefactoLigable extends Artifact {
5
6    int contador;
7
8    void init(){
9      contador = 0;
10   }
11
12   @LINK void inc(){
13     log("inc invocada.");
14     contador++;
```

```
  }

  @LINK void obtenerValor(OpFeedbackParam<Integer> v){
    log("obtenterValor invocada.");
    v.set(contador);
  }
}
```

Observen que a excepción de la etiqueta `@LINK` las operaciones de este artefacto se definen exactamente igual que aquellas que se definen en la interfaz del usuario con la etiqueta `@OPERATION`. De hecho, la operación `obtenerValor` usa un parámetro de salida `v`, como las operaciones de la interfaz del usuario. El artefacto que liga se define de la siguiente manera:

```
package tools;
import cartago.*;

@ARTIFACT_INFO(
  outports = { @OUTPORT(name="salida1") }
)

public class ArtefactoLigador extends Artifact{

  @OPERATION void test1() {
    log("Ejecutando test1.");
    try{
      execLinkedOp("salida1","inc");
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  @OPERATION void test2(OpFeedbackParam<Integer> v) {
    log("Ejecutado test2.");
    try{
      execLinkedOp("salida1","obtenerValor",v);
      log("Valor regresado: " + v.get());
    } catch (Exception ex){
      ex.printStackTrace();
    }
  }

  @OPERATION void test3() {
    log("Ejecutando test3.");
    try{
      ArtifactId id = makeArtifact("nuevoLigado",
        "tools.ArtefactoLigable",ArtifactConfig.DEFAULT_CONFIG);
      execLinkedOp(id,"inc");
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Las operaciones `test` y `test2` ejecutan respectivamente las operaciones de ligado `inc` y `obtenerValor` del artefacto ligado en el puerto de salida `salida1`. La operación `test3` crea un artefacto y sin mediar ligado explícito, ejecuta la operación de ligado `inc` de este aparato mediante su identificador

lógico guardado en `id`. Los puertos de salida `@OUTPORT` se declaran en la anotación `@ARTIFACT_INFO` de la clase `Artifact`, mediante el atributo `outports`.

La semántica de ejecución de las operaciones en la interfaz de ligado es la misma que la de las operaciones en la interfaz del usuario. La primitiva `execOpLinked` suspende la ejecución de la operación hasta que la ejecución de la operación del artefacto ligado ha sido completada. El agente que usa estos artefactos se define como sigue:

```
// Agent agente in project cartagoLigado.mas2j

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start <-
  makeArtifact("miArtefacto","tools.ArtefactoLigador",[],Id1);
  makeArtifact("contador","tools.ArtefactoLigable",[],Id2);
  linkArtifacts(Id1,"salida1",Id2);
  println("Artefactos ligados: procede prueba.");
  test1;
  test2(V);
  println("El valor regresado es: ",V);
  test3.
```

Observen que los parámetros de `linkArtifacts` incluyen el identificador del artefacto que liga, el identificador de su puerto de salida y el identificador del artefacto que ha sido ligado. La salida del SMA es como sigue:

```
  [agente] Artefactos ligados: procede prueba.
  [miArtefacto] Ejecutando test.
  [contador] inc invocada.
  [miArtefacto] Ejecutado test2.
  [contador] obtenterValor invocada.
  [miArtefacto] Valor regresado: 1
  [agente] El valor regresado es: 1
  [miArtefacto] Ejecutando test3.
  [nuevoLigado] inc invocada.
```

7.5.10 Multiples Espacios de Trabajo

En el siguiente ejemplo un agente creará dos espacios de trabajo, se moverá a ambos e imprimirá mensajes usando diferentes artefactos de tipo consola. También se ilustrará la configuración de los espacios de trabajo usando acciones internas. El SMA es como sigue:

```
MAS cartago09 {

  infrastructure: Centralised

  environment: jaca.CartagoEnvironment

  agents:
    agente agentArchClass jaca.CAgentArch;
```

```
9
10    aslSourcePath:
11      "src/asl";
12  }
```

El artefacto usado en este ambiente es de tipo contador, como el introducido en la sección 7.5.2 (página 171). El código del agente es como sigue:

```
1  // Agent agente in project cartagoWS.mas2j
2
3  /* Initial goals */
4
5  !start.
6
7  /* Plans */
8
9  +!start <-
10     ?joined(_,WSId0);
11     println("Espacio de trabajo actual: ",WSId0);
12     println("Creando nuevos espacios de trabajo...");
13     createWorkspace(ws1);
14     createWorkspace(ws2);
15     joinWorkspace(ws1,WSId1);
16     ?joined(_,WSId1);
17     println("Hola, ahora en el espacio ",WSId1);
18     makeArtifact("miContador","tools.Contador",[],ArtId);
19     focus(ArtId);
20     joinWorkspace(ws2,WSId2);
21     ?joined(_,WSId2);
22     println("Hola, ahora en el espacio ",WSId2);
23     println("Usando un artefacto de otro espacio de trabajo...");
24     inc[artifact_id(ArtId)];
25     joinWorkspace(ws1,WSId1);
26     println("Hola, de nuevo en ",WSName1);
27     println("Usando artefacto en el espacio actual...");
28     inc[artifact_id(ArtId)];
29     println("Saliendo...");
30     quitWorkspace;
31     ?joined(WSName,_);
32     println("De regreso en ",WSName);
33     joinWorkspace(ws1,_);
34     ?joined(ws1,_);
35     println("... y finalmente en el espacio ",ws1," otra vez.").
36
37  +valor(V) <- println("Valor actualizado: ",V).
38
39  +tick <- println("Ouch").
```

Los agentes pueden crear, unirse a y trabajar en multiples espacios de trabajo a un tiempo dado. Sin embargo, siempre hay un espacio de trabajo **actual** al cual se direccional las acciones que no están etiquetadas con el identificador lógico de un artefacto o espacio de trabajo. El espacio de trabajo actual puede conocerse usando la acción `current_wsp`. La acción interna `cartago.set_current_wsp(IdEspacio)` permite especificar el espacio de trabajo actual mediante su identificador lógico.

Las acciones sobre espacios de trabajo incluyen `createWorkspace` para crear nuevos espacios de trabajo en el nodo actual, provisto por el `NodeArtifact`, que también provee `joinWorkspace` para unirse a un espacio de trabajo. La

acción `quitWorkpace`, provista por `WorkSpaceArtifact` permite al agente salir de un espacio de trabajo. La salida del SMA es:

```
[agente] Espacio de trabajo actual: default
[agente] Creando nuevos espacios de trabajo...
[agente] Hola, ahora en el espacio miEspacioTrabajo1
[agente] Valor actualizado: 0
[agente] Hola, ahora en el espacio miEspacioTrabajo2
[agente] Usando un artefacto de otro espacio de trabajo...
[agente] Ouch
[agente] Hola, de nuevo en miEspacioTrabajo1
[agente] Valor actualizado: 1
[agente] Usando artefacto en el espacio actual...
[agente] Ouch
[agente] Valor actualizado: 2
[agente] Saliendo...
[agente] De regreso en miEspacioTrabajo1
[agente] ... y finalmente en el espacio default otra vez.
```

7.5.11 Espacios de Trabajo Distribuidos

Los agentes pueden unirse a espacios de trabajo que están alojados en nodos remotos, mediante la acción `joinRemoteWorkspace` (provista por `NodeArtifact`). Tan pronto como esta operación tiene éxito, la interacción con el espacio remoto es igual que con un espacio local.

En el siguiente ejemplo, un agente Jason se une a un espacio de trabajo `default` de un nodo CArtAgO corriendo en el *localhost*. El siguiente programa Java instala el nodo y lo hace accesible para los agentes remotos:

```
import cartago.*;
import cartago.util.BasicLogger;

public class EspacioRemoto {

  public static void main(String[] args) throws Exception {
    CartagoService.startNode();
    CartagoService.installInfrastructureLayer("default");
    CartagoService.startInfrastructureService("default");
    CartagoService.registerLogger("default", new BasicLogger());
    System.out.println("Nodo CArtAgO listo.");
  }
}
```

Recuerden que para compilar y ejecutar este código es necesario incluir en el *classpath* la ruta al código de CArtAgO. Por ejemplo:

```
tanteJulie:ej10 aguerra$ javac -classpath /Applications/Jason-1.4.2/lib/cartago.jar EspacioRemoto.java
tanteJulie:ej10 aguerra$ java -classpath /Applications/Jason-1.4.2/lib/cartago.jar:. EspacioRemoto
  Nodo CArtAgO listo.
```

En consola tenemos el aviso de que el nodo CArtAgO está listo y ahora los agentes que creemos podrían unirse a éste. El SMA para este caso es como sigue:

```
MAS cartago10 {

  infrastructure: Centralised

```

```
  environment: jaca.CartagoEnvironment

  agents:
    agente agentClass jaca.CAgentArch;
  aslSourcePath:
    "src/asl";
}
```

El agente en este SMA se define así:

```
// Agent agente in project cartagoWS.mas2j

/* Initial goals */

!start.

/* Plans */

+!start
  <-  ?current_wsp(Id,_,_);
      +default_wsp(Id);
    println("Usando espacio local...");
    makeArtifact("c1","c4j.Contador",[],Id1);
      focus(Id1);
      inc; inc;
      println("Probando espacio remoto...");
      joinRemoteWorkspace("default","tanteJulie.lan",IdWSRemoto);
      ?current_wsp(_,NombreWS,_);
      println("Hola Mundo remoto", NombreWS);
      !usarWSRemoto;
      quitWorkspace.

+!usarWSRemoto
  <-  makeArtifact("c0","c4j.Contador",[],Id0);
      focus(Id0); inc; inc.

+valor(V)
  <-  ?default_wsp(Id);
      println("Valor actualizado: ",V)[wsp_id(Id)].

-!usarWSRemoto [makeArtifactFailure("artifact_already_present",_)]
  <-  ?default_wsp(Id);
      println("Artefacto previamente creado ") [wsp_id(Id)];
      lookupArtifact("c0",Id0);
      focus(Id0); inc.
```

La ejecución de este SMA tiene dos efectos, por un lado en *shell* del sistema operativo que ejecuta `EspacioRemoto.class` la llegada del agente se registra, al igual que sus demás actividades en el nodo remoto:

```
[Basic logger ] At 1444263924405 agent agente joined the workspace.
    [Basic logger ] At 1444263924455 operation ( println Hola Mundo remoto default ) requested in artifact
    [Basic logger ] At 1444263924456 operation ( println Hola Mundo remoto default ) started in artifact c
    [agente] Hola Mundo remotodefault
    [Basic logger ] At 1444263924457 operation ( println Hola Mundo remoto default ) completed in artifact
    [Basic logger ] At 1444263924493 operation ( makeArtifact c0 c4j.Contador [Ljava.lang.Object;@67811c6
    [Basic logger ] At 1444263924493 operation ( makeArtifact c0 c4j.Contador [Ljava.lang.Object;@67811c6
    [Basic logger ] At 1444263924495 new percept generated about artifact c0 added properties: [Lcartago.A
    [Basic logger ] At 1444263924496 artifact c0 type: c4j.Contador has been created by agente
    [Basic logger ] At 1444263924496 new percept generated about artifact workspace added properties: [Lca
    [Basic logger ] At 1444263924496 operation ( makeArtifact c0
```

```
    c4j.Contador [Ljava.lang.Object;@67811c6f
    cartago.OpFeedbackParam@5b20f39 ) completed in artifact workspace
    ...
```

Y en la consola del SMA:

```
[agente] Probando espacio remoto...
Looking for rmi://localhost/cartago_node
[CartagoService] ADDED NODE INFO: localhost
[agente] Valor actualizado: 0
[agente] Valor actualizado: 1
[agente] Valor actualizado: 2
```

Puesto que el nodo remoto no se está ejecutando desde *Jason* es necesario incluir `Contador.class` en un subdirectorio `c4j` ya que de otra forma, el agente no encontrará el tipo de artefacto `Contador` cuando intente ejecutarlo en el nodo remoto.

7.6 LECTURAS Y EJERCICIOS SUGERIDOS

Ejercicios sugeridos

Ejercicio 7.1. *Implemente un conjunto de artefactos para llevar a cabo aprendizaje distribuido con el algoritmo J48. Reutilize el código de Weka para implementar J48. La idea central es que los ejemplos de entrenamiento están distribuidos de manera equilibrada (distribución de clase igual al conjunto de entrenamiento original) entre un grupo de agentes que aprenderán de manera colaborativa. Una primera aproximación es que un agente aprenda con su conjunto de entrenamiento y revise el modelo aprendido con los contra ejemplos que los otros agentes puedan aportar.*

Ejercicio 7.2. *Implemente un conjunto de artefactos para evaluar el sistema implementado en el punto anterior. Estos deben incluir: interfaz gráfica, validación cruzada y hold-out, por lo menos.*